The Case for Partitioning Virtual Machines on Manycore Architectures

Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu
Lawrence Berkeley National Laboratory
Email: {kzibrahim, shofmeyr, cciancu}@lbl.gov

Abstract**—***In this paper we characterize the behavior with respect to memory locality and I/O management of scientific computing applications running in virtualized environments. Our results using MPI, UPC and OpenMP implementations of the NAS Parallel Benchmarks, running on Intel and AMD based systems, indicate that lack of proper NUMA support determines an average performance degradation of 55% when compared to native. In the current full virtualization environments this performance degradation can be reduced by two techniques: i) hypervisor level page remapping; and ii) partitioning the NUMA domains between multiple virtual machines. Our analysis shows that hypervisor only schemes have little or no potential for performance improvement. When the programming model allows it, system partitioning with proper VM and runtime support is able to reproduce single node native performance: in a partitioned system with one virtual machine per socket the average workload performance is 5% better than native. Partitioning also improves I/O performance on clusters and we observe latency and bandwidth improvements as high as 10X in a two node experiment. This translates into end-to-end application performance improvements: partitioning decreases the average overhead of virtualization on the NAS MPI workload from 242% to 17%. Overall, our results indicate that partitioning is a simple and robust way to minimize the overhead of virtualization in HPC environments.*

I. Introduction

Virtualization technologies are ubiquitously deployed in data centers and offer the benefit of resource consolidation [1], performance and fault isolation, flexible migration [2] and easy creation [3] of specialized environments. They have been extensively used to run web server, E-commerce and data mining workloads. With the recent advent of the cloud computing paradigm, these workloads have been supplemented with High Performance Computing (HPC) applications: the Amazon Elastic Compute Cloud (EC2) already provides virtualized clusters targeting the automotive, pharmaceutical, financial and life sciences domains. The US Department of Energy is evaluating virtualization and cloud computing technologies in the Magellan [4] project. The tasks in commercial workloads are often independent and serve short lived requests; server tasks are started at virtual machine boot time and kept alive until shutdown. In contrast, HPC workloads have tasks tightly coupled by data movement and tend to persistently use a significant fraction of the system memory; applications are often run in batch jobs with multiple independent runs submitted simultaneously. In this paper we characterize the performance of HPC applications in virtualized multicore environments and quantify the performance expectations of several techniques designed to improve memory locality and I/O performance. For our evaluation we use a workload containing MPI, Unified Parallel C (UPC) and OpenMP implementations of the NAS Parallel Benchmarks [5].

In a virtualized environment, a virtual machine monitor (VMM or hypervisor) is inserted between the operating system and the hardware and multiple OS instances can run simultaneously, each inside a virtual machine (VM). Virtualized environments have to bridge a *semantic gap* between the hypervisor hardware resource management and the decoupled functionality inside the guest OS. Existing open source solutions (Xen [6], KVM [7]), as well as proprietary ones (VMware ESX [8] and hyperV) expose this gap in the areas of NUMA and I/O support: our study focuses on the behavior of KVM. We discuss related work in Section II and the details of the KVM architecture in Section III.

In Section V we discuss the performance provided by existing virtualization technologies on AMD and Intel UMA and NUMA processors. In contrast with previous HPC studies [9], [10], [11] which report little or no impact from virtualization on architectures with four or less cores, our results indicate a significant performance degradation (up to 82%) when VMs span sockets in 16 core NUMA architectures. Our analysis also indicates that distributed memory programming models such as MPI or Partitioned Global Address Space languages are better suited and provide better performance and scalability in virtualized environments when compared to shared memory programming models such as OpenMP.

NUMA affinity in virtualized environments is currently achieved by a combination of pinning guest VMs to CPUs and having the hypervisor memory management module allocate memory with affinity to CPUs. NUMA support in the OS running inside guests is usually disabled, but once a page is allocated by the hypervisor it will likely maintain the proper affinity. Our survey of the KVM and Xen developer mailing lists indicates that there is widespread belief that this cooperation provides most of the NUMA support needed.

