The RCU-Reader Preemption Problem in VMs

Aravinda Prasad and K Gopinath, ***Indian Institute of Science, Bangalore;***
Paul E. McKenney, ***IBM Linux Technology Center, Beaverton***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/prasad

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the
2017 USENIX Annual Technical Conference
is sponsored by USENIX.

The RCU-Reader Preemption Problem in VMs

Aravinda Prasad
Indian Institute of Science, Bangalore

K. Gopinath
Indian Institute of Science, Bangalore

Paul E. McKenney
IBM Linux Technology Center, Beaverton

Abstract

When synchronization primitives such as locking and read-copy update (RCU) execute within virtual machines (VMs), preemption can cause multi-second latency spikes, increasing peak memory footprint and fragmentation inside VMs, which in turn may trigger swapping or VM ballooning. The resulting CPU utilization and memory footprint increases can negate the server-consolidation benefits of virtualization. Although preemption of lock holders in VMs has been well-studied, the corresponding solutions do not apply to RCU due to its exceedingly lightweight read-side primitives.

This paper presents the first evaluation of RCU-reader preemption in a virtualized environment. Our evaluation shows 50% increase in the peak memory footprint and 155% increase in fragmentation for a microbenchmark, 23.71% increase in average kernel CPU utilization, 2.9× increase in the CPU time to compute a grace period and 2.18× increase in the average grace period duration for the Postmark benchmark.

1 Introduction

Virtualization brings server-consolidation benefits to the cloud environment by multiplexing physical resources across virtual machines (VMs), but can lead to problematic preemption. For example, preemption of the virtual CPU (vCPU) holding a lock can cause latency spikes [18] because other vCPUs continue spinning to acquire the lock until the lock-holder vCPU resumes.

Well-known solutions to lock-holder preemption include priority inheritance [16, 8], and more recent work proposes solutions for the preemption of vCPUs holding locks [18, 14, 17, 2, 20, 23, 21]. Unfortunately, the heavyweight solutions proposed for lock-holder vCPU preemption, such as priority inheritance, do not apply to RCU because (1) RCU's read-side primitives must be exceedingly lightweight, and (2) preemption of RCU readers provokes different failure modes such as increased memory footprint. Nevertheless, preemption of vCPUs executing RCU readers has received little attention.

To the best of our knowledge, this is the first evaluation of vCPU preemption within RCU readers.

2 The RCU synchronization technique

Read-Copy-Update (RCU) [9, 12, 13] is a highly scalable structured-deferral [11] synchronization technique. *RCU read-side critical sections* are bounded by `rcu_read_lock()` and `rcu_read_unlock()`, which are bounded population-oblivious wait-free primitives that need not directly synchronize with writers. In consequence, each writer must guarantee that all data structures may be safely traversed by readers at all times.

For example, a writer deleting an object from a linked list first removes the object, then uses `synchronize_rcu()` to wait for all pre-existing readers to finish. Because new readers cannot gain a reference to the newly removed object, once all pre-existing readers complete, only the writer will have a reference to that object, which can then be safely freed. This writer-wait time period is called an *RCU grace period* (GP). Writers that cannot block may instead use `call_rcu()`, which posts an *RCU callback* that invokes a specified function with a specified argument after the completion of a subsequent GP. Although GPs can be expensive, batching optimizations allow thousands of `synchronize_rcu()` and `call_rcu()` requests to share a single GP [15], resulting in extremely low per-request GP overhead.

While the RCU-reader preemption problem is applicable across all RCU variants, this paper focuses on the "classic" RCU used by server builds of the Linux kernel. The "classic" RCU prohibits readers from executing any sort of context switch, as is also prohibited for spinlock holders. Therefore, any time interval during which all CPUs execute a context switch is by definition an RCU GP, as illustrated by Figure 1 [9, 12].

