How to Deal with Lock Holder Preemption
[Extended Abstract]

Thomas Friebel and Sebastian Biemueller

Advanced Micro Devices
Operating System Research Center
thomas.friebel@amd.com, sebastian.biemueller@amd.com

1 Introduction

Spinlocks are a synchronization primitive widely used in current operating system kernels. With spinlocks a thread waiting to acquire a lock will wait actively monitoring the lock. With sleeping locks in contrast a waiting thread will block, yielding the CPU to other threads. While sleeping locks seem to provide better functionality and overall system performance, there are cases in wich spinlocks are the better alternative.

First, under some circumstances, e.g. in interrupt handler top halves, blocking is not feasible. Second, saving and restoring a thread's state, as sleeping locks do when yielding the CPU, costs time. If the lock-protected critical section is very short waiting for the lock to be released offers better performance. In both cases spinlocks provide advantages over sleeping locks. But spinlocks are used for very short critical sections only to avoid wasting CPU time waiting actively.

Virtual machine monitors (VMMs) schedule virtual CPUs (VCPUs) on physical CPUs for time slices to achieve pseudo-parallel execution. At the end of a time slice the current VCPU is preempted, the VCPU state is saved and the next VCPU starts executing.

If a VCPU is preempted inside the guest kernel while holding a spinlock this lock stays acquired until the VCPU is executed again. This problem is called lock holder preemption, identified and analyzed by Uhlig et al.[3] for a paravirtualized version of Linux 2.4 running on top of the L4 microkernel.

This work investigates the influence of lock holder preemption in the Xen hypervisor, a commodity virtualization system. We show that lock holder preemption can have a severe performance impact in today's systems. Furthermore, we describe two approaches to counteract the performance degradation, give some details to our implementation of one of the approaches, and show that we are able to fully prevent any performance degradation caused by lock holder preemption.

2 Spinlocks and Virtualization

Lock holder preemption describes the situation when a VCPU is preempted inside the guest kernel while holding a spinlock. As this lock stays acquired during the preemption any other VCPUs of the same guest trying to acquire this lock will have to wait until the VCPU is executed again and releases the lock. Lock holder preemption is possible if two or more VCPUs run on a single CPU concurrently. And the more VCPUs of a guest are running in parallel the more VCPUs have to wait if trying to acquire a preempted lock. And as spinlocks imply active waiting the CPU time of waiting VCPUs is simply wasted.

Traditionally virtualization systems do not handle spinlocks in a special way. But as multi- and many-core machines are becoming more and more common the impact of lock holder preemption grows. Table 1 shows execution times and spinlock wait times for kernbench – a Linux kernel compilation benchmark – running under Xen 3.1 on a 4-socket 16-core machine.

In the single-guest setup a single 16 VCPU guest is running on the host system and executing kernbench. Here lock holder preemption is very unlikely as each VCPU can run on a distinct CPU and thus no preemption is necessary. The two-guests setup introduces a second 16 VCPU guest running a CPU bound job without any IO. We simply used 16 processes executing an endless loop. This results in an

Setup	Guest time [s]		Time spent spinning [s]	
Single guest	109.0		0.2	(0.2%)
Two guests	117.3	(+7.6)	9.0	(7.6%)

Table 1. Performance numbers for kernbench i) as a single VM, and ii) in an overcommited system running the kernbench VM and a CPU bound VM concurrently to cause lock holder preemption.

overcommited system, provoking lock holder preemption. Table 1 shows an 8.8 second increase in time spent waiting for a spinlock. The kernbench execution time increases by 7.6 seconds, or 7.0%.