In Section VI we analyze the interaction between current OS and hypervisor level memory management techniques and show that current solutions *cannot* provide good NUMA locality. Applications that start on a *cold* VM (right after booting) exhibit as much as 28% better performance than subsequent runs on warm VMs. The degradation is caused by the two-level memory management inherent in virtualized systems combined with the lazy page reclamation policies implemented in modern OSes: the end result is a locality leakage where pages are recycled from remote NUMA domains. The difference in performance between *cold* and *warm* runs provides a good upper bound for the expected performance gains of improved

NUMA support in virtualization technologies. The analysis also indicates that up to 25% of the pages used in runs on warm VMs have bad locality.

We then implement and analyze techniques designed to improve locality in full virtualization environments: 1) hypervisor only approaches (Section VII); and; 2) system partitioning (Section VIII), which required modifications in both the hypervisor and the parallel programming runtimes. These two approaches maintain the advantages of virtualization while requiring minimal modifications to existing software. In contrast, "enlightenment" [12] has been proposed as a Xen extension to inform the guest about the underlying hardware through hypercalls. No performance evaluation is available and this approach is facing resistance from the community since it breaks the virtualization tenets by enforcing a one-to-one virtual to physical CPU mapping. Paravirtualization approaches require guest modifications, e.g. Linux or Windows.

Our results indicate that hypervisor only approaches, while the most portable, have little potential for performance improvements: up to 90% of the page translation activities on a warm VM are serviced by the guest OS and only 2% of the page translations can be correctly handled by the hypervisor.

In Section VIII we explore system partitioning using multiple guest VMs. In order to achieve good performance we had to extend the KVM support and implement shared memory bypass in the MPI and UPC runtimes. Partitioning [13], [14], [15] is increasingly mentioned as an approach to improve performance on manycores: our results are the first presented for multicore NUMA systems in an application setting and add quantitative proof to the intuitive expectations. The results indicate that partitioning is able to provide the best overall performance and we observe up to 60% improvements when compared to the performance on VMs with "improved" NUMA support, as captured by the performance of the *cold* runs. This improvement is caused by a combination of good locality and decreased VM contention. Best performance is always obtained for the configurations where VMs are contained within one socket or NUMA domain.

In Section IX we explore the impact of partitioning on I/O performance and we observe as much as 10X latency and bandwidth improvements in a two node experiment. This translates into end-to-end application performance improvements: partitioning decreases the average overhead of virtualization on the NAS MPI workload from 242% to 17%. In the cluster environment, partitioning also enables configurations using full I/O virtualization to attain similar or better performance than paravirtualized configurations. This contradicts the current community agreement that paravirtualization is required for good I/O performance.

Our paper makes several contributions. We characterize the impact of current memory management techniques in virtualized environments and quantify the performance expectations of several KVM solutions designed to improve memory locality. We provide a bound on the performance expectations of improving NUMA support in virtualized environments without exposing the hardware architecture. Overall, for the workloads considered the existing implementations cause a 55% average performance degradation on KVM when compared to native performance, the average performance of the *cold* runs and better NUMA support is within 27%, while partitioning reduces this impact to 11%. Since there are no published results for techniques such as "enlightenment", our evaluation is also of interest to the proponents of providing contracts between virtual machines and hypervisors. We also show that partitioning also provides a simple and robust technique to improve the performance of the I/O subsystem.

II. Related work

Memory translation in virtualized environments has been extensively studied and hardware acceleration technology is provided by Intel VT-X [16] and AMD AMD-V [17]. The software solutions Xen [6], KVM [18] and VMware [8], use these hardware mechanisms for hypervisor page table traversal.

The impact of virtualization for scientific workloads has received its fair share of attention. Xu et al. [10] study the performance of multiple programming paradigms. Youssef et al. [19] evaluate the impact of Xen on MPI performance and report a low overhead of virtualization. They [11] also evaluate the impact of virtualization on multi-threaded linear algebra software and report low overhead on UMA systems. The NUMA studies to date have been conducted on systems with four or less cores and report a low performance impact.

