Paravirtualization for scalable Kernel-based Virtual Machine (KVM)

K. T. Raghavendra
*raghavendra.kt@in.ibm.com
Linux Technology Center
IBM India

Srivatsa Vaddagiri
vsrivatsa@gmail.com
India

Nikunj Dadhania
nikunj.dadhania@in.ibm.com
Linux Technology Center
IBM India

Jeremy Fitzhardinge
jeremy@goop.org
Exablox Inc
USA

***Abstract*—In a multi-CPU Virtual Machine(VM), virtual CPUs (VCPUs) are not guaranteed to be scheduled simultaneously. Operating System (OS) constructs, such as busy-wait (mainly spin locks and TLB shoot-down), are written with an assumption of running on bare-metal wastes lot of CPU time, resulting in performance degradation. For e.g., suppose a spin lock holding VCPU is preempted (aka LHP) by the host scheduler, other VCPUs waiting to acquire the same spin lock, waste lot of CPU cycles. Worsening this is the ticket based spin lock implementation, which requires next eligible VCPU to acquire the lock to be running. Similarly, remote TLB flushing API's does a busy wait for other VCPUs to flush the TLB. Above problems result in a higher synchronization latency for the workloads running within a VM. Especially in a massively over-committed environment like cloud, these problems become worse.**

One of the existing solutions is the hardware supported Pause Loop Exiting (PLE) mechanism which detects such busy-wait constructs inside the VCPU of a VM, and automatically does VCPU exit. Alternate implementation could be gang scheduling which tries to ensure VCPUs of VMs are scheduled simultaneously. But both the implementations suffer from scalability problem and are ill suited for cloud environments.

Paravirtualization is the best approach, where the guest OS is made aware that it is running in a virtualized environment, and optimize the busy-wait. Host OS also coordinate with guest to bring further performance benefits.

This paper discusses about paravirtualized ticket spin locks where VCPU waiting for spin lock sleeps, and unlocker identifies next eligible lock-holder VCPU and wakes it up. In paravirtualized remote TLB flush, a VCPU does not wait for other VCPUs that are sleeping, but all the sleeping VCPUs flush the TLB when they run next time.

Results show that, on a non-PLE machine, these solutions bring huge speed up in over-committed guest scenarios.

I. Introduction

Virtualization has gradually dominated Information and Technology ecosystem to leverage efficient use of resources and to provide isolation (security). Virtualization is used to consolidate the workloads running on several under-utilized servers to a fewer powerful machines, resulting in saving space, energy, cost, management and administration overheads of the server infrastructure.

Adapting existing Operating Systems (OS) to virtualization is a complex process. There are several ways to achieve virtualization [1], [2] and each of these have their own benefits and drawbacks. Addressing scalability, security and performance have been the major challenges [3], [4]. Paravirtualization is one of the novel virtualization technique to improve the performance. Here, the guest OS is made aware that it is running in virtualized environment and takes the advantage of the knowledge to adapt itself.

Paravirtualization is also used to solve the new set of problems that are introduced by virtualization that never existed in non-virtualized environments. Traditional OS were written with an assumption that they run on a Symmetric Multi Processing (SMP) environment where the physical CPUs run in parallel. But in a virtualized environment, the virtual CPUs (VCPUs) representing the physical CPUs, do not run in parallel. This is one of the basic assumption that the virtualization technique breaks. The VCPU that the guest[1] is running on, are the tasks for the hypervisor. The hypervisor[2] scheduler would preempt these tasks, hindering the progress of such critical tasks.

One such instance is the handling of busy-wait constructs which provide synchronization mechanism for shared resources' usage.

For e.g., (1) An initiator CPU busy-waits for all the other CPUs until they have executed the requested procedure, such as Translation Lookaside Buffer (TLB) Shootdown.

(2) Multiple CPUs waiting to acquire a shared resource in a busy-loop such as spin locks.

The above constructs become very inefficient in virtualized environment. We discuss paravirtualized spin locks and paravirtualized flush TLB algorithms for Kernel based Virtual Machine (KVM) [5] to solve the problem.

II. Paravirtual spin lock

Spin locks are generally implemented using busy-wait constructs. Spin locks have inherent problem of **L**ock **H**older **P**reemption (LHP) in virtualized environment. This problem is because, when a lock holder is preempted, all the VCPUs of the virtual machine (VM) waiting on the same spin lock burn the CPU time until the lock holder releases the lock. Fig. 1 depicts the typical LHP problem [6].

