Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud

Jiannan Ouyang
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
ouyang@cs.pitt.edu

John R. Lange
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
jacklange@cs.pitt.edu

Abstract

When executing inside a virtual machine environment, OS level synchronization primitives are faced with significant challenges due to the scheduling behavior of the underlying virtual machine monitor. Operations that are ensured to last only a short amount of time on real hardware, are capable of taking considerably longer when running virtualized. This change in assumptions has significant impact when an OS is executing inside a critical region that is protected by a spinlock. The interaction between OS level spinlocks and VMM scheduling is known as the *Lock Holder Preemption* problem and has a significant impact on overall VM performance. However, with the use of ticket locks instead of generic spinlocks, virtual environments must also contend with waiters being preempted before they are able to acquire the lock. This has the effect of blocking access to a lock, even if the lock itself is available. We identify this scenario as the *Lock Waiter Preemption* problem. In order to solve both problems we introduce Preemptable Ticket spinlocks, a new locking primitive that is designed to enable a VM to always make forward progress by relaxing the ordering guarantees offered by ticket locks. We show that the use of Preemptable Ticket spinlocks improves VM performance by $5.32X$ on average, when running on a non paravirtual VMM, and by $7.91X$ when running on a VMM that supports a paravirtual locking interface, when executing a set of microbenchmarks as well as a realistic e-commerce benchmark.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Mutual exclusion

Keywords Virtual Machines; Lock Holder Preemption; Paravirtualization

1. Introduction

Synchronization has long been recognized as a source of bottlenecks in SMP and multicore operating systems. With the increased use of virtualization, multi-core CPUs, and consolidated Infrastructure as a Service (IaaS) clouds this issue has become more significant due to the *Lock Holder Preemption* problem [15]. Lock holder preemption occurs whenever a virtual machine's (VM's) virtual CPU (vCPU) is scheduled off of a physical CPU while a lock is held inside the VM's context. The result is that when the VM's other vCPUs are attempting to acquire the lock they must wait until the vCPU holding the lock is scheduled back in by the VMM so it can release the lock. As kernel level synchronization is most often accomplished using spinlocks, the time spent waiting on a lock is wasted in a busy loop. While numerous attempts have been made to address this problem, the solutions have targeted only generic spinlock behaviors and not more advanced locking primitives such as ticket spinlocks (spinlocks that ensure consistent ordering of acquisitions). As a result of the introduction of ticket spinlocks virtual machine synchronization now must contend not only with Lock Holder Preemption but also *Lock Waiter Preemption*.

Ticket spinlocks [9] are a form of spinlock that enforces ordering among lock acquisitions. Whenever a thread of execution attempts to acquire a ticket spinlock it either (1) acquires the lock immediately, or (2) is granted a ticket which determines the order among all outstanding lock requests. The introduction of ticket spinlocks was meant to ensure fairness and prevent starvation among competing threads by preventing any single thread from obtaining a lock before another thread that requested it first. In this manner each thread must wait to acquire a lock until after it has been held by every other thread that previously tried to acquire it. This ensures that a given thread is never preempted by another thread while trying to acquire the same lock, and thus guarantees that well behaved threads will all acquire the lock in a timely manner.

While ticket spinlocks have been shown to provide advantages to performance and consistency for native OS environments, they pose a new challenge for virtualized environments. This is due to the fact that when running inside a VM, the use of a ticket spinlock can result in multiple threads waiting to acquire a spinlock that is currently available. This problem exists whenever a VMM preempts a waiter that has not yet acquired the lock. In this case even if the lock is released, no other thread is allowed to acquire it until the next waiter is allowed to run, resulting in a scenario where there is contention over an idle resource. We denote this situation as the *Lock Waiter Preemption* problem.

Lock holder preemption has traditionally been addressed using a combination of configuration, software and hardware techniques. Initial workarounds to the lock holder preemption problem required that every vCPU belonging to a given VM be gang scheduled in order to avoid this problem altogether [16]. While this eliminates the lock holder preemption problem, it does so in a way that dramatically reduces the amount of possible consolidation and increases the amount of cross VM interference. As a result of these drawbacks, several attempts have been made to address the lock holder

VEE'13, March 16–17, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1266-0/13/03. . . $15.00

preemption problem on a per vCPU level in order to move away from the gang scheduling model. These approaches focus on detecting when a given vCPU is stuck spinning on a busy lock, so that the VMM can adjust its scheduling decisions based on the lock dependency. The detection techniques vary, but can be roughly classified by where they are implemented: inside the VMM (relying on hardware virtualization features such as Pause Loop Exiting), or inside both VMM and guest OS (paravirtual). Unfortunately, while these approaches have been effective in addressing the *lock holder preemption* problem they are unable to handle the *lock waiter preemption* problem.

We propose to address the problem of lock waiter preemption through the introduction of a new spinlock primitive called Preemptable Ticket spinlocks. Preemptable Ticket spinlocks improve the performance of traditional ticket spinlocks by allowing preemption of a waiter that has been detected to be unresponsive. Unresponsiveness is determined via a linearly increasing timeout that allows earlier waiters a window of opportunity in which they can acquire a lock before the lock is offered to later waiters. Preemptable Ticket spinlocks provide performance benefits when running either with or without VMM support (a specialized paravirtual interface), however VMM support does provide overall superior performance.

Preemptable Ticket spinlocks are based on the observation that forward progress is preferable to fairness in the face of contention. In the case where lock waiter preemption is preventing a guest from acquiring a lock, then a thread waiting on the lock should be able to preempt the next thread in line if that thread is incapable of acquiring the lock in a reasonable amount of time. While Preemptable Ticket spinlocks do allow preemption, which technically breaks the ordering guarantees of standard ticket locks, it does so in a way that minimizes the loss of fairness by always granting priority to earlier lock waiters. Priority is granted via a time based window which gradually increases the number of ticket values capable of acquiring the lock. The choice of a time based window is based on the observation that VMM level preemption typically results in large periods of unresponsiveness, while other causes of unresponsiveness typically result in periods orders of magnitude smaller. This means that a timeout based detection approach can be implemented with high accuracy and relatively low overhead. The use of timeouts also allows the implementation of linearly expanding exclusivity windows, which ensure that if an earlier ticket holder is able to acquire a lock it will do so before a later ticket holder is offered the chance. In this way early ticket holders are only preempted if they are inactive for a long period of time, almost always as the result of the vCPU being preempted by the VMM.

In this paper we make the following contributions:

- Identify the lock waiter preemption problem and quantify its effects on VM performance
- Propose Preemptable Ticket spinlocks as an alternative spinlock primitive to address the lock waiter preemption problem
- Describe the implementation of Preemptable Ticket spinlocks inside the Linux kernel and KVM VMM
- Evaluate the performance of Preemptable Ticket spinlocks over a set of micro and macro level benchmarks

The rest of the paper is organized as follows. Section 2 briefly reviews previous works. In section 3 we present some background of the problem. Then in section 4 we introduce our solution, the Preemptable Ticket spinlock and discuss its implementation issues in section 5. We evaluate our proposed solution in section 6, and discuss possible future optimization in 7. Finally, we conclude in section 8.