Huang et al. present several Xen extensions to improve virtualization performance. They present Xen-IB [20] where shared memory bypass is implemented for communication between MPI processes on the guest OS and InfiniBand hardware. This shared memory is not used for inter-VM communication. They also implement [9] Inter-VM communication (IVC) using MVAPICH and Xen extensions to provide shared memory in the communication stack. They report good performance improvements on a cluster with dual-socket single core UMA nodes when running one VM per core. Inter-VM communication using shared memory for the TCP/IP stack is also discussed by Zhang et al. [21] for XenSocket, and by Kim et al. [22], again at low core counts. In contrast, our OpenMPI implementation bypasses completely the networking (IP) stack with shared memory between processes and it is able to provide better performance.

Lange et al. [23] present the implementation of Palacios and Kitten, a hypervisor and a lightweight OS for high performance computing. Although they report performance close to native for large scale systems, to our knowledge their results are obtained using *only one* of the eight cores per node and Palacios is not yet tuned for multicore.

In general, most of the cited studies [9], [20], [23], [10], [11], [19] for HPC workloads were conducted on systems with a low core count and do not discuss NUMA effects. Most studies report a low performance impact of virtualization: we could replicate this behavior only on UMA systems or a single socket in a NUMA system.

Partitioned operating system design on manycores has received a fair share of attention recently. Tessellation [13], Barrelfish [14] and `fos` [15] while describing different implementations, advocate for partitioning, running OS services as servers and for replacing the reliance on shared memory with message passing. These are relatively young projects and there is not enough experimental evidence using complicated workloads to show their promised benefits. Our results with partitioning are an encouragement. In particular, Tessellation advocates for resource management and a space time partitioning scheme using two level scheduling. Our analysis of locality leakage is of direct interest.

As all these projects advocate a lightweight OS design, a simplified approach to page management might alleviate the need for better NUMA support in virtualized environments. HPC specific systems such as Palacios and Kitten already restrict the virtual memory support. On the other hand, configurations currently used in cloud computing tend to use commodity Operating Systems (Linux) and commercial virtualization technologies (KVM, Xen) in configurations where a virtual machine spans all the available cores.

III. Memory Management in Virtual Machines

Handling memory translation is one of the difficult problems addressed in virtualized environments. Within any OS, paging is used to map the separate per process virtual address spaces to the single machine physical memory space. With virtualization, any VM presents a single address space to the hypervisor and a two level page translation scheme is required. Depending on the vendor and implementation, NUMA support is provided by one or more components in the virtualized environment.

The hypervisor functionality in KVM/Qemu is split between the Qemu emulator, the KVM device driver, and the host OS. Qemu is a machine emulator that creates a virtual machine image for the guest OS: the emulated machine can have any architectural configuration independent of the hardware support available. Qemu relies on the KVM Linux kernel driver to provide accelerated memory translation and I/O support. In turn, the KVM driver relies on the Linux host OS for NUMA support. While Qemu can create a NUMA based virtual machine, this emulated machine is intentionally decoupled from the underlying architecture. This decoupling is to allow flexible scheduling and migration of guests and also to allow dynamic resource allocation and management.

Similar to KVM/Qemu, VMware provides a solution based on a kernel driver and a machine emulator, but it does not expose any NUMA support to the guest. The VMware hypervisor relies on the guest OS support for NUMA. Xen provides a bare-metal hypervisor that interacts directly with the hardware. As such, Xen implements its own NUMA policy and statically allocates physical memory from the first NUMA domain on which the guest resides. This allocation is subject to space availability and does not maintain affinity if the guest spans multiple NUMA nodes. HyperV [24] does not have any explicit NUMA support.

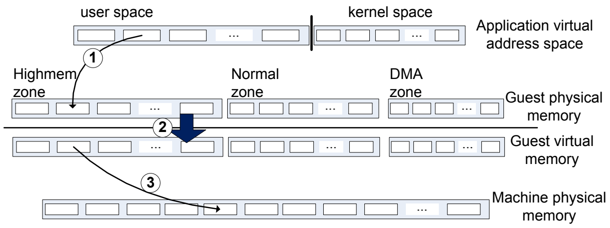

Fig. 1. *The three page-translation stages for guest application virtual addresses in KVM. The guest OS is responsible for the first translation phase. The KVM device driver manages the two other phases with the assistance of the host OS (hypervisor). Stage two of the mapping is static.*

A. The KVM Memory Management Unit

In KVM, the hypervisor runs inside the Linux host kernel. Since in KVM terminology the hypervisor is also referred to as the host, we will use interchangeably the terms host OS and hypervisor. Creating a virtual machine with KVM involves allocating memory zones that reflect the simulated architecture: some regions require direct mapping, e.g. I/O or DMA, while most of the physical memory is virtualized. The KVM/Qemu solution allocates all these regions and defines a one-to-one mapping between the guest physical memory and the host virtual memory. This mapping is transparently registered with the KVM device driver.

