Coexisting Scheduling Policies Boosting I/O Virtual Machines

Dimitris Aragiorgis, Anastassios Nanos, and Nectarios Koziris

Computing Systems Laboratory,
National Technical University of Athens
{dimara,ananos,nkoziris}@cslab.ece.ntua.gr

Abstract. Deploying multiple Virtual Machines (VMs) running various types of workloads on current many-core cloud computing infrastructures raises an important issue: The Virtual Machine Monitor (VMM) has to efficiently multiplex VM accesses to the hardware. We argue that altering the scheduling concept can optimize the system's overall performance.

Currently, the Xen VMM achieves near native performance multiplexing VMs with homogeneous workloads. Yet having a mixture of VMs with different types of workloads running concurrently, it leads to poor I/O performance. Taking into account the complexity of the design and implementation of a universal scheduler, let alone the probability of being fruitless, we focus on a system with multiple scheduling policies that coexist and service VMs according to their workload characteristics. Thus, VMs can benefit from various schedulers, either existing or new, that are optimal for each specific case.

In this paper, we design a framework that provides three basic coexisting scheduling policies and implement it in the Xen paravirtualized environment. Evaluating our prototype we experience 2.3 times faster I/O service and link saturation, while the CPU-intensive VMs achieve more than 80% of current performance.

1 Introduction

Currently, cloud computing infrastructures feature powerful VM containers, that host numerous VMs running applications that range from CPU– / memory–intensive to streaming I/O, random I/O, real-time, low-latency and so on. VM containers are obliged to multiplex these workloads and maintain the desirable Quality of Service (QoS), while VMs compete for a time-slice. However, running VMs with contradicting workloads within the same VM container leads to suboptimal resource utilization and, as a result, to degraded system performance. For instance, the Xen VMM [1], under a moderate degree of overcommitment (4 vCPUs per core), favors CPU–intensive VMs, while network I/O throughput is capped to 40%.

In this work, we argue that by altering the scheduling concept on a busy VM container, we optimize the system's overall performance. We propose a framework that provides multiple *coexisting* scheduling policies tailored to the workloads' needs. Specifically, we realize the following scenario: the driver domain

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part II, LNCS 7156, pp. 407–415, 2012.

is decoupled from the physical CPU sets that the VMs are executed and does not get preempted. Additionally, VMs are deployed on CPU groups according to their workloads, providing isolation and effective resource utilization despite their competing demands.

We implement this framework in the Xen paravirtualized environment. Based on an 8-core platform, our approach achieves 2.3 times faster I/O service, while sustaining no less than 80% of the default overall CPU-performance.

2 Background

To comprehend how scheduling is related to I/O performance, in this section we refer shortly to the system components that participate in an I/O operation.

Hypervisor. The Xen VMM is a lightweight hypervisor that allows multiple VM instances to co-exist in a single platform using ParaVirtualization (PV). In the PV concept, OS kernels are aware of the underlying virtualization platform. Additionally, I/O is handled by the *driver domain*, a privileged domain having direct access to the hardware.

Breaking down the I/O path. Assuming for instance that a VM application transmits data to the network, the following actions will occur: i) Descending the whole network stack (TCP/IP, Ethernet) the netfront driver (residing in the VM) acquires a socket buffer with the appropriate headers containing the data. ii) The netfront pushes a request on the ring (preallocated shared memory) and notifies the netback driver (residing in driver domain) with an event (a virtual IRQ) that there is a pending send request that it must service. iii) The netback pushes a response to the ring and en-queues the request to the actual driver. iv) The native device driver, who is authorized to access the hardware, eventually transmits the packet to the network.

In PV, multiple components, residing in different domains, take part in an I/O operation (frontend: VM, backend–native driver: driver domain). The whole transaction stalls until pending tasks (events) are serviced; therefore the targeted vCPU has to be running. This is where the scheduler interferes.

The Credit Scheduler. Currently, Xen's default scheduler is the Credit scheduler and is based on the following algorithm: *(a)* Every physical core has a local run-queue of vCPUs eligible to run. *(b)* The scheduler picks the head of the run-queue to execute for a time-slice of 30ms at maximum. *(c)* The vCPU is able to block and yield the processor before its time-slice expires. *(d)* Every 10ms accounting occurs which debits credits to the running domain. *(e)* New allocation of credits occurs when all domains have their own consumed. *(f)* A vCPU is inserted to the run-queue after all vCPUs with greater or equal priority. *(g)* vCPUs can be in one of 4 different priorities (ascending): IDLE, OVER, UNDER, BOOST. A vCPU is in the OVER state when it has all its credits consumed. BOOST is the state when one vCPU gets woken up. *(h)* When a run-queue is empty or full with OVER / IDLE vCPUs, Credit migrates neighboring UNDER / BOOST vCPUs to the specific physical core (load-balancing).