2. Related Work

Due to the impact virtual machine scheduling has on synchronization performance, significant research efforts have sought to optimize the interaction between the guest OS and VMM. In general the existing approaches have fallen into the following categories.

Preemption Aware Scheduling Initially, VMM architectures dealt with guest locking problems by requiring that every VM be executed using co-scheduling [11]. This approach was adopted by the virtual machine scheduler in VMware ESX [16]. Advances on this approach were explored by [18, 19], that proposed an adaptive co-scheduling scheme, that allowed the VMM scheduler to dynamically alternate between co-scheduling and asynchronous scheduling of a VM's set of vCPUs. The choice of scheduling approaches is based on the detection of long lived lock contention in the guest OS. Finally, in [14] the authors proposed a "balancing scheduler" scheme that associates a VM's individual vCPUs with dedicated physical CPUs, but does not require that the vCPUs be co-scheduled. These solutions add host side scheduling constraints, and are complementary to Preemptable Ticket spinlock.

Paravirtual Locks Paravirtual approaches to solve the lock holder preemption problem was first explored in [15], where the authors adopted a lock avoidance approach in which the guest OS provided scheduler hints to the underlying VMM. These hints demarcated non-preemptable regions of guest execution that corresponded to critical sections in which a non-blocking lock was held. [6] proposed another paravirtual lock approach (later adopted by Xen and KVM [12]), that uses a loop counter to detect "unusually long" wait times for a spinlock. When the time spent waiting for a lock reaches a given threshold, the VMM is notified via a hypercall that a vCPU is currently blocked by a held lock. The VMM then halts the waiting vCPU until the lock is detected to be available. These solutions are capable of delivering good performance, however, paravirtual approaches require guest kernel modifications, leading to compatibility and standardization issues. Moreover, these approaches are also designed to handle generic spinlocks, and so do not take into account the behavior of ticket spinlocks and the resulting problem caused by lock waiter preemption.

Hardware enabled Pause-Loop Exiting Hardware based solutions to the lock holder preemption problem were introduced in [17]. In this work authors proposed an approach that relied on a spin detection buffer (SDB) that served to detect vCPUs spinning on a preempted lock. Similar hardware feature has already been adopted by both Intel's (Pause-Loop Exiting) and AMD's (Pause Filter) virtualization extensions. With these features enabled, hardware is able to detect spinning vCPUs by generating VM exits as a result of executing certain number of pause instructions during the the spinlocks' busy wait loops. Unfortunately, even with these features in place, it remains difficult for a VMM to accurately detect a preempted lock holder due to the lack of information resulting from the semantic gap [5].

Preemptable Adaptive Locks Alternative spinlock behaviors have been proposed in [7] that allow adaptive preemption of queue-based locks. This approach has been implemented as adding time publishing heuristic into queue based lock such as MCS lock [10], which requires each thread periodically records its current timestamp to a shared memory location. The preemptive feature of these locks allows a thread to be removed from the queue after a certain period of time. These locks are meant to gracefully fail in the face of preemption, allowing a thread to specify a timeout value that determines how long it is willing to wait to acquire a given lock. However publishing timestamp is too expensive to implement in kernel spinlock, and a lock that may fail do not directly attempt to solve either the lock holder or waiter preemption problem. Instead they provide a mechanism by which the programmer can react to preemption when it occurs, thus placing a greater burden on OS developers.

3. VM based OS synchronization

Among the challenges that virtualization poses to OS designers is fact that the underlying virtual hardware can be arbitrarily scheduled by the underlying VMM. This has serious consequences for timing sensitive operations in the guest OS that requires and assumes exclusive access to the underlying hardware as well as atomic execute. These regions are generally protected by disabling interrupts and acquiring a spinlock, based on the assumption that the operations will be short in duration and so won't result in long delays for other contending threads. Unfortunately, due to the semantic gap [5], a VMM is incapable of taking into consideration current lock state when it is scheduling the CPU amongst various vCPUs. The result is that occasionally a VM that is holding a lock for what should be a short period of time is scheduled out by the underlying VMM, resulting in an orders of magnitude increase in the duration of a critical region.

Lock Holder Preemption The scheduling out of a vCPU currently holding a lock is referred to as the Lock Holder Preemption problem. These situations often result in serious performance degradation, especially if the lock being held is one that is frequently acquired by other vCPUs in the system. In this case, each vCPU attempting to acquire the lock will enter into a busy wait loop and stall the entire vCPU until the VMM reschedules the lock holder for execution. While OS developers have introduced new synchronization primitives [8] that avoid some of these pitfalls, spinlocks remain as one of the primary synchronization primitives in modern operating systems. Previous work [6] has shown that up to 7.6% of guest execution time can be attributed to stalls due to lock holder preemption in generic spinlocks. And this problem get more severe under queue-based locks, up to 99.3% of guest execution time can be wasted on spinning.

Solving the lock holder preemption problem has been the focus of a number of different approaches looking to optimize performance for multicore VMs. While these approaches have focused on different techniques for actually handling a preempted lock holder, they have all relied on heuristic based detection of lock contention. In particular, they have focused their efforts on detecting when a thread begins to spin on a lock that is currently held by a preempted vCPU. The use of heuristics is necessary to avoid significant performance overheads introduced by more accurate sampling or monitoring approaches. Once a spinning vCPU has been detected it is up to the VMM to either schedule out the spinning vCPU or schedule in the vCPU currently holding the lock.

Ticket Spinlocks Ticket spinlocks are a relatively recent modification to the global spinlock architecture found in Linux. Introduced in kernel version 2.6.25, ticket spinlocks are designed to improve lock fairness and prevent starvation. Each ticket spinlock includes a "head" as well as a "tail" field indicating the current number of threads waiting for the lock. The lock is always granted to the next waiter in the queue, thus guaranteeing that locks are dispatched in FIFO order and no thread will ever experience starvation.

Lock Waiter Preemption Restricting lock acquisitions to a FIFO schedule expands the lock holder preemption problem by creating an environment where anyone with an earlier position in a lock's queue is effectively holding the lock as far as threads later in the queue are concerned. Thus, when executing inside a virtual machine environment, if a vCPU currently holding a ticket is preempted, all subsequent ticket holders must wait for the preempted vCPU to be rescheduled. This can result in execution being blocked by lock contention even when the lock in question is available. We call this problem Lock Waiter Preemption.

To determine the severity of the lock waiter preemption problem, we instrumented the Linux ticket lock implementation to profile VM preemptions during lock operations. Lock preemption was identified by detecting inordinately long wait times for a given lock, where long wait times were conservatively chosen to be 2048 iterations of the inner loop of a busy waiting spinlock. On our machine, 2048 iterations corresponded to roughly 1μs, an amount of time that exceeds the time a thread would spend holding a lock according to statistics [6]. Next we separated the lock waiter preemption scenarios from the set of detected preemptions, by checking whether the stalled lock was in fact available. To make this determination we modified the existing spinlock structure to include a `holder_id` variable that served as an indicator of lock availability. The value of `holder_id` was set to the thread id of a given lock holder on acquisition and cleared when the lock was released.