Figure 1 illustrates the KVM address translation steps. A memory reference within an application undergoes three levels of translation: *first*, application virtual address to guest physical address, performed inside the guest OS; *second*, guest physical to guest virtual memory on the hypervisor space, performed by the KVM modules, if hardware acceleration is active; and *third*, host virtual address to host physical address, performed by the (host OS) hypervisor. In KVM, the guest physical to host virtual mapping is static, and is usually divided into a few memory slots. The mapping on the host is dynamic, but the host OS cannot handle it directly. When a virtual CPU, registered with KVM, faults in a memory reference, the host OS (hypervisor) redirects the fault to the KVM driver. In Xen, the hypervisor runs directly on the bare metal and there are only two levels of page translation: since the mid-level KVM mapping is static the overall Xen translation process is similar to KVM.

Page Mapping Policy: Page translation for a first touch reference in a guest OS process involves four stages. 1) The process tries to reference its virtual memory, but as there is no physical translation, it traps into the guest OS to handle the fault. If the the virtual address is valid, the guest OS creates a page translation entry (PTE) in the page table. 2) When the application tries to access memory based on the new PTE, it faults again, but this time the fault is intercepted by the hypervisor (host OS) as the guest OS cannot handle it. The hypervisor redirects this fault to the KVM driver. 3) The KVM driver in turn reads the faulting PTE and updates the guest mapping. Then, it performs the second translation, from guest physical to host virtual, and requests the actual physical memory from the host OS. 4) The KVM module then updates the guest PTE with the correct physical memory. The application can resume execution normally. Any subsequent TLB refilling of this page will require only reading the page table of the application.

Page Reclamation: The host OS may decide be reclaim memory pages from the running guest OS, for instance to satisfy another VM or application. As the host cannot directly access the guest page tables, it notifies the KVM module about the page reclamation. The driver maintains a reverse mapping of all virtual CPUs address mappings that are using this page, it invalidates the relevant PTEs in the guest before returning the page to the hypervisor for reclamation. Obviously, the hypervisor involvement is not needed if page translations get cached in the guest OS. The host translation also remains intact if the same host page is reused multiple times by the guest.

B. Page Allocation Policy and Multi-Socket NUMA nodes

When a page is first touched, the actual physical page is chosen by the hypervisor. As the faulting virtual CPU is already occupying a physical CPU, the physical page can be allocated on the NUMA node with proper affinity. Thus, for the first touch of a page, NUMA-awareness is transparently provided by the hypervisor. When virtual CPUs are bound to physical CPUs the advantage of NUMA proximity of allocation can be maintained. KVM exposes the hardware architecture and a virtual CPU can be explicitly pinned to any physical CPU.

NUMA domains can be identified only at the host/hypervisor level and they are not exposed to the guest OS for two reasons. First, what appears to guests as physical memory is not allocated on the machine physical memory until it is touched for the first time inside guests. Second, while hypervisors try to observe NUMA allocation, no strong affinity guarantees are implemented or provided: the hypervisor retains its right to migrate or free pages as needed by the memory management policy to balance between concurrent activities. Consequently, only best-effort guarantees are provided for NUMA allocation.

For these reasons, virtualization technologies (KVM, Xen, VMware, hyperV) advocate “node confinement on NUMA architectures: they restrict the virtual machine to run within one NUMA domain, a strategy effective only when the number of virtual CPUs is less than the number of physical CPUs on a domain. As our results show, significant performance degradation occurs when this requirement is not met. The memory management in VMware [1] is relatively similar to that in KVM. In KVM, when faults are propagated, the hypervisor provides memory affinity with the faulting CPU, while in VMware affinity management is solely delegated to guests. In Xen, dom0 controls the pinning and while it provides guarantees that a virtual CPU is pinned to a physical CPU, it does not expose the identity of the latter. Affinity is provided using the `domain_to_node` API which determines the node associated with the first `vcpu` of the domain. Thus, Xen will exhaust one NUMA domain before moving to another, regardless of the faulting CPU.