Credit's Shortcomings: As a general purpose scheduler, Credit as expected falls shorts in some cases. If a VM yields the processor before accounting occurs, no credits are debited [7]. This gives the running VM an advantage over others that run for a bit longer. BOOST vCPUs are favored unless they have their credits consumed. As a result, in the case of fast I/O, CPU-bound domains get neglected. Finally CPU-bound domains exhaust their time-slice and I/O-bound domains get stalled even if data is available to transmit or receive.

3 Motivation

3.1 Related Work

Recent advances in virtualization technology have minimized overheads associated with CPU sharing when every vCPU is assigned to a physical core. As a result, CPU–bound applications achieve near-native performance when deployed in VM environments. However, I/O is a completely different story: intermediate virtualization layers impose significant overheads when multiple VMs share network or storage devices [6]. Numerous studies present significant optimizations on the network I/O stack using software [5,8] or hardware approaches [3].

These studies attack the HPC case, where no CPU over-commitment occurs. However, in service-oriented setups, vCPUs that belong to a vast number of VMs and run different types of workloads, need to be multiplexed. In such a case, scheduling plays an important role.

Ongaro et al. [7] examine the Xen's Credit Scheduler and expose its vulnerabilities from an I/O performance perspective. The authors evaluate two basic existing features of Credit and propose run-queue sorting according to the credits each VM has consumed. Contrary to our approach, based on multiple, co-existing scheduling policies, the authors in [7] optimize an *existing*, *unified* scheduler to favor I/O VMs.

Cucinotta [2] in the IRMOS[1] project proposes an real-time scheduler to favor interactive services. Such a scheduler could be one of which coexist in our concept.

Finally, Hu et al. [4] propose a dynamic partitioning scheme using VM monitoring. Based on run–time I/O analysis, a VM is temporarily migrated to an isolated core set, optimized for I/O. The authors evaluate their framework using one I/O–intensive VM running concurrently with several CPU–intensive ones. Their findings suggest that more insight should be obtained on the implications of co-existing CPU– and I/O– intensive workloads. Based on this approach, we build an SMP-aware, static CPU partitioning framework taking advantage of contemporary hardware. As opposed to [4], we choose to bypass the run-time profiling mechanism, which introduces overhead and its accuracy cannot be guaranteed.

Specifically, we use a monitoring tool to examine the bottlenecks that arise when *multiple* I/O–intensive VMs co-exist with multiple CPU–intensive ones.

[1] More information is available at: http://www.irmosproject.eu

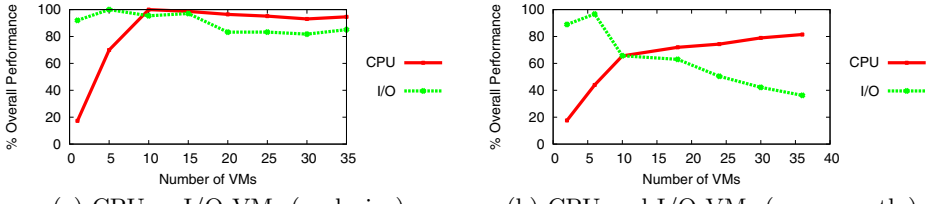
We then deploy VMs to CPU-sets (pools) with their own scheduler algorithm, based on their workload characteristics. In order to put pressure on the I/O infrastructure, we perform our experiments in a modern multi-core platform, using multi-GigaBit network adapters. Additionally, we increase the degree of overcommitment to apply for a real-world scenario. Overall, we evaluate the benefits of coexisting scheduling policies in a busy VM container with VMs running various types of workloads. Our goal is to fully saturate existing hardware resources and get the most out of the system's performance.

3.2 Default Setup

In this section we show that, in a busy VM container, running mixed types of workloads leads to poor I/O performance and under-utilization of resources.

We measure the network I/O and CPU throughput, as a function of the number of VMs. In the default setup, we run the vanilla Xen VMM, using its default scheduler (Credit) and assign one vCPU to the driver domain and to each of the VMs. We choose to keep the default CPU affinity (any). All VMs share a single GigaBit NIC (bridged setup).