To analyze the different behavior of Linux's spinlocks in more detail we instrumented the spinlock code to collect histogram information of spinlock wait times. Figure 1 shows the distribution of number of waits over their time spent waiting. Most of the waits (97.8%) do not take longer than 2^{16} CPU cycles. A second small fraction of waits, taking between 2^{24} and 2^{26} cyles, occurs only in the two-guests setup. These newly introduced waits match Xen's time slice lenght of 30ms and show lock holder preemtion: The VCPUs of the CPU bound guest always run for complete time slices as they do not block for I/O. Therefore a lock holder preempted by a CPU bound VM will keep the lock for at least a complete time slice. Any other VCPUs trying to acquire that lock will busy wait for at least a part of their time slice – until the lock holder is rescheduled and releases the lock.

Figure 2 plots the time spent waiting rather than the number of waits. This reveals that almost all of the time spent waiting is caused by the small number of waits caused by lock holder preemption.

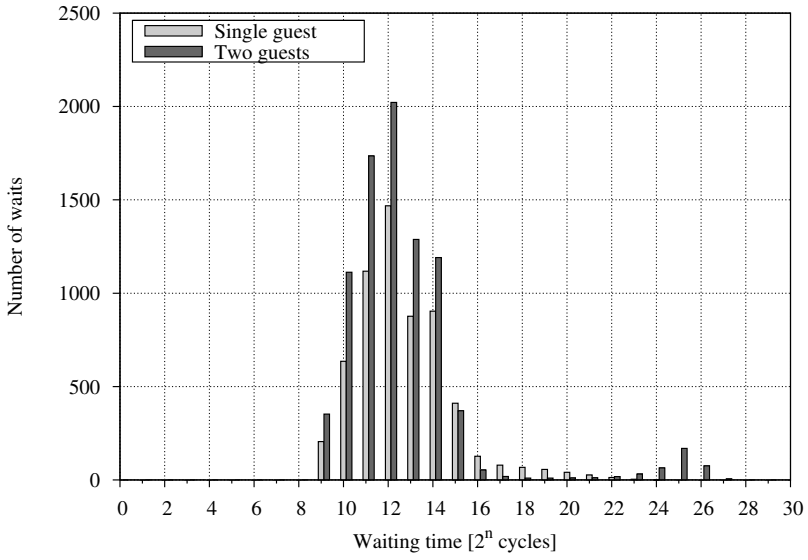

Fig. 1. Histogram of time spent waiting for a spin lock – number of waits for each histogram period. The spinlock waits are aggregated by waiting time into bins of exponentially growing size, e.g. bin 10 shows the number of waits that took between 2^9 to 2^{10} CPU cycles.

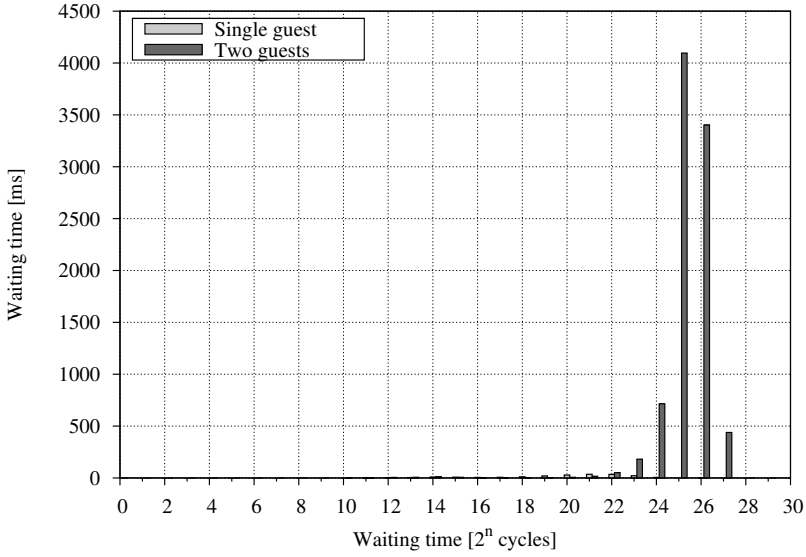

Fig. 2. Spinlock wait time histogram – duration of waits for each histogram period. The small number of very long waits around 2^{15} account for almost all of the time spent waiting.