Table 1 includes the results of our analysis after running the hackbench [1] and ebizzy [2] benchmarks with 1 and 2 VMs. While the amount of detected preemption was low, previous work [6] has shown that even with a low rate of preemption, significant performance degradation can occur. Furthermore, as more VMs are deployed on the system, it is expected that preemption will increase. Column 2 shows the number of preemptions that occurred in the midst of lock operations during the benchmark's execution. Interestingly, as the number of VMs increased the number of preemptions declined, we surmise that this is due to decreasing VM performance due to the overcommitment of resources. Column 3 shows the number of preemptions in which lock acquisitions were delayed because of either lock holder or lock waiter preemption. While these delays were infrequent when compared to the total number of lock acquisitions in column 2, it should be noted that even a limited degree of preemption can cause significant performance degradation. Furthermore, while the total number of preemptions declined when additional VMs were added, the number of preemptions resulting in stalled lock acquisitions actually increased. More critically, the stalled lock operations were predominately due to a preempted lock waiter and *not* a preempted lock holder. The degree of the issue is shown more clearly in column 4, which provides the percentage of stalled lock acquisitions resulting from a preempted lock waiter. As can be seen, even when a physical machine is overcommitted by a factor of only 2, lock waiter preemption becomes the dominant source of synchronization overhead.

	N	$N_h + N_w$	N_w	$\frac{N_w}{N_h+N_w}$
hackbench x1	$1.11E8$	1089	452	41.5%
hackbench x2	$9.65E7$	44342	39221	88.5%
ebizzy x1	$2.86E8$	294	166	56.5%
ebizzy x2	$9.56E5$	1017	980	96.4%

Table 1. An analysis of the Lock Waiter Preemption Problem in the Linux Kernel. N is the number of lock acquisitions, while N_h and N_w represent the number of lock holder and lock waiter preemptions, respectively. $N_w/(N_h + N_w)$ shows the percentage of preemptions due to the lock waiter preemption problem.

4. Preemptable Ticket Spinlocks

In order to address the Lock Waiter Preemption problem, we introduce Preemptable Ticket spinlocks. Preemptable Ticket spinlocks are a hybrid spinlock architecture that combines the features of both ticket and generic spinlocks in order to preserve the fairness of ticket spinlocks while avoiding the Lock Waiter Preemption problem.

4.1 Approach

The intuition behind Preemptable Ticket spinlocks is that making forward progress is more important than ensuring fairness. Preemptable Ticket spinlocks leverage the advantages of both generic spinlocks and ticket locks in order to ensure fairness in the absence of preemption while also supporting out of order lock acquisition when the waiters in the queue are preempted. In these situations performance of a given lock waiter is degraded primarily by the VMM scheduler and not by the violation of the ordering of lock acquisitions. That is, a lock waiter can be preempted without perceptively adding to the waiter's execution time.

The primary goal of Preemptable Ticket spinlocks is to add adaptive preemptibility to ticket locks, while retaining the ordering guarantees as much as possible. This is done via the use of a *proportional timeout threshold* that determines the ability of a thread to acquire a lock based on that thread's position among the set of threads currently waiting on the lock. In Preemptable Ticket spinlocks, a thread can acquire a lock out-of-order *if* it has been waiting longer than its *timeout threshold*. We denote such a thread as a *timed out* waiter. The *timeout threshold* is calculated from a standard timeout period τ that is multiplied with the thread's current lock queue position index n as shown in the following equation,

$$timeout\_threshold = n \times \tau \qquad (1)$$

in which τ is a constant parameter of Preemptable Ticket spinlock.

To calculate the position index value n, two variables are maintained for each lock, (1) `num_request` indicates the total number of lock requests of a lock, and (2) `num_grant` indicates the total number of lock requests that have been granted. In addition, each thread has a local variable named `ticket`, which represents the queue position of the request. `num_request` and `num_grant` are maintained by each thread in a distributed fashion for each lock. When acquiring a lock, the current `num_request` value is stored into the thread's local `ticket` variable, and then atomically incremented by 1. Conversely, when releasing a lock, `num_grant` is atomically incremented by 1.

The location of a thread in a given lock's queue is denoted as the position index value n, and is calculated based on a thread's `ticket` value as well as the current value of `num_grant`,

$$n = ticket - num\_grant \qquad (2)$$

This position value indicates the number of waiters for a given lock before the current thread. Because `ticket` stores the number of outstanding lock requests at the request time, and `num_grant` contains the number of lock requests that have been granted, we can determine the number of pending requests before current thread (and thus the thread's queue position) as `(ticket - num_grant)`. Note that it is possible for a thread to have a negative position in the queue (`ticket` < `num_grant`). This can result whenever a lock has been preemptively acquired and the preempted core is then rescheduled at a later point in time. In this case a negative position indicates that later threads have violated the lock order, in which case the preempted thread should attempt to acquire the lock immediately.

With the described behavior, Preemptable Ticket spinlocks are able to preserve lock ordering in the absence of VM preemption, while also adapting to increased physical resource contention by allowing limited ordering violations. Figure 1 provides an illustrative example of the functionality of Preemptable Ticket spinlocks. The timeout threshold is indicated by the number above each node, and is set proportionally based on the node's position in the queue. In the initial stage (a), four nodes are waiting while $N1$ is holding the lock. At this point the vCPU hosting $N2$ is preempted by VMM. In the following stage (b), $N1$ releases the lock, causing the timeout threshold to be updated for each node. At this point the lock is available but no node can acquire it because the next waiter in the queue $N2$ is currently preempted. This is the lock waiter preemption problem. In stage (c) node $N3$ reaches the timeout threshold and acquires the lock out-of-order before $N2$. Finally at stage (d), $N3$ releases the lock, causing $N4$ to update it's timeout threshold. At this point, $N4$ has still not reached the timeout threshold, so $N2$ is able to immediately acquire the lock without contention.