To this end, we examine two separated cases:



(a) CPU or I/O VMs (exclusive) (b) CPU and I/O VMs (concurrently)

Fig. 1. Overall Performance of the Xen Default Case

Exclusive CPU– or I/O–intensive VMs. Figure 1(a) shows that the overall CPU operations per second are increasing until the number of vCPUs becomes equal to the number of physical CPUs. This is expected as the Credit scheduler provides fair time-sharing for CPU intensive VMs. Additionally, we observe that the link gets saturated but presents minor performance degradation in the maximum degree of overcommitment as a result of bridging all network interfaces together while the driver domain is being scheduled in and out repeatedly.

Concurrent CPU– and I/O–intensive VMs. Figure 1(b) points out that when CPU and I/O VMs run concurrently we experience a significant negative effect on the link utilization (less than 40%).

4 Co-existing Scheduling Polices

In this section we describe the implementation of our framework. We take the first step towards distinctive pools, running multiple schedulers, tailored to the

needs of VMs' workloads and evaluate our approach of coexisting scheduling policies in the Xen virtualization platform.

In the following experiments we emulate streaming network traffic (e.g. stream/ftp server) and CPU/Memory-bound applications for I/O– and CPU–intensive VMs respectively using generic tools (dd, netcat and bzip2). We measure the execution time of every action and calculate the aggregate I/O and CPU throughput. To explore the platform's capabilities we run the same experiments on native Linux and evaluate the utilization of resources. Our results are normalized to the maximum throughput achieved in the native case.

Testbed. Our testbed consists of an 8-core Intel Xeon X5365 @ 3.00 GHz platform as the VM container, running Xen 4.1-unstable with linux-2.6.32.24 pvops kernel , connected back–to–back with a 4-core AMD Phenom @ 2.3 GHz via 4 Intel 82571EB GigaBit Ethernet controllers.

4.1 Monitoring Tool

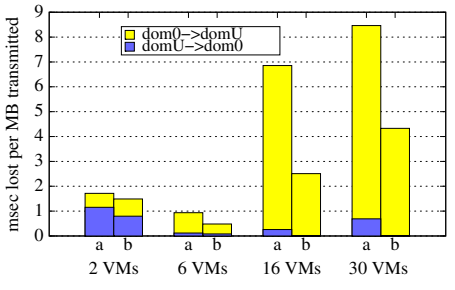
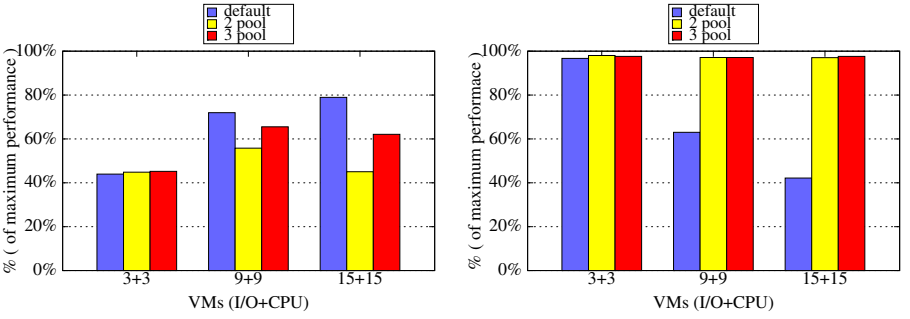


Fig. 2. Monitoring tool: msecs lost per MB transmitted: (a) default setup; (b) 2 pools setup

To investigate the apparent sub-optimal performance discussed in Section 3.2, we build a monitoring tool on top of Xen's event channel mechanism that measures the time lost between event handling (Section 2). Figure 2 plots the delay between domU event notification and dom0 event handling (dark area) and vice-versa (light area). The former includes the outgoing traffic, and the latter the acknowledges of driver domain and the incoming traffic (e.g. TCP ACK packets). We observe a big difference between both directions; this is debited to the fact that the driver domain gets more often awaken due to I/O operations of other domains, so it is able to batch work. Most important the overall time spent is increasing proportionally to the degree of over-commitment. This is an artifact of vCPU scheduling: the CPU-bound vCPUs exhaust their time-slice and I/O VMs get stalled even if data is available to receive or transmit. Moreover I/O VMs, including driver domain who is responsible for the I/O multiplexing get scheduled in and out, eventually leading to poor I/O performance.

4.2 The Driver Domain Pool

To eliminate the effect discussed in Section 4.1, we decouple the driver domain from all VMs. We build a primitive scheduler that bounds every newly created vCPU to an available physical core; this vCPU does not sleep and as a result does not suffer from unwanted context switch. Taking advantage of the pool concept of Xen, we launch this *no-op* scheduler on a separate pool running the driver domain. VMs are deployed on different pool and suffer the Credit scheduler policy.