Ticket spin lock [7], [8] is a variation of spin lock which guarantees fairness by scheduling the waiters in FIFO to avoid starvation. In ticket spin lock, lock is divided into a head and

[1]guest and Virtual Machine (VM) are used interchangeably.
[2]host and hypervisor terms are used interchangeably.

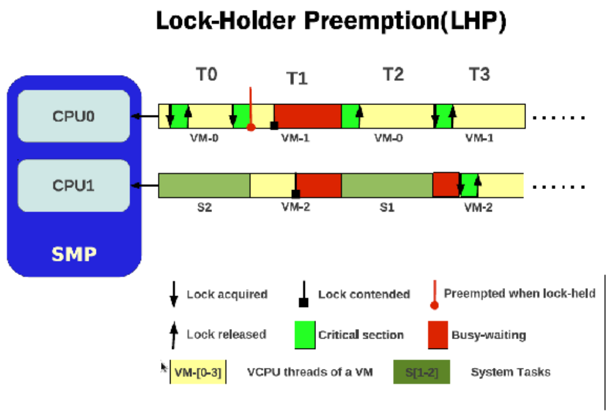

Fig. 1. Lock-Holder Preemption

a tail part as shown in Fig. 2. The head part indicates the current owner of the lock. The lock acquirer increments the tail value and waits for his turn. This approach adds even more overhead, because after a spin lock is released, there is exactly one eligible lock-holder, drastically slowing down the lock progress.

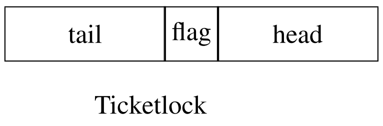

Fig. 2.

Here we explain paravirtual algorithm [9] to solve the above issue.

A guest VCPU waits in a busy-loop to acquire the lock for specified amount of time. In contended case, VCPU adds itself to a wait queue and sleeps. The contended case is indicated by setting the flag bit encoded in tail part of the lock. When the lock holding VCPU releases the lock, it checks whether there is any VCPU waiting for the lock. In such a case, it identifies the next eligible lock-holder and does a hypercall[3] to hypervisor to wake that VCPU. The guest side spin lock and spin unlock code is given in Fig. 3 and 4.

When a VCPU exits GUEST_MODE because of the halt instruction, it traps to halt handler in hypervisor, where it sleeps until, lock-holder releases the lock and wakes it up through kick hypercall. The halt handler is efficient since it donates it's time by invoking a directed yield call to an eligible VCPU. The algorithm for hypervisor is given in Fig. 6, 7 and 8.

Fig. 3. Guest: Ticket Spin lock

```
function TICKET_SPIN_LOCK(ticketlock L)
    $SPIN\_THRESHOLD = 8192$        ▷ Busy looping counter
    $L.tail = L.tail + 1$
    $myturn = L.tail$
    $counter = SPIN\_THRESHOLD$
    $mycpu = PROCESSOR\_ID()$
    while $true$ do
        while $counter$ do
            if $L.head = myturn$ then
                return
            end if
            $counter = counter - 1$
        end while
            ▷ Mark that current CPU is waiting for the lock. We assume a global array of waiting CPUs with (lock , want) pair to identify the next candidate
        $waiting[mycpu].want = myturn$
        $waitng[mycpu].lock = L$
        $L.flag = 1$

        halt()        ▷ This traps to halt handler in hypervisor
    end while
end function
```

Fig. 4. Guest: Ticket Spin-unlock

```
function TICKET_SPIN_UNLOCK(ticketlock L)
    $L.head = L.head + 1$
    if $L.tail = L.tail$ then
        $L.flag = 0$
    end if
    if $L.flag = 1$ then        ▷ flag indicates somebody is waiting.
        $vcpu = find\_eligible\_vcpu(L, L.head)$
                ▷ Below procedure traps to hypervisor and finally results in waking up the vcpu.
        $kick(vcpu)$
    end if
end function
```

III. Paravirtual flush TLB

A TLB is a cache of translations from virtual memory addresses to physical memory addresses. When a CPU changes virtual to physical mapping of an address, it needs to tell the other CPUs to invalidate the old mapping in their respective caches. This processes is called TLB shootdown.

In Virtualization, the initiator VCPU sends messages to other VCPUs to do the invalidation and busy-waits for all the target VCPUs to acknowledge the invalidation. When all the target VCPUs have invalidated their caches, the initiator VCPU progresses further. Here, the target VCPUs would have been preempted, resulting in the initiator VCPU wait for long amount of time.