IV. Experimental Setup

We experiment with KVM/Qemu 0.14.1 using the hardware virtualization support provided by CPU vendors: Intel VT-X and AMD-V. As guest OS we use the Linux kernel 2.6.32.8; for KVM we use the same kernel for the host OS. The three architectures used for the evaluation are: 1.6 GHz quad-core quad-socket UMA Intel Xeon E7310 (Tigerton), 2 GHz quad-socket quad-core NUMA AMD Opteron 8350 (Barcelona) and 2.4 GHz dual-socket quad-core NUMA Intel Xeon E5530 (Nehalem EP).

As a workload we use implementations of the NAS Parallel Benchmarks [5] in popular parallel programming paradigms: MPI (OpenMPI 1.4.2 with `gcc` 4.3.2), UPC (Berkeley UPC with `gcc` 4.3.2) OpenMP (`gcc` 4.3.2 with GOMP). We run the problem classes B and C and the memory footprint of the workload varies from tens of MBs to tens of GBs. Asanović et al [25] examined six different promising domains for commercial parallel applications and report that most of them use methods encountered in the scientific domain. In particular, all methods used in the NAS benchmarks appear in at least one commercial domain. Thus, beside their HPC relevance, these benchmarks are of interest to other communities.

All benchmarks are executed using all the cores available (16-way and 8-way parallelism) and each experiment is repeated at least 30 times. Some benchmarks (MPI BT and SP) require a square number of processors at runtime and we did not execute them in the 8-way configuration. The performance variation between all runs of the same experiment is less than 10% in all cases.

For network I/O, we use two UMA Intel Tigerton nodes connected through a Gigabit Intel 82545EM Ethernet controller. We assess the network performance with the *Multi-BW*(bandwidth) and *Multi-latency* OSU microbenchmarks 3.5 [26] between two nodes with 16 pair of processes.

For the multi NIC experiments we use mpich2 (version 1.4.1), as OpenMPI deadlocks. Single NIC performance for OpenMPI and mpich is statistically equivalent. For the network driver we use full virtualization of the *rtl8139* NIC adapter and a para-virtualized driver based on virtio [27].

Newer versions of the system software KVM/Qemu 1.0.1 and the Linux kernel 3.3.7 became available at the time of the writing, although not yet deployed in production settings. The NUMA performance and behavior on the newer releases is identical to that observed in our setting. I/O performance is improved from 40% of the hardware peak in our setting to 57% of peak. Although we have not tried to port all of our kernel modifications to the new releases, we feel confident that all conclusions and trends reported are still valid.

V. Single Node Multi-Socket Performance

Figure 2 shows the KVM performance compared to native performance when virtual machines span an increasing number of sockets. On a single socket, the performance on virtual

[1] Free VMware licenses allow only 2 vcpus, while commercial licenses are limited to eight vcpus. A HyperV is restricted to 4 vcpus.