(a) CPU Overall Performance (b) I/O Overall Performance

Fig. 3. Overall Performance using Pools: default; 2 pools; 3 pools

Taking a look back at Figure 2, we observe that the latency between domU and dom0 (dark area) is eliminated. That is because dom0 never gets preempted and achieves maximum responsiveness. Moreover the time lost in the other direction (light area) is apparently reduced; more data rate is available and I/O domains can batch more work.

Figure 3 plots the overall performance (normalized to the maximum observed), as a function of concurrent CPU and I/O VMs. The first bar (dark area) plots the default setup (Section 3.2), whereas the second one (light area) plots the approach discussed in this Section. Figure 3(b) shows that even though the degree of over-commitment is maximum (4 vCPUs per physical core) our framework achieves link saturation. On the other hand, CPU performance drops proportionally to the degree of over-commitment (Figure 3(a)).

The effect on CPU VMs is attributed to the driver domain's ability to process I/O transactions in a more a effective way; more data rate is available and I/O VMs get notified more frequently; according to Credit's algorithm I/O VMs get boosted and eventually steal time-slices from the CPU VMs.

Trying to eliminate the negative effect to the CPU–intensive VMs, we experiment with physical resources distribution. Specifically we evaluate the system's overall performance when allocating a different number of physical CPUs to the aforementioned second pool (Fig. 4). We observe that with one CPU, the GigaBit link is under-utilized, whereas with two CPUs link saturation is achieved. On the other hand, cutting down resources to the CPU-intensive VMs does not have a negligible effect; in fact it can shrink up to 20%.

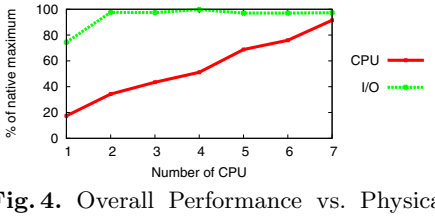


Fig. 4. Overall Performance vs. Physical Resources Distribution to VM pool

4.3 Decoupling vCPUs Based on Workload Characteristics

Taking all this into consideration we obtain a platform with 3 pools: *pool0* with only one CPU dedicated to the driver domain with the *no-op* scheduler; *pool1* with 2 CPUs servicing I/O intensive VMs (running potentially an I/O–optimized scheduler); and *pool2* for the CPU-intensive VMs that suffer the existing Credit scheduling policy. Running concurrently a large number of VMs with two types of workloads we experience GigaBit saturation and 62% CPU utilization, as opposed to 38% and 78% respectively in the default case (Fig. 3, third bar).

In addition to that, we point out that there is no overall benefit if a VM finds itself in the "wrong" pool, albeit a slight improvement of this VM's I/O performance is experienced (Table 1). This is an artifact of Credit's fairness discussed in previous sections (Section 4.2 and 3.2).

Table 1. VM Misplacement effect to individual Performance

	Misplaced VM	All other
CPU	-17%	-1.3%
I/O	+4%	-0.4%

5 Discussion

5.1 Credit Vulnerabilities to I/O Service

The design so far has decoupled I/O– and CPU–intensive VMs achieving isolation and independence, yet a near optimal utilization of resources. But is the Credit scheduler ideal for multiplexing *only* I/O VMs? We argue that slight changes can benefit I/O service.

Time-slice allocation: Having achieved isolation between different workloads we now focus on I/O pool (*pool1*). We deploy this pool on the second CPU-package and reduce the time-slice from 30ms to 3ms (accounting occurs every 1ms). We observe that I/O throughput outperforms the previous case, despite the decreasing packet-size (Fig. 5). Such a case, differs from the streaming I/O workload scenario (e.g. stream/ftp server) discussed so far (Section 4), and can apply to a random I/O workload (such as busy web server).

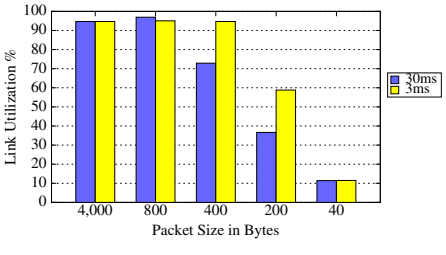


Fig. 5. Time-slice: 30ms vs 3ms

Anticipatory Concept: Moreover we propose the introduction of an anticipatory concept to the existing scheduling algorithm; for the implementation multi-hierarchical priority sets are to be used, while the scheduler, depending the previous priority of the vCPU, adjust it when gets woken up, sleeps, or gets credits debited. Thus, the vCPU will sustain the boost state a bit longer and take advantage the probability of transmitting or receiving data in the near future.