This problem is addressed by making the initiator VCPU sending messages only to the running VCPUs. For all the non-running VCPUs, it will queue up a message for TLB flush [10]. When a VCPU enters GUEST_MODE, it checks whether a

[3]Hypercalls are one of the communication mechanism between the hypervisor and a guest.

Fig. 5. Guest: Finding next eligible lock-holder

```
function FIND_ELIGIBLE_VCPU(ticketlock L, integer T)
    for all $V$ do
            ▷ we assume $V$ spans 1 to total VCPU in VM
        if $waiting[V].lock = L$ AND $waiting[V].want = T$ then
            return $V$
        end if
    end for
end function
```

Fig. 6. Hypervisor: Halt handler

```
function HALT_HANDLER(vcpu V)
    while $V.kicked \neq true$ do
        $do\_directed\_yield()$    ▷ Donate time to other vcpu
    end while
            ▷ When loop is over, the vcpu enters the GUEST_MODE and continues execution.
end function
```

TLB flush is needed and proceed accordingly. This is given by algorithm in Fig. 10.

In our algorithm, we record the VCPU running state at every entry and exit of GUEST_MODE as indicated in Fig. 9.

IV. Results

We have used kernbench [11], ebizzy [14], hackbench [13] and sysbench [12] benchmarks for evaluation. Hypervisor and guest kernels are based on 3.5 Linux kernel [15]. Fig. 11 explains more about test environment.

Results show a huge improvement on a non-PLE machines [16] in over-committed[4] scenarios. However ebizzy, kernbench and hackbench have degraded in 1:1 over-commit scenario. Unlock path overhead is found to be the reason for this.

On PLE machines, the benchmarks have shown improvement in 1:1 scenario. PLE feature of hardware and the paravirtual spin locks serve similar cause and hence there is a performance degradation in over-committed cases.

[4]Represent a scenario where number of VCPUs in VMs exceed total number of physical cores.

Fig. 7. Hypervisor: Kick Hypercall

```
function KICK_HYPERCALL(int cpu)
    for all $vcpu$ do    ▷ map the vcpu of the VM to cpu
        if $vcpu.cpu = cpu$ then
            $vcpu.kicked = true$
            $break$
        end if
    end for
end function
```

Fig. 8. Hypervisor: Donating the execution time

```
function DO_DIRECTED_YIELD
    for all $vcpu$ do    ▷ map the vcpu of the VM to cpu
        if $vcpu.kicked = true$ then
            $continue\_execution(vcpu)$    ▷ vcpu continues execution
            $break$
        end if
    end for
end function
```

Fig. 9. Hypervisor: Guest entry and exit

```
function GUEST_ENTER(vcpu V)
    $V.running = true;$
    if $V.need\_flush = true$ then
        $tlb\_flush(V)$
    end if
end function

function GUEST_EXIT(vcpu V)
    $V.running = false;$
    if $V.need\_flush = true$ then
        $tlb\_flush(V)$
    end if
end function
```

Fig. 10. Guest: Flushing other VCPU's TLB

```
function FLUSH_TLB_OTHERS(Bitmap flushmask)
    for all $vcpu$ in $flusmask$ do
        if $vcpu.running \neq true$ then
            $vcpu.need\_flush = true$
            $clear\_cpu\_mask(vcpu, flushmask)$
        end if
    end for
    ▷ Ask running VCPUs to flush the TLB. This function blocks until all running CPUs flush TLB.
    $smp\_call\_function(flushmask, flush\_tlb)$

end function
```

Machine	A hardware with 32 CPU cores, 256 GB RAM, with 32 VCPU guests.
OS	base: 3.5 kernel patched: 3.5 kernel with paravirtual algorithm changes
Scenarios	1x: benchmark running on a single guest. 2x : benchmark running on two guests. 3x : benchmark running on three guests.
Run method	#threads = 2 * #vcpu

Fig. 11. Test set up

	base	patched	% improvement
kernbench 1x	42.0740	66.6613	-58.43823
kernbench 2x	299.4255	125.7638	57.99830
kernbench 3x	3020.4700	210.3157	93.03699
sysbench 1x	12.2632	12.0083	2.07858
sysbench 2x	14.5993	13.8636	5.03928
sysbench 3x	24.9869	18.8729	24.46882
hackbench 1x	44.0333	79.2499	-79.97720
hackbench 2x	6062.811	167.8835	97.23093

Fig. 12. non-PLE : Time taken in seconds. Lower is better

	base	patched	% improvement
ebizzy 1x	8178.7500	865.3750	-89.41923
ebizzy 2x	178.5000	757.2500	324.22969
ebizzy 3x	145.0000	672.6667	363.90807