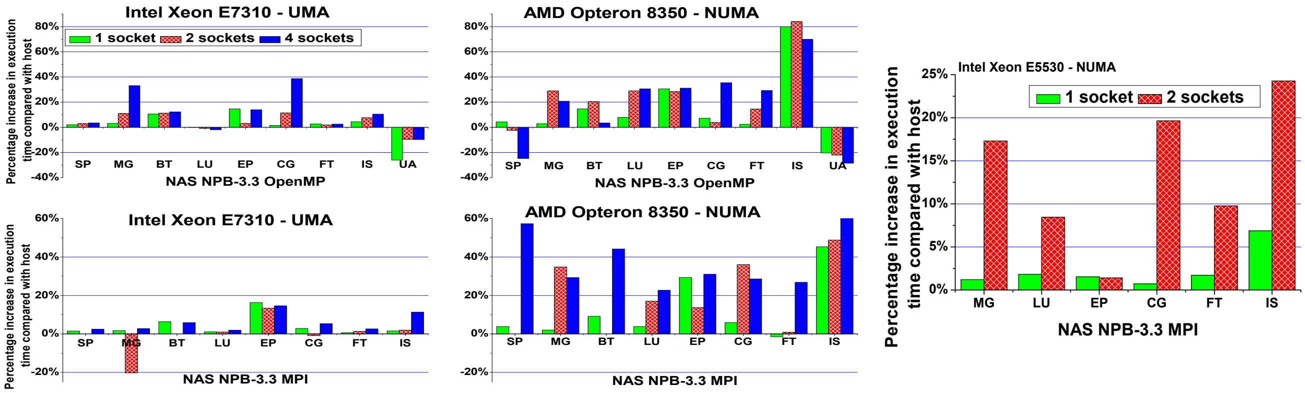

Fig. 2. *Performance of the NAS NBP3.3 benchmarks MPI and OpenMP implementations running with KVM on UMA and NUMA architectures.*

machines is mostly within 5% of the native performance, regardless of the programming model used in the benchmark implementation and the hardware utilized. Similar trends are reported by earlier studies [20], [23], [11] that show hardware virtualization being able to provide near native performance.

On the UMA architecture, the performance decrease is caused mostly by a combination of slower memory translation and slower performance in synchronization operations, e.g. MPI and OpenMP barrier performance decreases by roughly 20% with virtualization across four sockets. For brevity, we omit the detailed analysis of this behavior. On the NUMA architecture, the lack of proper support causes additional performance degradation. For example, on the AMD Opteron, MPI runs using four sockets are slowed down on average by roughly 40%, while single socket runs slow down by 10%. When increasing the number of sockets used by the applications, the performance degrades by up to 82%. The UPC results are similar to the MPI and omitted for brevity.

Many performance studies indicate that using large pages provides a performance benefit in virtualized environments. We have repeated the experiments using large pages with `libhugetlbfs` in all combinations of host and guest OS. For our particular workload, large pages cause performance degradation. While all the trends reported in this paper are valid when using large pages, all the results presented are for runs with small pages.

We have explored [28] many configurations for virtualization including multiple pinning strategies, different paging granularities and emulated NUMA on the guest. In particular, the same set of experiments repeated for Xen shows up to 4X slowdown on NUMA architectures. All results lead us to conclude that multi-socket NUMA architectures are associated with degraded performance for the current implementations of virtual machines. In Sections VII and VIII we discuss possible solutions and quantify their impact.

VI. Analysis of Paging Behavior

To monitor paging activities, we have extended the KVM device drivers to gather page faults and NUMA locality statistics. We also monitor the paging activity inside the guest OS: note that the information required to determine NUMA locality is not available at this level. Figures 3 and 4 show the paging activity observed at all translation levels, when running 16-way parallel MPI jobs (NPB class B) on the AMD Opteron NUMA system with a VM spanning the four sockets. We plot the percentage of page faults handled at each translation level grouped by their memory locality.

Figure 3 plots the faults observed during the MPI implementations run on a *cold* VM; that is, the monitored application is the first application running on the system after booting the VM. As shown, for a *cold* run most of the memory is not mapped and a significant percentage (up to 96%) of the application page faults reaches the KVM driver which enforces NUMA locality. These faults are captured by the bars labeled “Unmapped”. A smaller percentage (19% on average) of the page mapping is serviced by the guest OS without the need of the host involvement, even for a cold run, as illustrated by the bars labeled “Handled by guest”. As explained in Section VI-A, the guest OS filters faults due to the process based implementation of MPI. For the OpenMP case with `pthreads`, faults are not filtered by the guest and all faults are observed by the hypervisor. We could not determine the locality of these pages but we expect them to be local, similarly to the “Unmapped” pages.

The bars labeled “Local Node” and “Remote*” capture the percentage of faults that reaches the KVM driver for pages already mapped. “Local Node” pages have the correct affinity. The bars labeled “Remote Multiple” indicate faults for pages shared by multiple processes running inside the guest OS, while the “Remote Single” pages are not shared and are candidates for page migration. As shown, the combined contribution of all these pages is small.