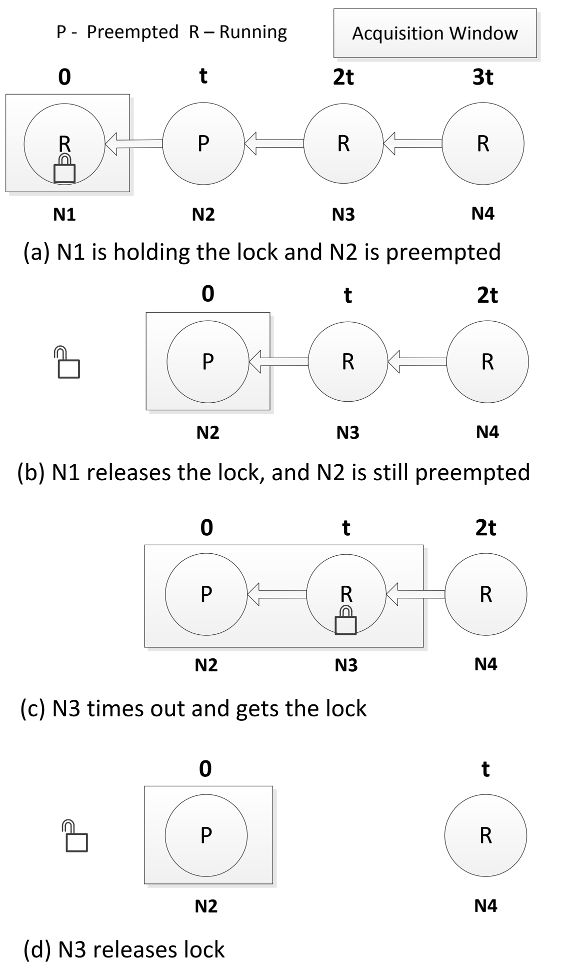

P - Preempted R – Running
Acquisition Window
0
t
2t
3t
R
P
R
R
N1
N2
N3
N4
(a) N1 is holding the lock and N2 is preempted
0
t
2t
P
R
R
N2
N3
N4
(b) N1 releases the lock, and N2 is still preempted
0
t
2t
P
R
R
N2
N3
N4
(c) N3 times out and gets the lock
0
t
P
R
N2
N4
(d) N3 releases lock

Figure 1. Preemptable Ticket Spinlock Illustration. R indicates a running vCPU, P means a vCPU is preempted. Nodes in acquisition window are timed out nodes and are able to acquire the lock in random order. On top of each node is its timeout threshold, and below is its node ID.

4.2 Preemption Adaptivity

Preemptable Ticket spinlocks are a hybrid lock algorithm that combines the benefits of generic spinlocks and ticket locks by relaxing the ordering guarantees provided by ticket locks. The underlying feature of Preemptable Ticket spinlocks is a timeout threshold that controls when a given waiter can acquire the lock in a random order. The timeout threshold is derived from a tunable constant denoted as τ, combined with a waiter's queue position index n. The behavior of a Preemptable Ticket spinlock can be tuned to match the behavior of either a generic spinlock, a ticket lock, or a combination of the two depending on the value assigned to τ. The following equation shows the behavior of Preemptable Ticket spinlock for different values of τ.

$$lock = \begin{cases} spinlock & \tau = 0 \\ preemtable \quad ticket \quad spinlock & 0 < \tau < \infty \\ ticket \quad lock & \tau = \infty \end{cases}$$

A τ value of 0 results in an immediate timeout that mimics the behavior of a generic spinlock, while setting $\tau = \infty$ will prevent a timeout from ever occurring and so generate the strict ordering behavior of a standard ticket lock. Preemptable Ticket spinlocks are thus able to tune their behavior by trading off between aggressiveness and fairness depending on the state of the system and the behavior of the underlying VMM scheduler.

A well chosen τ value can provide both good performance and fairness. Fairness is ensured when τ is large enough that lock waiters will not time out prematurely. Performance is ensured when τ is small enough that a lock waiter is able to promptly detect when an earlier waiter is preempted. According to previous work [6], the lock holding time and preemption time in fact differ by orders of magnitude. Typically lock holding time is less than $1\mu s$ while the time between a vCPU's preemption and rescheduling is at least $1ms$. Thus we choose a value of τ that is slightly larger than typical lock holding time, $\sim 2\mu s$ for our implementation.

4.3 Fairness

Locking *fairness* relates to the variance in wait times that threads experience while trying to acquire a lock. A truly fair lock implementation should result in a variance near 0, that is every thread waits the same amount of time to acquire a lock. The standard technique for achieving fairness is to ensure that locks are granted in the same order in which the requests were made, FIFO ordering. With generic spinlocks all waiters have an equal chance of acquiring a lock, regardless of when the waiter first requested it. This makes generic spinlocks an "unfair" locking implementation. Ticket spinlocks implement strict ordering that enforced via the use of tickets assigned consecutively to new lock requests. An earlier waiter with smaller ticket value always get the lock before a waiter that requested the lock at a later point in time.

In contrast to these locking behaviors, Preemptable Ticket spinlocks ensure that,

- For all waiters yet to reach their timeout threshold, strict ordering is preserved
- Waiters that have reached their timeout threshold have priority over those who have not
- All waiters that have reached their timeout threshold have equal priority among themselves

The first point holds because the timeout threshold is proportional to a waiter's queue position. Earlier waiters have smaller thresholds, and thus they time out earlier than those who are later in the queue. The second point holds because according to equation 2, the position index of a non-timed-out waiter is larger or equal than the number of timed out waiters. Thus all non-timed-out waiters have timeout thresholds no less than $x \times \tau$, where x is the number of waiters who have passed their timeout threshold. With a proper value of τ the threshold is long enough for every timed out waiter to complete their critical sections in the absence of preemption. In other words, every thread that has reached its timeout threshold will have time to acquire and release the lock *before* the next waiter times out. This ensures that priority is given to a preempted waiter immediately after it is rescheduled, and furthermore all non-timed-out waiters will wait until every thread that has timed out has acquired the lock. Thus while ordering is violated, the violations are minimized to those vCPUs that have been preempted by the VMM.

Based on the description above, it is straight forward to show that the number of ordering violations experienced by a given lock is bounded by the number of vCPUs assigned to a VM. Furthermore, the probability that a preempted waiter is unable to immediately acquire the lock after rescheduling is given by $P(x) = x/R$, where x is the number of lock acquisitions that have occurred since a lock waiter was preempted, and R is the number of outstanding waiters that have been preempted. Thus for a simple case, where a set of waiters are preempted and rescheduled simultaneously, we can derive the cumulative density function shown in figure 2. From this we can see that the probability that a preempted waiter has acquired the lock increases linearly based on the number of waiters that were simultaneously preempted by the VMM. The number of ordering violations is limited by the number of lock waiters preempted by the VMM at any given point at time, and in the worst case is bounded to the number of active vCPUs assigned to the VM. While more dynamic scheduling cases will alter the shape of the CDF, they will not change the worst case bounds.

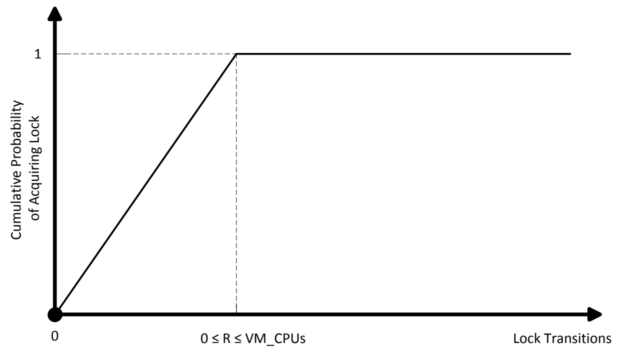


Figure 2. Cumulative Density Function showing the probability of having acquired a lock after being rescheduled following a preemption. The horizontal axis represents the number of lock acquisitions necessary before the preempted waiter is able to acquire the lock. R is the number of preempted waiters currently contending for the lock. VM_CPUS is the number of vCPUs assigned to the VM.