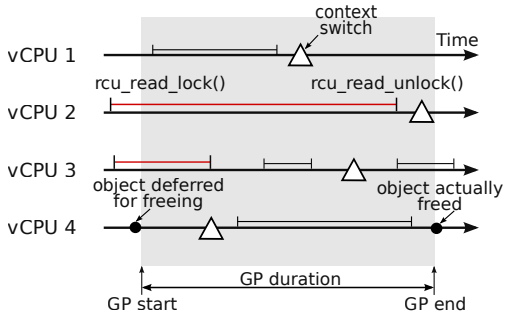


Figure 1: Linux-kernel grace period (GP). Red critical sections marked might hold references to the deferred object.

3 The RCU-reader preemption problem

RCU GPs cannot complete while a vCPU is preempted within an RCU read-side critical section. Thus, calls to `synchronize_rcu()` cannot return, and although calls to `call_rcu()` continue to return immediately, their callbacks cannot be invoked. Linux-kernel code can therefore continuously invoke `call_rcu()`, resulting in an unbounded quantity of memory that cannot be reused until the GP completes.

For example, consider an RCU-protected hash table that is searched incessantly and updated frequently, with deletions invoking `call_rcu()` to safely free old hash-table elements after a GP has elapsed. Suppose that just one vCPU is preempted within an RCU read-side critical section, but that the other vCPUs continue execution unhindered. These other vCPUs will continue their reads and updates, but because GPs cannot complete, elements deleted from the hash table cannot be freed until the preempted vCPU resumes its execution. This will increase memory footprint, which can in turn increase CPU utilization, for example, due to increased numbers of cache and TLB misses. CPU utilization can also increase because RCU takes increasingly aggressive measures in an attempt to force the preempted vCPU to execute the context switch needed to allow GP to complete. Unfortunately, these measures are futile because the vCPU itself has been preempted.

The RCU-reader preemption vs lock-holder preemption: While the usual symptom of lock-holder preemption is to hang all or part of the system, RCU-reader preemption instead bloats memory footprints.

Techniques to handle lock-holder preemption such as preemption-aware scheduling [23, 21] make the hypervisor aware of lock contention within the guest, and can be augmented by hardware support [20]. For instance, Intel's hardware-based Pause-Loop Exiting feature can detect a vCPU spinning on a lock. However, these techniques cannot be applied directly to RCU because RCU's server-build read-side primitives do not make any state change detectable by hypervisor or hardware (in fact the RCU's server-build read-side primitives are a *no-op* [10]). Although read-side primitives could make such a state change, doing so is problematic for two reasons. First, RCU's primary goal is zero or low-overhead read-side primitives, so RCU must push such overheads to writers. Second, state-change overheads are unacceptable for read-only or read-mostly data structures tracking the systems hardware configuration (e.g., active disks and online CPUs) where the read-to-write ratio (e.g., accessing a disk to replacing a disk) is well in excess of ten to the ninth power.

Therefore, alternative approaches are required to handle the RCU-reader preemption problem.

4 Impact of RCU-reader preemption

In this section we discuss both primary and secondary impacts due to the RCU-reader preemption problem.

Latency: Guest OSes invoking `synchronize_rcu()` can incur latency spikes of several *seconds* on overcommitted hosts. These spikes' durations depend directly on the vCPU preemption time.

Transient memory spikes: As discussed earlier, when using `call_rcu()`, GP delay due to vCPU preemption can cause transient memory-footprint spikes, which can in turn increase peak memory footprint.

Fragmentation inside VMs: Frequent transient memory-footprint spikes can scatter the kernel pages throughout the system, which can increase external memory fragmentation [4]. This fragmentation can cause premature memory-allocation failure, especially for hugepage allocations.

Swapping and Ballooning: Cloud environments often provision memory on an as-needed basis in order to reduce memory costs. Increased peak-memory footprint can trigger swapping, degrading performance and generating additional I/O load.

Furthermore, some cloud service providers oversubscribe memory because VMs do not always consume all their memory [22]. The combination of memory-footprint spikes and oversubscription can cause balloon drivers [19] to be frequently invoked as the hypervisor reacts to these spikes, further increasing overhead.