A completely different behavior is observed after warming the VM, as shown in Figure 4, which captures the paging behavior for subsequent runs of the same application. The percentage of faults handled by the guest OS is high, up to 94%, due to caching of page mappings inside the guest. The NUMA node locality information is *not available* at the guest level and, while we cannot determine the locality of these pages, based on the behavior of the *cold* run we expect most of them to have the correct affinity. The percentage of page faults reaching the KVM driver is significantly reduced and we observe a high bias towards providing locality: for the vast majority of faults reaching the hypervisor the memory

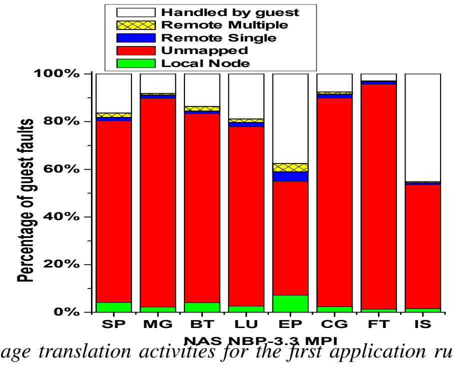

Fig. 3. *The page translation activities for the first application running after booting (cold) the VM.*

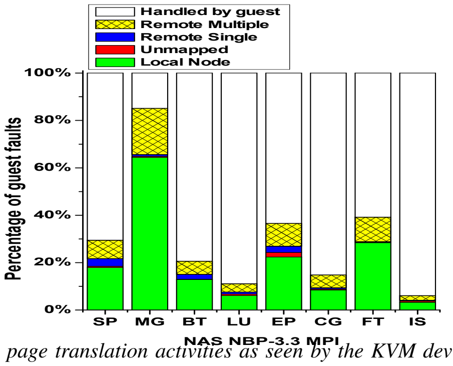

Fig. 4. *The page translation activities as seen by the KVM device driver for the second run of an application.*

is already mapped with the proper ("Local Node") locality. On the other hand, we observe a very noticeable increase in the percentage (up to 25% for "Remote Multiple") of faults that are found mapped in a remote node. Until explicitly returned for remapping, these pages will be "inherited" between applications and provide bad locality inside the guest OS.

This locality leakage is the cause for the performance degradations observed on NUMA architectures. Looking at the execution time, we found that MPI and UPC runs on *cold* VMs are always faster than subsequent runs, as shown in Figure 5. For example, a run of IS on a cold VM is 20% slower than the native run, while the subsequent runs are 60% slower than the native runs. On average, warm runs are 30% slower than cold runs. When measuring the system time on the guest (spent inside KVM and the host OS for handling faults) we found that cold runs have a much higher system time than subsequent runs. Later runs exhibit less than 25% of the system time of the first run, for most cases– but user time suffers significantly. All warm runs exhibit similar performance: this indicates that locality is lost mostly between the first and second execution of an application. The variation in performance for 30 *cold* and *warm* runs is within 5% for all benchmarks but IS which exhibits a 25% variation. The OpenMP cold runs are indistinguishable from warm runs and we explain the difference in the next section.

The temporal distribution of page faults is determined by the application's memory footprint and access pattern. We evaluate NPB implementations using class B and C settings; class C has the largest footprint of the two. Unless explicitly stated otherwise, the results presented are for class B problems. As shown in Section VIII, increasing the dataset size to class C increases the negative performance impact of virtualization.

For the class B problems, in all but two benchmarks (BT and SP) the majority of page faults happens at problem initialization time which is not accounted for by the NPB performance measurement methodology. Thus, most class B benchmarks do not fault during the measured runtime, only in BT and SP about 10% of the faults occur during performance measurements. This implies that the performance trends reported for class B are solely determined by the ability of the system to provide good locality when pages are initially allocated. Note also the clustering of performance trends in Figure 5: four benchmarks (MG, CG, LU, FT) provide *cold* run performance better than native, while four benchmarks (BT, SP, IS, EP) are slower than native. In the "fast" benchmarks the percentage of page faults filtered by the guest OS during a cold run is small with a peak observed by LU at 20% and there are few faults during the measured runtime. In contrast, the "slower" benchmarks either observe runtime faults (BT, SP) or have a high percentage ($\approx 40\%$) of faults filtered by the guest OS in the IS and EP case.