Note that the above discussion only holds for a lock waiter preemption case in which the time spent holding a lock is less than the base timeout value τ. While our approach is effective in addressing the lock waiter preemption problem, it is important to note that it does not address lock holder preemption. In the case of lock holder preemption, all lock waiters will time out due to the fact that preemption time is considerably larger than the timeout threshold. Lock holder preemption represents the worst case in regards to fairness, in that every waiter in the queue will reach its timeout threshold and compete equally for the lock. In this case our approach will degenerate to a generic spinlock behavior. However this behavior will still be bounded by the number of vCPUs. Furthermore, we believe that the case of lock holder preemption will be relatively low based on our earlier results in Table 1.

4.4 Host Independence

Unlike previous solutions [6, 12–15, 17–19], Preemptable Ticket spinlocks work in the absence of any VMM side support, which makes it a solution where host side modifications such as paravirtualization are not feasible. Preemptable Ticket spinlocks can be implemented entirely inside a guest OS and are capable of detecting lock waiter preemption adaptively based on a single timeout directly measurable by the guest. However, while Preemptable Ticket spinlocks are capable of operating independently without VMM support, it is possible to further improve their performance by combining them with existing lock holder preemption solutions. This is important because while Preemptable Ticket spinlocks address lock waiter preemption they actually increase the likelihood of lock holder preemption due to the fact that they allow the lock to be acquired more often. We have investigated the integration of Preemptable Ticket spinlocks with existing solutions to the lock holder preemption problem, and provide results of this integration in our evaluation.

5. Implementing Preemptable Ticket Spinlocks in Linux/KVM

In order to evaluate the efficacy of Preemptable Ticket spinlocks, we have implemented them inside version 3.5.0 of the Linux kernel. Our implementation acts as a drop in replacement for the standard ticket spinlock implementation currently supported by the kernel. The implementation consisted of only $\sim$60 lines of C and assembly code, and consists of a modified spinlock datatype as well as modifications to the lock, unlock, islocked and trylock operations. The implementation resides entirely in the guest kernel and does not require any additional VMM side support in order to function correctly. While Preemptable Ticket spinlocks are capable of functioning fully on top of any virtualization environment, they are also able to benefit from extended paravirtual operations, which we will discuss later.

5.1 Preemptable Ticket

Figures 3 shows the implementation of *lock* and *unlock* operations. As part of our modifications we added code to maintain the timeout threshold, while also changing the semantics of some of the existing data fields. The existing kernel spinlock data structure contains variables to track the head and tail of a queue in order to implement the proper ticket semantics. These fields are equivalent to the `num_request` and `num_grant` fields we discussed in section 4.1. In order to detect a preempted lock waiter we have added another field named `lock` which indicates the availability of the lock.

In the lock function, we declare a local struct `inc` which acts as a local copy of the `head` and `tail` values, `timeout` which is used as the timeout threshold, and `current_head` which is another local copy of `head` used to detect changes to the value of `head`. At line 7 the code atomically updates `inc` in order to increase the value of `head`. At this point `inc.tail` is regarded as the “ticket” of the current thread. Line 10-11 shows the fast path, which handles the case of an uncontended lock acquisition.

Lines 13-27 implement the core of the proportional timeout functionality. The timeout threshold is initialized in line 14, and updated in line 23 whenever `head`’s value changes. This ensures that the timeout threshold is always proportional to the number of pending lock requests that arrived previously, according to equation 2. A timed out thread will break out of the loop at line 27. A thread should also break out of the loop when it’s ticket is equal to the current head, meaning that it is due to acquire the lock based on the ticket ordering. Besides, a thread breaks out of the loop if it’s ticket is less than the current value of `head`, which can result from a preemption followed by a rescheduling as discussed in Section 4.1. Finally, line 29–35 implement a generic spinlock which is invoked by every thread that is allowed past the wait loop.

```
1  #define TIMEOUT_UNIT (1<<14)
2  void __ticket_spin_lock(arch_spinlock_t *lock)
3  {
4    register struct __raw_tickets inc={.tail=1};
5    unsigned int timeout = 0;
6    __ticket_t current_head;
7    inc = xadd(&lock->tickets,inc);
8
9    // fast path
10   if (likely(inc.head == inc.tail))
11     goto spin;
12
13   // wait in queue
14   timeout = TIMEOUT_UNIT
15   * (inc.tail - inc.head);
16   do {
17     current_head =
18         ACCESS_ONCE(lock->tickets.head);
19     if (inc.tail <= current_head) {
20       goto spin;
21     } else if (inc.head != current_head) {
22       inc.head = current_head;
23       timeout =  TIMEOUT_UNIT
24           * (inc.tail - inc.head);
25     }
26     cpu_relax();
27   } while (timeout--);
28
29 spin:
30   for (;;) {
31     if (xchg(&lock->lock, 1) == 0)
32       goto out;
33     cpu_relax();
34   }
35 out: barrier();
36 }
37
38 void __ticket_spin_unlock(arch_spinlock_t *
       lock) {
39   __add(&lock->tickets.head, 1,
         UNLOCK_LOCK_PREFIX);
40   xchg(&lock->lock, 0);
41 }
```

Figure 3. Kernel Implementation: lock and unlock

The unlock operation is relatively simple, and is implemented by combining the unlock operations of both ticket and generic spinlocks. The operation atomically increments `head` by 1 and clears the `lock` value.

While the lock and unlock operations provide the necessary functionality for basic locking, the Linux kernel also requires additional locking semantics for certain cases. In particular Linux makes consistent use of other spinlock primitives such as islocked and trylock. In order to fully support Preemptable Ticket spinlocks through the kernel, we had to modify these operations as well. Figure 4 shows the implementation of these primitives. At line 3 our code modifications return true if it detects the presence of earlier waiters for the lock or if the lock is currently not available. The trylock operation attempts to acquire a given lock, but immediately returns 0 if the lock is not available. In order to support Preemptable Ticket spinlocks we modified the implementation at lines 18-19, where we added an atomic check to determine whether the lock is free and if there are no earlier waiters. Other than these mini-

```
int __ticket_spin_is_locked(arch_spinlock_t *
    lock) {
  struct __raw_tickets tmp = ACCESS_ONCE(lock
      ->tickets);
  return (tmp.tail != tmp.head) || (
      ACCESS_ONCE(lock->lock)==1);
}

int __ticket_spin_trylock(
        arch_spinlock_t *lock) {
  arch_spinlock_t old, new;
  *(u64 *)&old = ACCESS_ONCE(*(u64 *)lock);
  if (old.tickets.head != old.tickets.tail)
    return 0;
  if (ACCESS_ONCE(lock->lock) == 1)
    return 0;
  new.head_tail = old.head_tail +
      (1 << TICKET_SHIFT);
  new.lock = 1;
  /* cmpxchg is a full barrier */
  if (cmpxchg((u64 *)lock, *(u64 *)&old,
        *(u64 *)&new) == *(u64 *)&old) {
    return 1;
  } else return 0;
}
```

Figure 4. Kernel Implementation: islocked and trylock

mal changes, the existing implementations were left as originally written.