CPU utilization: The above issues can increase CPU utilization. For example, fragmentation might trigger compaction, which can consume significant CPU time while scanning and migrating memory.

VM density and consolidation: Increased peak-memory footprint require VMs to be provisioned with more memory, degrading VM density and consolidation, in turn increasing costs and energy utilization.

5 Factors influencing the impact of RCU-reader preemption

vCPU preemption time: GP-completion delays depend on vCPU preemption duration, which in turn depends on the hypervisor's CPU overcommit factor; higher overcommit factors increase vCPU preemption frequency which increases GP-completion delays.

RCU read-side critical section length: GP duration depends on read-side critical-section duration which, in the non-preemptible kernels this paper focuses on, depends on the time between voluntary context switches. As a rule of thumb, the longer this time, the greater the probability of preemption, and thus the greater the probability of GP-completion delays.

Objects allocation and defer free rate: Given vCPUs being preempted within RCU read-side critical sections, workloads that invoke `call_rcu()` frequently will see larger memory-footprint spikes than workloads that instead use `synchronize_rcu()`. Of the workloads that invoke `call_rcu()` frequently, those that allocate larger blocks of memory will see correspondingly larger memory-footprint spikes.

Total kernel time: Compute-intensive workloads spend little time in the kernel, which in turn means a given vCPU spends little time executing in-kernel RCU read-side critical sections. Therefore, RCU-reader preemption has a smaller effect on these workloads.

6 Evaluation

We evaluate a mail server benchmark, a memory-allocator intensive microbenchmark and a namespace cloning microbenchmark to understand the RCU-reader preemption impact under different stress conditions.

6.1 Benchmarks

Postmark [5] simulates a mail server's file create, delete, read and write operations. We run the benchmark on an in-memory filesystem starting with 128K files.

Memory microbenchmark, implemented as a kernel module, allocates an object of size 1K followed by a call to `call_rcu()` to reclaim the object after a GP.

Clone microbenchmark measures how quickly a new namespace can be cloned by calling the `clone()` system call in a loop from a user space program. Namespace cloning, for example, is employed by chroot jailing to create filesystem-isolated processes [6] and also in web server security that places the per user worker process into an isolated network [1].

6.2 Test setup

The host is an Intel Xeon E5-4640 processor having 64 CPUs (4 CPU sockets, 8 cores per socket and two-way hyper-threading) and 236 GB of physical memory. The host uses KVM [7] virtualization under Linux kernel 4.5.0 for both host and guests. Baseline measurements boot only the VM running the benchmark.

Experiment 1:

Instance	vCPUs	CPU Affinity	Memory
VM1	32	0–31	8 GB
VM2	32	32–63	8 GB
VM3	8	0–31	4 GB
VM4	8	32–63	4 GB

VM1 runs Postmark benchmark with 32 instances. VM2 runs memory microbenchmark with 32 parallel kernel threads. Both VM3 and VM4 run a bursty workload with 8 user space process. The bursty workload randomly executes 0.1 to 10 million arithmetic operations followed by randomly sleeping for 1 to 200 milliseconds.

Experiment 2:

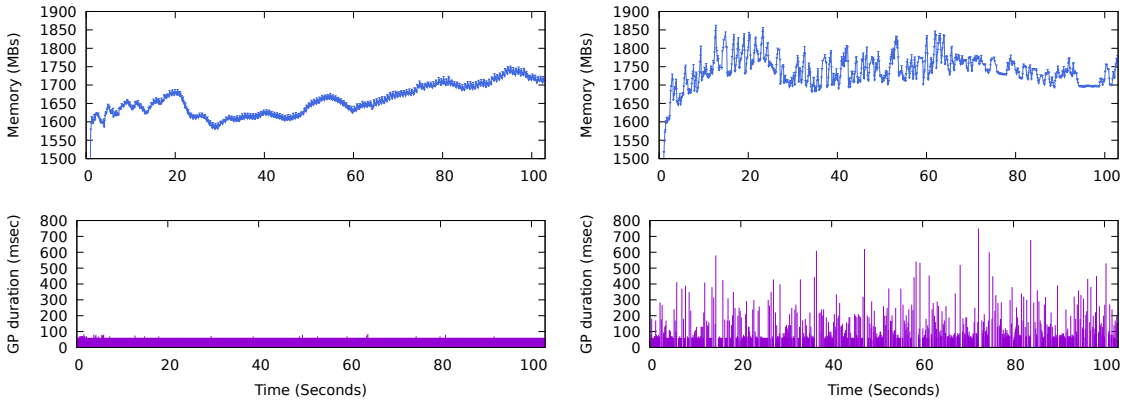
Instance	vCPUs	CPU Affinity	Memory
VM1	64	0–63	8 GB
VM2	16	0–31	8 GB