3 Tolerating Lock Holder Preemption

We found two approaches to avoid the overhead caused by lock holder preemption. First, preventing lock holder preemption entirely by instrumenting the guest operating system as discussed by Uhlig et al.[3]. Their work leverages three specifics of spinlocks: 1) spinlocks are only used inside the kernel, 2) inside the kernel almost always one or more spinlocks are held, and 3) spinlocks are released before leaving the kernel. This allows to delay the preemption of a VCPU found to run in kernel space until it returns to user thus effectively preventing preempting a lock holder.

The second approach, the approach we follow in this work, tolerates lock holder preemption but prevents unnecessary active waiting. To achieve this we need to detect unusually long waits, and switch to a VCPU that is likely to not suffer from lock holder preemption. Ideally we would switch to the preempted lock holder to help it finish its critical section and release the lock. This is similar to locking with helping as described by Hohmuth and Haertig in [2].

To inform the virtual machine monitor of unusually long waits we extended the spinlock backoff code to issue a hypercall when waiting longer than a certain threshold. Motivated by the results of Figure 1 we chose a threshold of 2^{16} cycles. After this time almost all native spin-lock waits are finished as the results of the single-guest setup show. On reception of the hypercall the VMM schedules another VCPU of the same guest, preferring VCPUs preempted in kernel mode because they are likely to be preempted lockholders. The performance results after these modifications are presented in Table 2. Virtually no time is spent busy waiting in spinlock anymore. The CPU time spent for kernbench guest CPUs decreased by 7.6% compared to the unmodified two-guest setup and even by 0.6% compared to the single-guest setup.

Wall clock time decreased by 3.9%, which is only about half of the 7.6% guest time decrease. This is expected because our setups use shadow paging and kernbench induces a lot of shadow paging work into the VMM by creating a high number of processes. The VMM needs about as much time to handle the shadow paging requests as kernbench needs to complete the kernel compilation. As our modifications only affect the kernbench performance and not the hypervisor we achieve only about half of the guest performance improvement for the complete system. Switching to nested paging would probably yield additional performance.

Setup	Wall clock [s]	Guest time [s]		Time spent spinning [s]	
Two guests	34.8	117.3	(+7.6)	9.0	(7.6%)
Two guests, helping	33.5	108.4	(-0.6)	0.0	(0.0%)
Helping improvement	3.9%	7.6%		9.0	(7.6%)

Table 2. Kernbench performance numbers for lock holder preemption, and our helping approach.

4 FIFO Ticket Spinlocks

In early 2008, Piggin[1] introduced FIFO ticket spinlocks to the Linux kernel. Ticket spinlocks try to improve fairness in multi-processor systems by assigning locks to threads in the order they arrive at the lock. This intentionally constrains the number of threads able to acquire a contended lock to one – the next thread in FIFO order. In case of contention a released locks can not be acquired by any other thread when the next thread in FIFO order is preempted. This effect, called ticket holder preemption, can heavily impair performance.

Table 3 shows the performance impact of ticket holder preemption for our kernbench setup. The observed execution time drastically increases from 33 seconds to 47 minutes. The kernbench guest spends 99.3% of its time actively waiting to acquire a preempted spinlock. Using our lock holder preemption aware schedulding the execution time decreases to 34.1 seconds.

Setup	Wall clock [s]	Guest time [s]	Time spent spinning [s]	
Two guests	2825.1	22434.2	22270.4	(99.3%)
Two guests, helping	34.1	123.6	6.6	(5.4%)

Table 3. Lock holder preemption with ticket spin locks: Kernbench performance numbers for lock holder preemption, and our helping approach.

References

1. J. Corbet. Ticket spinlocks. *LWN.net*, 2008.
2. M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 217–230, Berkeley, CA, USA, 2001. USENIX Association.
3. V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.