5.2 Paravirtual Preemptable Ticket Spinlock

In addition to the fully encapsulated Preemptable Ticket spinlock implementation, we also implemented a paravirtual version based on a paravirtual ticket lock patch submitted to the Linux Kernel Mailing List (LKML) [12] on May 2, 2012. It includes both guest and host side modifications, which we adopted and extended to support Preemptable Ticket spinlocks.

The paravirtual interface includes the ability to capture `halt` instructions from the guest vCPU. These instructions are emulated by switching the halting vCPU to a sleep state until a special hypercall is received to wake it up. This interface allows a guest OS to notify the VMM when it is appropriate to place a vCPU into a sleep state and when to wake it up via a hypercall invocation. The purpose of this interface is to allow a guest to place a lock waiter vCPU into sleep state on the host since it is unable to make forward progress due to a preempted lock holder.

In the original implementation, the `halt` instruction is executed whenever a thread reaches a timeout threshold (2048 iterations of a spinlock by default). As soon as the lock is released, the next waiter is woken up using a hypercall. This approach essentially converts a busy wait lock into a blocking lock, and prevents a spinning vCPU from wasting a significant amount of time spinning on an unavailable lock.

Our paravirtual Preemptable Ticket spinlock implementation also executes `halt` after spinning on the `lock` variable longer than a threshold (2048 iterations in this paper). However, when releasing the lock, a wakeup hypercall is sent for every sleeping vCPU instead of only the next thread in the queue. Because Preemptable Ticket spinlocks allow out-of-order lock acquisition, an in-order wake up can actually cause a deadlock scenario when $\texttt{ticket} < \texttt{num\_grant}$.

While we have implemented a paravirtual version of Preemptable Ticket spinlock, it is important to note that they are designed to function correctly with either full system or paravirtual VMM architectures. As we will show, Preemptable Ticket spinlocks provide performance benefits when used with either environment. The rationale for a non-paravirtual locking implementation is that while paravirtual interfaces do provide benefits to performance and information sharing, they are not always portable and can introduce compatibility issues across different VMMs as well as different versions of the same VMM. Preemptable Ticket spinlocks are capable of functioning on top of any unmodified or paravirtual VMM architecture.

6. Evaluation

In this section, we empirically evaluate how *ticket locks*, paravirtual ticket locks (*pv-lock*), and paravirtual preemptable ticket locks (*pv-preemptable-lock*) improve application performance when running in a VM on either a full system or paravirtual VMM architecture. Our evaluation uses a combination of microbenchmarks as well as a real world workload based on the Dell DVD Store [4] benchmark.

6.1 Experimental Setup

Each experiment was run on a single Dell Optiplex with an 8 core 2.6 GHz Intel Core i7 CPU, 8 GB of RAM, and a 1 Gbit NIC. The experiments were all executed inside an 8 core VM image configured to use 1GB RAM. A Fedora 17 environment was used for both the host and guests, and was configured to use a modified version of the Linux kernel based on version 3.5.0. In order to conduct a fair evaluation, we implemented *pv-lock* and *pv-preemptable-lock* in otherwise identical configurations of the 3.5.0 kernel, we also include results that compare the various spinlock implementations against the stock kernel implementation.

For the evaluation we selected benchmarks that focus on CPU intensive, memory intensive and I/O intensive workloads. These benchmarks include three microbenchmarks (ebizzy, hackbench, and kernbench) as well as a real world web application benchmark (the Dell DVD store).

Ebizzy [2] is designed to generate a workload that resembles a common web application server. It is highly threaded, has a large in-memory working set size, and allocates and deallocates memory frequently. We execute ebizzy 5 times using 16 threads for each run, and performance is measured as the sustained throughput (records/second).

Hackbench [1] is a multi-threaded program that exercises Unix-socket (or pipe) performance. We execute hackbench 5 times using 4 threads with 10,000 loops. Performance is measured based on the completion time (seconds).

Kernbench [3] executes parallel kernel compilations using a variable degree of parallelization of the compilation process. Kernbench was executed 3 times and configured to use 8 compilation processes in order to saturate the vCPUs of an 8 core VM. Performance was measured based on the completion time (seconds).

Dell DVD Stores [4] is an open source simulation of an online e-commerce site. The benchmark interfaces an Apache website with a MySQL database running in the same VM. For our evaluation we configured a single client machine to emulate 32 independent clients each issuing search requests for 3 minutes following 1 minute warmup period. Performance is determined based on the transaction throughput (operations per minute) observed by the client.

In order to evaluate lock performance under realistic cloud scenarios we overcommitted the physical resources to a set of VMs all running the same benchmark. To simplify our evaluation we recorded the performance of a single VM randomly selected from the set.

In addition to evaluating different guest locking implementations, we also evaluated each guest lock implementation when running on both a paravirtual and full system VMM environment.

These results are meant to demonstrate the portability of the approaches, and determine how well they will perform in both optimized and non-optimized environments.

6.2 Microbenchmarks

Figure 5 shows the experimental results of the three microbenchmarks for each locking implementation. As expected, performance degrades as the number of competing VMs increases, however the degree of degradation depends on the choice of locking behavior. As cloud providers seek to maximize utilization by increasing consolidation as much as possible, the ability to sustain performance in the face of competing workloads becomes critical. As time spent waiting for a lock is wasted from the point of view of the resource provider, we try to measure the degree to which the different locking approaches can minimize the overheads due to lock contention.

Figures 5(a) shows results of each locking approach on *hackbench*, which mainly exercises the IPC subsystem. While each locking implementation has comparable performance in the single VM case, those designed to handle preemption are significantly better when executing in an overcommitted environment. In the two VM case, the speedups of *preemptable-lock*, *pv-lock* and *pv-preemptable-lock* are $3.89X$, $4.80X$ and $4.97X$ respectively, indicating that (1) Preemptable Ticket spinlocks significantly improve lock performance under overcommitted configurations without any host side support, (2) host side paravirtual interfaces improve lock performance further, (3) *pv-preemptable-lock* performs even better than *pv-lock* because it addresses both lock holder and waiter preemption. This trend is more obvious in the three VM case, where the speedups of *preemptable-lock*, *pv-lock* and *pv-preemptable-lock* are $9.63X$, $13.68X$ and $15.35X$ respectively. Note that less than 6% of Preemptable Ticket spinlock overhead can be observed in one VM case. This is due to the overhead of code added into ticket lock, which slows down the code path slightly. In less severe overcommitted configurations, where the preemption rate is low, the overhead becomes observable. However, with greater overcommitting of resources the overhead swamped by the overall performance improvement.