VM1 runs the clone microbenchmark and VM2 runs 16 CPU-hogging user processes on 32 vCPUs.

6.3 Results

Postmark: The file create and delete operations issued by the Postmark benchmark allocate filesystem objects such as `inode` and `dentry` (directory entry), and delete the objects by invoking `call_rcu()`. While the reclamation of the deferred objects' memory is delayed due to longer GPs, other benchmark threads continue performing file creation and deletion resulting in increased memory footprint.

Figure 2 reveals memory-footprint spikes in overcommit scenario due to delayed reclamation of `inode` and `dentry` objects. The vCPU preemption induces longer GPs which in turn delays the reclamation of deferred objects. There are no spikes in the baseline scenario because timely GP completion results in timely reclamation of memory.



(a) Baseline (b) Overcommit

Figure 2: Memory trace and GP durations for the first 100 seconds of the Postmark benchmark execution

Description	Baseline	Overcommit
Mean GP duration (ms)	57.6 (± 10.8)	125.9 (± 114.3)
Max GP duration (ms)	89.93	2372.12
Min GP duration (ms)	5.62	4.32
90th %-tile (ms)	60.05	251.36
50th %-tile (ms)	59.99	80.18
CPU consumed per GP (μs)	633.85	1833.54

Table 1: GP statistics for the Postmark benchmark

Description	Baseline	Overcommit
Mean GP duration (ms)	53.27 (± 13.4)	69.39 (± 30.4)
Max GP duration (ms)	87.66	317.59
Min GP duration (ms)	8.88	9.13
90th %-tile (ms)	60.18	109.98
50th %-tile (ms)	59.94	60.32
CPU consumed per GP (μs)	860.26	1095.72

Table 2: GP statistics for the memory microbenchmark

Table 1 shows a 2.18× increase in the average GP duration due to 6.33× increase in the number of RCU-reader preemption events extending GP duration. RCU's aggressive context-switch forcing results in a 2.9× increase in GP-computation time and further contributes to a 23.71% increase in kernel CPU utilization on overcommitted hosts.

Scattering of kernel pages due to frequent memory-footprint spikes results in a 32.5% increase in external fragmentation (computed using the `debugfs` "unusable free space index" for huge page allocations [3]) during benchmark execution when the host is overcommitted.

The above factors contribute to a 66.73% decrease in the throughput of the Postmark benchmark. However, the throughput is also affected by other factors including increased context-switch rates, preemption of vCPU holding a spinlock and reduction in number of vCPU assigned to the VM during host overcommit. We are currently investigating how much of this throughput decrease is due to RCU-reader preemption.