5.2 Exotic Scenarios

In this section we argue that in the case of multiple GigaBit NICs, a uni–core driver domain is insufficient. As in Section 5.1, we focus on *pool1* (I/O). This time we compare the link utilization of 1-4 x Gbps, when the driver domain is deployed on 1,2,3 or 4 physical cores (Fig. 6).

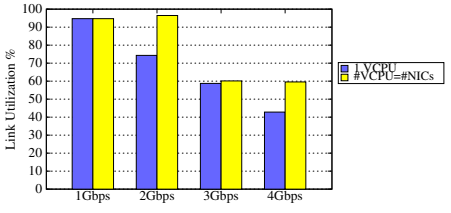


Fig. 6. Multiple GigaBit NICs

To exploit the SMP characteristics of our multi-core platform, we assign each NIC's interrupt handler to a physical core, by setting the `smp_affinity` of the corresponding irq. Thus the NIC's driver does not suffer from interrupt processing contention. However, we observe that after 2Gbps the links do not get saturated. Preliminary findings suggest that this unexpected behavior is due to Xen's network path. Nevertheless, this approach is applicable to cases where the driver domain or other stub-domains have demanding responsibilities such as multiplexing accesses to shared devices.

5.3 Dynamic Instead of Static

After having proved that the coexisting scheduling policies can benefit I/O performance and resources utilization we have to examine how such a scenario can be automated or adaptive. How to implement the VM classification and the resources partitioning? Upon this we consider the following design dilemma; the profiling tool should reside in the driver domain or in the Hypervisor? The former is aware of the I/O characteristics of each VM while the latter can keep track of their time-slice utilization. Either way such a mechanism should be lightweight and its actions should respond to the average load of the VM and not to random spikes.

6 Conclusions

In this paper we examine the impact of VMM scheduling in a service oriented VM container and argue that co-existing scheduling policies can benefit the overall resource utilization when numerous VMs run contradicting types of workloads. VMs are grouped into sets based on their workload characteristics, suffering scheduling policies tailored to the need of each group. We implement our approach in the Xen virtualization platform. In a moderate overcommitment scenario (4 vCPUs/ physical core), our framework is able to achieve link saturation compared to less than 40% link utilization, while CPU-intensive workloads sustain 80% of the default case.

Our future agenda consists of exploring exotic scenarios using different types of devices shared across VMs (multi-queue and VM-enabled multi-Gbps NICs,

hardware accelerators etc.), as well as experiment with scheduler algorithms designed for specific cases (e.g. low latency applications, random I/O, disk I/O etc.). Finally our immediate plans are to implement the anticipatory concept and the profiling mechanism discussed in the previous section.

References

1. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I.A., Warfield, A.: Xen and the Art of Virtualization. In: SOSP 2003: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 164–177. ACM, New York (2003)
2. Cucinotta, T., Giani, D., Faggioli, D., Checconi, F.: Providing Performance Guarantees to Virtual Machines Using Real-Time Scheduling. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 657–664. Springer, Heidelberg (2011)
3. Dong, Y., Yu, Z., Rose, G.: SR-IOV networking in Xen: architecture, design and implementation. In: WIOV 2008: Proceedings of the First Conference on I/O Virtualization, p. 10. USENIX Association, Berkeley (2008)
4. Hu, Y., Long, X., Zhang, J., He, J., Xia, L.: I/o scheduling model of virtual machine based on multi-core dynamic partitioning. In: IEEE International Symposium on High Performance Distributed Computing, pp. 142–154 (2010)
5. Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in Xen. In: ATEC 2006: Proceedings of the Annual Conference on USENIX 2006 Annual Technical Conference, p. 2. USENIX Association, Berkeley (2006)
6. Nanos, A., Goumas, G., Koziris, N.: Exploring I/O Virtualization Data Paths for MPI Applications in a Cluster of VMs: A Networking Perspective. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 665–671. Springer, Heidelberg (2011)
7. Ongaro, D., Cox, A.L., Rixner, S.: Scheduling i/o in virtual machine monitors. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2008, pp. 1–10. ACM, New York (2008)
8. Ram, K.K., Santos, J.R., Turner, Y.: Redesigning xen's memory sharing mechanism for safe and efficient I/O virtualization. In: WIOV 2010: Proceedings of the 2nd Conference on I/O Virtualization, p. 1. USENIX Association, Berkeley (2010)