Figure 5(b) depicts the performance and speedup of each lock algorithm when executing *kernbench*. The results show that each of the four lock algorithms provides comparable performance when only one VM is executing, and lock preemption is rare. When the number of VMs increases, all three lock implementations yield significantly better performance compared to the generic *ticket lock*. Similar to *hackbench*, the same patterns are observed in the two VM case. In these scenarios the speedups of *preemptable-lock*, *pv-lock* and *pv-preemptable-lock* are $2.37X$, $2.47X$, $3.12X$. The result again confirms our hypothesis that Preemptable Ticket spinlocks improve performance even without paravirtual interfaces, and also yield better performance than *pv-lock* on a host with a paravirtual locking interface because it uniquely identifies and adapts to instances of lock waiter preemption. It is interesting that when configured with three VMs, all three preemption optimized lock algorithms exhibit almost the same performance, around $7.3X$ speedup. A possible reason for this is that kernbench is an I/O intensive workload, and with three parallel instances running the host I/O capacity becomes the bottleneck. In such a scenario, locking performance cannot improve performance past what the hardware I/O system is capable of.

In figure 5(c) we scale up to five VMs running *ebizzy* to compare lock algorithms with high preemption rates. Results show that while comparable performance is achieved under the single VM case, the high preemption rate (three VMs or more) results in the Preemptable Ticket spinlocks outperforming the others. Moreover, *preemptable-lock* achieves the best speedup in the 3 VM case even without host side paravirtual support. This may due to the fact that *preemptable-lock* does not have context switching overheads caused by the paravirtual interfaces entering and exiting the VMM. It also should be noted that *pv-lock* has superior performance when executing with 2 VMs. However, as the number of VMs increases on the same hardware *pv-preemptable-lock* begins to achieve better performance, as a result of the greater levels of contention. Intuitively this is because *pv-lock* is able to perform well when the preemption rate is low, however it's performance degrades as the level of resource contention increases resulting in a greater number of preemptions.

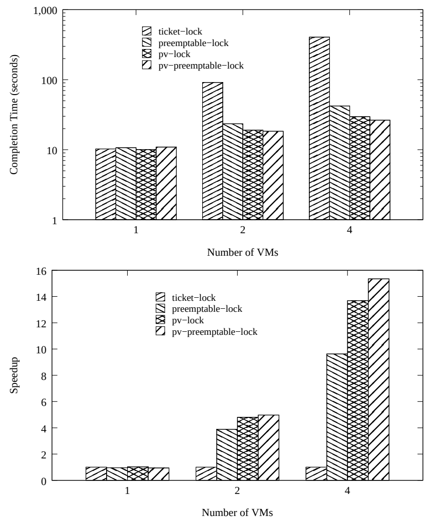
In summary, the lock waiter preemption problem is a situation best handled inside a guest OS without the need of VMM support. Because of this Preemptable Ticket spinlocks are specifically designed for both full system and paravirtual virtual environments. This allows our approach to adapt to guest behavior, and does not require communication with the VMM. This property becomes more significant as more VMs share the host and preemption becomes much more frequent. Our results show that when executing on a non-paravirtual VMM, *preemptable-lock* is able to improve guest performance significantly compared to *ticket-lock* when the host is overcommitted.

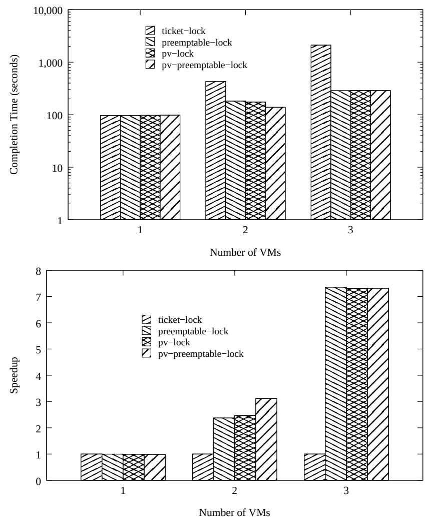
Paravirtual lock interfaces enable the guest to notify the VMM whenever it should transition a vCPU into and out of a sleep state due to a long waiting lock. This approach essentially converts a busy waiting lock in the guest into a sleep and wake up lock in the host. The benefit arises from the fact that the duration of preemption for a vCPU is on the order of milliseconds, which is $1000X$ longer than normal lock waiting time. With paravirtualization we are able to reduce this wait time at the cost of additional context switches and sleep-wakeup overhead. Our results show that *pv-lock* and *pv-preemptable-lock* are able to outperform *ticket-lock* significantly and yield better performance than *preemptable-lock* in most overcommitted cases.

While paravirtualization is able to deliver substantial improvements for performance, our *pv-preemptable-lock* is still able to outperform the *pv-lock* implementation. This is due to the fact that *pv-preemptable-lock* is able to address the problem from two directions. First when there is a preempted lock waiter, we do not require traps into the VMM that trigger sleep and/or wakeup operations. Instead, the ticket queue is reordered entirely inside the guest OS. The lock which is only available to the preempted lock waiter in *pv-lock* is available to other waiters in *pv-preemptable-lock* after a given timeout period. This allows the *pv-preemptable-lock* implementation to avoid unnecessary overheads due to the exit and entry costs required for a paravirtual interface. Second, for the case of preempted lock holders *pv-preemptable-lock* is able to improve system performance by leveraging the paravirtual lock interfaces. In other words, paravirtual locking is required to solve the lock holder preemption problem, whereas Preemptable Ticket spinlocks are required to address the lock waiter preemption problem. Obtaining the optimized performance requires utilizing a combination of both approaches.

6.3 Real World Workload benchmark

Finally, we evaluated the performance of different lock implementations when running a real world web application benchmark. The Dell DVD Store is a three tier benchmark, where tier 1 is a php web store application, tier 2 is an apache web server, and tier 3 is a MySQL database server. For these tests we ran the three tiers along with a client program sending login and search requests inside of a single VM environment. The experiments were conducted with up to 4 VMs executing the benchmark in parallel on the same physical host in order to show the performance of the different locking approaches under varying real world load scenarios.


(a) hackbench


(b) kernbench

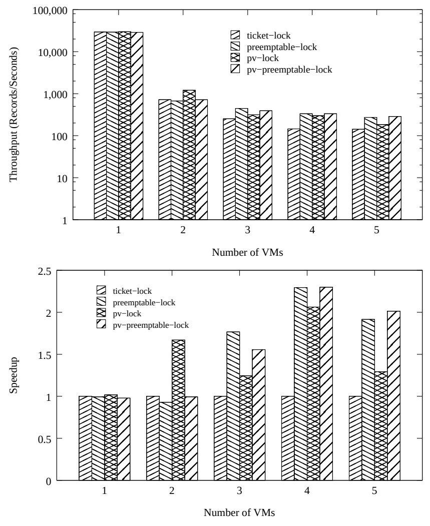

(c) ebizzy

Figure 5. Microbenchmarks Performance. *ticket-lock* and *preemptable-lock* show non-paravirtual performance results for a ticket lock and Preemptable Ticket spinlock kernel respectively. *pv-lock* and *pv-preemptable-lock* show performance results for paravirtual ticket locks and paravirtual Preemptable Ticket spinlocks

Figure 6 depicts the performance of each lock implementation when running on both a paravirtual and non-paravirtual VMM. Again, all lock algorithms show comparable performance under low overcommitted cases (one or two VMs), however, as the number of VMs increases, the results begin to mirror what was seen with the other benchmarks. Three of the preemption optimized algorithms outperformed *ticket-lock* significantly, which indicates the severity of performance degradation caused by preemption under overcommitted cases. While *preemptable-lock* yields obvious performance boosts without host side support, the two paravirtual solutions improved performance further. Moreover, *pv-preemptable-lock* is even better than *pv-lock* because it's unique ability to address lock waiter preemption. *pv-lock* yields best speedup in case of 2 VMs, but it has largest performance degradation in 1 VM case, and is outperformed by *pv-preemptable-lock* as number of VMs goes up.