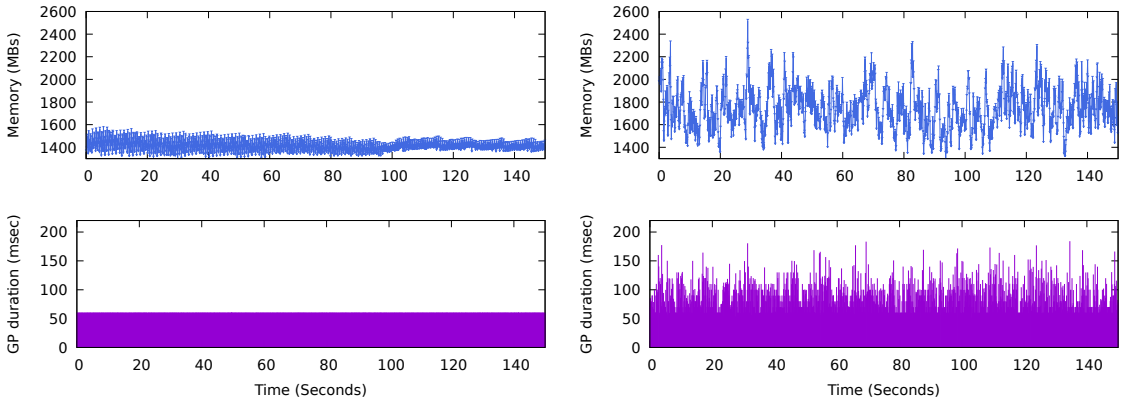
Memory microbenchmark: We run a memory-allocator-intensive benchmark to evaluate and understand the impact of RCU-reader preemption on GP durations and memory-footprint spikes. The microbenchmark issues 2.5K pairs of allocations and `call_rcu()` invocations per second per CPU. It also invokes the scheduler after every ten allocation-`call_rcu()` pairs to limit the duration of the resulting RCU read-side critical sections.

Figure 3 shows memory-footprint spikes of several hundred MBs due to longer GPs when the host is overcommitted. The resulting RCU-reader preemption results in a 50% increase in the peak memory footprint (and an 842 MB increase in peak memory footprint), a 30.26% increase in the average GP duration (Table 2) and a 155.32% increase in external fragmentation.

This microbenchmark shows a significant memory-footprint sensitivity to GP duration: A short 100-millisecond GP delay results in spikes of several hundred MBs in the memory footprint. In contrast, the Postmark benchmark, with its lower `call_rcu()` frequency, has a smaller memory-footprint sensitivity to GP duration, so that a longer 400-millisecond GP delay results in a memory-footprint spike of only about 50-100 MB.

Clone microbenchmark: The `clone` system call allocates several kernel objects during namespace cloning which are passed to `call_rcu()` when the last process exits that namespace. The `clone` microbenchmark therefore repeatedly invokes `clone` in a loop.

Figure 4 reveals occasional spikes in GP duration inside the VM running clone microbenchmark, even when the host's average CPU utilization is 28%. Such spikes depend on the vCPU preemption timing and result in



(a) Baseline (b) Overcommit

Figure 3: Memory trace and GP durations for the first 150 seconds of the memory microbenchmark execution

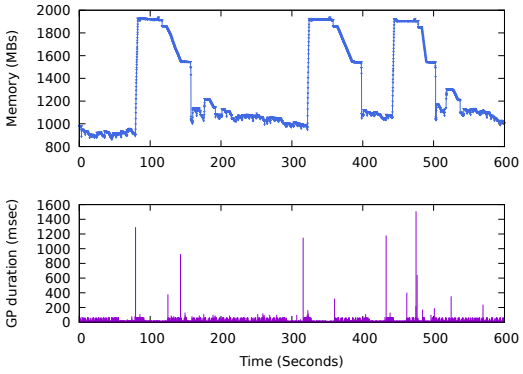


Figure 4: Memory trace and GP duration for the clone microbenchmark when the host is overcommitted

1 GB memory-footprint spikes persisting for several seconds. This result means that adding VMs (thus increasing the rate of `clone` invocations) can have the counterproductive effect of disproportionately increasing memory footprint due to increased RCU-reader preemption.

7 Discussion

RCU-reader preemption on an overcommitted host can result in latency spikes, increasing peak memory footprint and fragmentation within VMs. These increases can in turn increase CPU utilization due to increases in cache and TLB misses and due to additional memory-compaction operations. This increase in CPU utilization can reduce or even negate the cost and energy-efficiency benefits of server consolidation.

Cloud service providers and VM users should consider host overcommit ratios and workload sensitivities to delayed GPs while provisioning VM resources. Although GP-sensitive workloads can be identified via kernel profiling of `call_rcu()` and `synchronize_rcu()` invocations, it is currently difficult to determine the required changes to per-VM resource provisioning.