Fig. 13. non-PLE : Records per second. Higher is better

	base	patched	% improvement
kernbench 1x	50.1392	51.0608	-1.83808
kernbench 2x	95.5267	94.6316	0.93702
kernbench 3x	200.0147	199.9703	0.02220
sysbench 1x	13.4145	12.1692	9.28324
sysbench 2x	14.2148	14.4314	-1.52376
sysbench 3x	19.9757	19.3714	3.02518
hackbench 1x	88.6456	72.7246	17.96028
hackbench 2x	124.3814	148.7440	-19.58701

Fig. 14. PLE : Time taken in seconds. Lower is better

	base	patched	% improvement
ebizzy 1x	1128.7500	1063.1250	-5.81395
ebizzy 2x	1861.6250	1154.5000	-37.98429
ebizzy 3x	1548.1111	861.7778	-44.33359

Fig. 15. PLE : Records per second. Higher is better

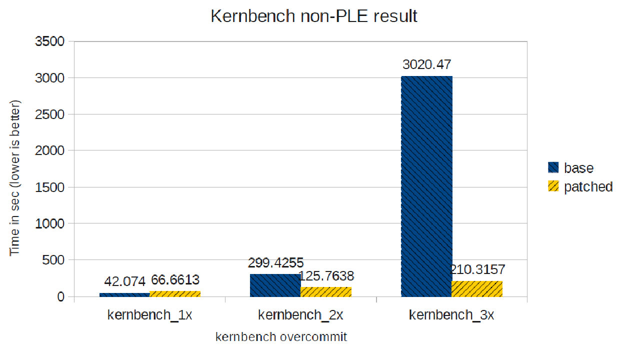

Fig. 16. Kernbench with non-PLE

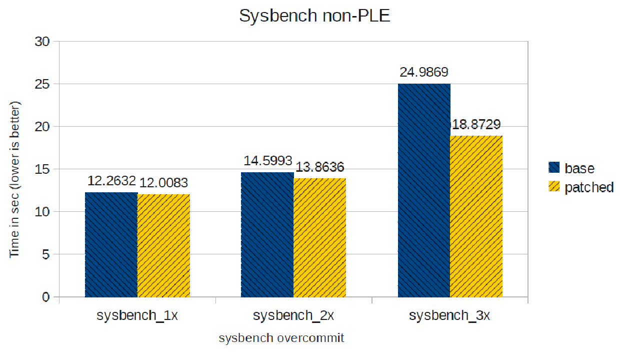

Fig. 17. Sysbench with non-PLE

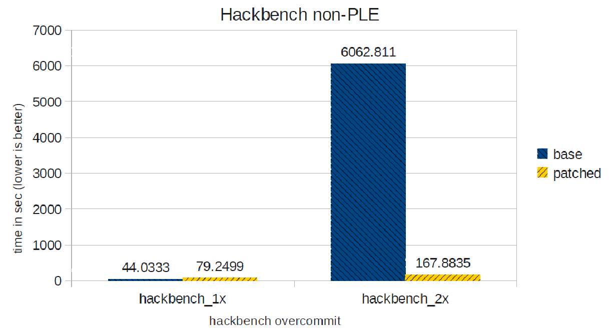

Fig. 18. Hackbench with non-PLE

V. Related Work and Conclusions

Gang scheduling [17] is one way to solve the busy-wait problem. It tries to schedule all the VCPUs of same VM simultaneously, thus making a VM run closer to bare-metal environment. But it adds more complexity and intrusive changes to the existing scheduler. The gang scheduling also needs the communication between all the CPUs to achieve synchronization between the tasks belonging to the same VM during hypervisor task scheduling. Hence suffers from scalability problem.

Hardware enabled Pause Loop Exiting [16] feature detects pause loops inside a guest VM and does a guest exit. The hypervisor PLE handler takes the control of further execution.