7. Discussion and Future Works

By calculating the average speedup across all cases, we get that Preemptable Ticket spinlock can improve VM performance on average by $5.32X$ compared to the existing ticket lock architecture when running on a full system VMM. On a VMM that supports a paravirtual locking interface, Preemptable Ticket spinlocks can achieve $7.91X$ speedup over ticket locks on average, and a $1.08X$ speedup compared to pv-lock. The results show that Preemptable Ticket spinlocks effectively address the lock waiter preemption problem and can do so without any VMM modification. However when coupled with a paravirtual locking interface, Preemptable Ticket spinlocks can improve VM performance further compared to previous approach.

Though Preemptable Ticket spinlocks have demonstrated the ability to adapt to preemption on a overcommitted host, there are still further opportunities to fully optimize the lock behavior. In particular, further performance gains might be achieved through the integration of Preemptable Ticket spinlocks with other VMM scheduling algorithms. For instance, Complete Fair Scheduling (CFS), as used by KVM, tries to give equal shares of the CPU to each vCPU. Preemptable Ticket spinlocks could be used to provide inputs for other scheduling algorithms, such as co-scheduling and balanced scheduling, that can utilize dependency information about the currently running vCPUs. While we expect that a combined approach would improve overall performance, it is important to note that a combined approach is not required to achieve per-

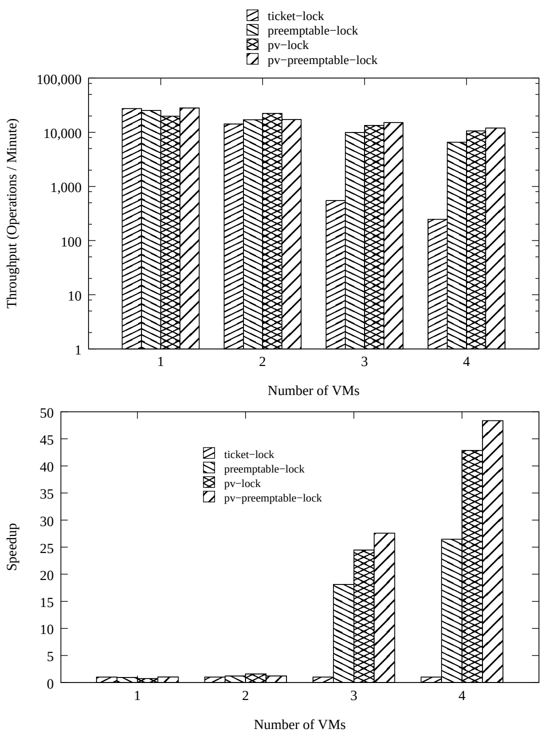


Figure 6. Dell DVD Store Performance

formance benefits when deploying our approach on other VMM architectures.

8. Conclusions

In this paper we have introduced Preemptable Ticket spinlocks as a new locking primitive targeting virtual machine environments that addresses lock waiter preemption. In particular, Preemptable Ticket spinlocks are able to avoid performance overheads that result from both lock holder and lock waiter preemption. While existing solutions are designed to only address lock holder preemption, Preemptable Ticket spinlocks are the first to fully address both preemption issues. Preemptable Ticket spinlocks are capable of addressing lock waiter preemption independently from the underlying VMM architecture, but when combined with a paravirtual lock interface it can handle lock holder preemption as well. With Preemptable Ticket spinlocks we are able to show that VM performance can be improved on average by $5.32X$, when running on a non paravirtual VMM, and by $7.91X$ when running on a VMM that supports a paravirtual locking interface.

Acknowledgments

We would like to thank Wencan Luo for insightful feedbacks and Raghavendra K. T. for discussions on the *pv-lock* patch.We would like to also thank the anonymous reviewers for helpful comments on the paper.

References

[1] Hackbench, 2008. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c/.

[2] Ebizzy 0.30, 2009. http://sourceforge.net/projects/ebizzy/.

[3] Kernbench 0.50, 2009. http://freecode.com/projects/kernbench.

[4] Dell dvd store database test suite 2.1, December 2010. http://linux.dell.com/dvdstore/.

[5] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (2001).

[6] FRIEBEL, T. How to deal with lock-holder preemption. Presented at the Xen Summit North America, July 2008.

[7] HE, B., SCHERER, W., AND SCOTT, M. Preemption adaptivity in time-published queue-based spin locks. In *High Performance Computing HiPC 2005*, D. Bader, M. Parashar, V. Sridhar, and V. Prasanna, Eds., vol. 3769 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 7–18.

[8] MCKENNEY, P., AND SLINGWINE, J. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[9] MELLOR-CRUMMEY, J., AND SCOTT, M. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS) 9*, 1 (1991), 21–65.

[10] MELLOR-CRUMMEY, J., AND SCOTT, M. Synchronization without contention. *ACM SIGPLAN Notices 26*, 4 (1991), 269–278.

[11] OUSTERHOUT, J. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (1982), pp. 22–30.

[12] RAGHAVENDRA, K., AND FITZHARDINGE, J. Paravirtualized ticket spinlocks, May 2012.

[13] RIEL, R. V. Directed yield for pause loop exiting, 2011.

[14] SUKWONG, O., AND KIM, H. S. Is co-scheduling too expensive for smp vms? In *Proceedings of the sixth conference on Computer systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 257–272.

[15] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3* (Berkeley, CA, USA, 2004), VM'04, USENIX Association, pp. 4–4.

[16] VMWARE, I. Vmware(r) vsphere(tm): The cpu scheduler in vmware esx(r) 4.1, 2010.

[17] WELLS, P. M., CHAKRABORTY, K., AND SOHI, G. S. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2006), PACT '06, ACM, pp. 124–133.

[18] WENG, C., LIU, Q., YU, L., AND LI, M. Dynamic adaptive scheduling for virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing* (New York, NY, USA, 2011), HPDC '11, ACM, pp. 239–250.

[19] ZHANG, L., CHEN, Y., DONG, Y., AND LIU, C. Lock-visor: An efficient transitory co-scheduling for mp guest. In *Proceedings of the 41st International Conference on Parallel Processing* (Pittsburgh, PA, USA, 2012), pp. 88–97.