Furthermore, given systems with CPU overcommit, a CPU-consumption spike in one VM might cause a GP-duration spike in another VM. This sort of cross-VM interaction poses significant challenges for VM resource provisioning, which further motivates an effective solution to the problem of preemption of vCPUs running RCU read-side critical sections.

We are therefore currently investigating a holistic solution for the RCU-reader preemption problem that combines changes to the Linux-kernel RCU implementation, the guest-OS memory allocator, the hypervisor scheduler and the subsystems using RCU. The solution aims to reduce the GP duration on overcommitted hosts.

8 Conclusion

This paper introduces the RCU-reader vCPU preemption problem and demonstrates that it has significant and far-reaching performance impacts. We are investigating potential solutions to this problem.

9 Acknowledgments

We thank our shepherd, Eddie Kohler, and the anonymous reviewers for their helpful comments.

Disclaimer: The views in the article are solely of the authors and not of their employers.

References

[1] EDGE, J. Namespaces in operation, part 7: Network namespaces. `https://lwn.net/Articles/580893/`, 2014.

[2] FRIEBEL, T., AND BIEMUELLER, S. How to deal with lock holder preemption. *Xen Summit North America* (2008).

[3] GORMAN, M. mm: Export unusable free space index via debugfs. `https://lkml.org/lkml/2010/4/20/307`, 2010.

[4] GORMAN, M., AND WHITCROFT, A. The what, the why and the where to of anti-fragmentation. In *Ottawa Linux Symposium* (2006), vol. 1, Citeseer, pp. 369–384.

[5] KATCHER, J. Postmark: A new filesystem benchmark. Tech. rep., Technical Report TR3022, Network Appliance, 1997.

[6] KERRISK, M. Namespaces in operation, part 1: namespaces overview. `https://lwn.net/Articles/531114/`, 2013.

[7] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, pp. 225–230.

[8] MCKENNEY, P. Priority-Boosting RCU Read-Side Critical Sections. `https://lwn.net/Articles/220677`, 2007.

[9] MCKENNEY, P. E. *Exploiting Deferred Destructions: An Analysis of Read-Copy-Update Techniques in Operating System kernels*. PhD thesis, Oregon Health & Science University, 2004.

[10] MCKENNEY, P. E. What is RCU? Part 2: Usage. `http://lwn.net/Articles/263130/`, 2007.

[11] MCKENNEY, P. E. Structured deferral: synchronization via procrastination. *Commun. ACM 56*, 7 (July 2013), 40–49.

[12] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-Copy Update. In *AUUG Conference Proceedings* (2001), AUUG, Inc., p. 175.

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

[14] OUYANG, J., AND LANGE, J. R. Preemptable ticket spinlocks: Improving consolidated performance in the cloud. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 191–200.

[15] SARMA, D., AND MCKENNEY, P. E. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)* (2004), pp. 182–191.

[16] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39*, 9 (1990), 1175–1185.

[17] SUKWONG, O., AND KIM, H. S. Is co-scheduling too expensive for SMP VMs? In *Proceedings of the Sixth Conference on Computer Systems* (2011), ACM, pp. 257–272.

[18] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *Virtual Machine Research and Technology Symposium* (2004), pp. 43–56.

[19] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[20] 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* (2006), ACM, pp. 124–133.

[21] 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* (2011), ACM, pp. 239–250.

[22] WILLIAMS, D., JAMJOOM, H., LIU, Y.-H., AND WEATHERSPOON, H. Overdriver: Handling memory overload in an oversubscribed cloud. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 205–216.

[23] ZHANG, L., CHEN, Y., DONG, Y., AND LIU, C. Lock-visor: An efficient transitory co-scheduling for MP guest. In *Proceedings of the 2012 41st International Conference on Parallel Processing* (Washington, DC, USA, 2012), ICPP '12, IEEE Computer Society, pp. 88–97.