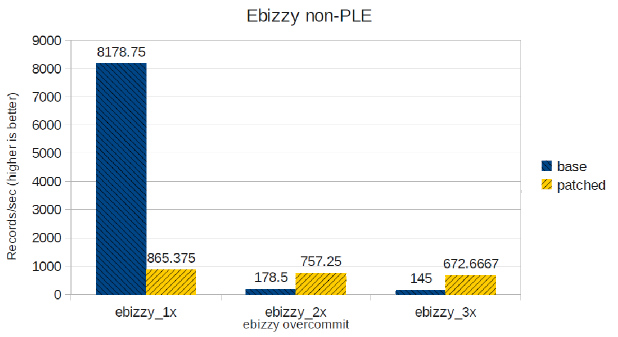

Fig. 19. Ebizzy with non-PLE

This saves CPU burning time. On the downside, the pause loop exit (PLE) handler have to be made intelligent to choose a best eligible [18] VCPU. Choosing a bad VCPU can further degrade performance. Since PLE handler works on some heuristics than exact data, it is difficult to improve it further. PLE is controlled by a PLE window parameter, which decides maximum number of cycle in a pause loop before a guest exit. PLE window needs to be tuned dynamically to get a good result. Though we can potentially control PLE window at runtime of VM, it is difficult to address the situation when we have a mix of workloads running in multiple VMs under same hypervisor.

Paravirtualizing the busy-wait constructs for KVM shows a huge benefit in over-committed scenarios on non-PLE environment. This is a necessary ingredient for cloud environment, where over-commits are common. However our approach also has a drawback of paravirtualization itself, since we need to modify both guest and host OS. Degradation in 1:1 over-commit scenario for PLE disabled and over-committed PLE enabled cases are yet to be improved as a future enhancement.

Synchronization technique, in particular ticket lock based implementation [19], [20] with backoff approaches are interesting. A similar approach, where a VCPU halts based on their turn in the wait queue can be adapted.

ACKNOWLEDGMENT

We thank Avi Kivity, Marcelo Tosatti, Peter Zilstra and many others who contributed as reviewers of the source code and suggested improvements in the kernel mailing list. We also thank Dipankar Sharma, Tarundeep S Karla for their suggestions and encouragement while writing this paper.

LEGAL STATEMENT

REFERENCES

[1] S. Vaughan-Nichols, "New approach to virtualization is a lightweight," *Computer*, vol. 39, no. 11, pp. 12 –14, nov. 2006.

[2] V. Chaudhary, M. Cha, J. Walters, S. Guercio, and S. Gallo, "A comparison of virtualization technologies for hpc," in *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, march 2008, pp. 861 –868.

[3] E. Mathisen, "Security challenges and solutions in cloud computing," in *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, 31 2011-june 3 2011, pp. 208 –212.

[4] Z. Amsden, D. H. Daniel Arai, A. Holler, and P. Subrahmanyam, "Vmi: An interface for paravirtualization," Linux Symposium, July 2006.

[5] "Kernel-based virtual machine (kvm)," "http://www.linux-kvm.org".

[6] T. Friebel, "Preventing guests from spinning around," Xen Summit, June 2008, "http://www.xen.org/files/xensummitboston08/LHP.pdf".

[7] J. Fitzhardinge, "Ticket to a tarpit," Linux Plumbers Conference, November 2010, "http://wiki.linuxplumbersconf.org/_media/2010:06-lpc2010-spinning.pdf"%.

[8] J. Corbet, "Ticket spinlocks," February 2008, "http://lwn.net/Articles/267968/".

[9] S. V. K. T. Raghavendra, Jeremy Fitzhardinge, "Paravirtualized ticket spinlocks," May 2012, "https://lkml.org/lkml/2012/5/2/119".

[10] N. Dadhania., "Kvm paravirt remote flush tlb," July 2012, "http://comments.gmane.org/gmane.comp.emulators.kvm.devel/95303".

[11] C. Kolivas, "Kernbench," December 2009, "http://mirror.sit.wisc.edu/pub/linux/kernel/people/ck/apps/kernbench/"%.

[12] A. Kopytov, "System performance benchmark," March 2009, "http://sourceforge.net/projects/sysbench/files/sysbench/0.4.12/".

[13] "Hackbench," 2008, "https://build.opensuse.org/package/files?package=hackbench&project=ben%chmark".

[14] V. Henson, "Ebizzy," January 2008, "http://sourceforge.net/projects/ebizzy/files/ebizzy/0.3/".

[15] "The linux kernel archives," "http://www.kernel.org/pub/linux/kernel/v3.x/".

[16] "Intel virtualization. technology specification for the ia-32 intel architecture," April 2005.

[17] N. Dadhania., "Gang scheduling in linux kernel scheduler," January 2012, "http://lanyrd.com/2012/linuxconfau/spdzq/".

[18] K. T. Raghavendra, "Improving directed yield in ple handler," July 2012, "https://lkml.org/lkml/2012/7/18/247".

[19] J. M. Mellor-crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, pp. 21–65, 1991.

[20] J. B. Carter, C.-C. Kuo, and R. Kuramkote, "A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors," Tech. Rep., 1996.