A. Programming Model Interaction with Virtualization

Figure 6 presents the distribution of the page faults intercepted by the hypervisor with respect to the NUMA nodes for a run on a cold VM. This is an indirect measure of the load balance of the application as well as its quality of locality optimizations. For the MPI and UPC runs, memory is evenly distributed across NUMA nodes for all benchmarks. For the OpenMP runs, three out of eight implementations are not well balanced across NUMA. Note that EP uses very little memory so the outlier is not really illustrative of bad locality. The even distribution of memory across NUMA nodes indicates that all benchmark implementations are optimized for locality.

For the MPI case, the cumulative number of faults observed by the hypervisor is well below 100% in most cases and as low as 50%. In contrast, for OpenMP the hypervisor observes almost 100% of the faults. This difference is caused by the implementation of the two programming models. OpenMP runs with `pthreads` and the application faults only once per page since a mapping due to a fault is observed by all threads. MPI runs with processes and provides a shared memory region for efficient inter-process communication inside its runtime. In MPI, only the first fault for a page in the shared memory region reaches the hypervisor and the faults generated by all other processes on the same page are served by the guest OS. The MPI implementations also have a larger memory footprint than the OpenMP implementations and observe a higher number (up to three times more) of page faults for each benchmark.

The Berkeley UPC implementation allows execution using either processes or `pthreads`. UPC, as well as other PGAS languages, exports as shared a large fraction of its heap, while MPI shares only "little" memory for communication buffers. The UPC NPB implementations have memory evenly distributed across NUMA nodes and in terms of fault propagation can behave in a similar manner to either MPI or OpenMP, e.g. 50% or 100% fault propagation. In UPC, *native executions*

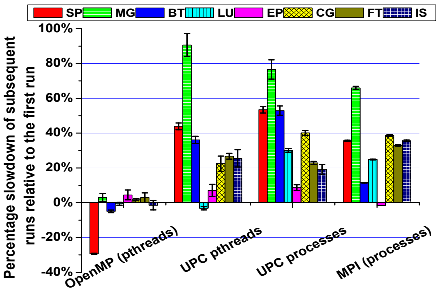

Fig. 5. *Execution time increase of runs on warm VMs compared to the first run after booting the VM. MPI and UPC applications become slower.*

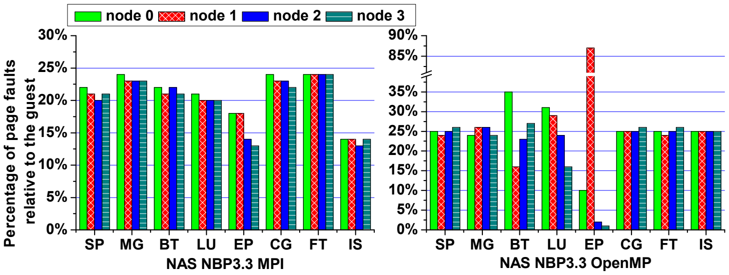

Fig. 6. *NUMA node distribution of page faults served by hypervisor for MPI and OpenMP cold runs. Faults are separated by NUMA nodes, evenly allocated pages show better NUMA allocation.*

with either processes or pthreads exhibit indistinguishable performance and we observe a pronounced difference between *cold* and *warm* runs, as shown in Figure 5. Comparing the process (38% slowdown) and pthreads (31% slowdown) based UPC implementations, the former shows a larger difference between the performance of *cold* and *warm* runs. This difference is explained by the paging behavior.

The OpenMP implementations exhibit identical performance in *cold* and *warm* runs. We attribute this behavior to the differences in the programming models. MPI and UPC have an inherent notion of locality and data is copied before reference, while OpenMP encourages a pure shared memory style programming with repeated access to possibly remote data. Intuitively, the OpenMP implementations have a worse NUMA locality of reference than MPI and UPC: the affinity shuffling that occurs between *cold* and *warm* runs degrades locality in MPI and UPC, while it does not significantly change the OpenMP locality.

VII. Hypervisor Extensions for NUMA Support

Page locality leakage in virtualized environments is caused by the current OS design paradigms which optimize for fast page fault handling at the expense of the reclamation mechanisms. Because page faults are in the critical execution path, the Linux kernel, as well as all other kernels, has an eager policy and a page fault causes an immediate trap to the OS for service. Virtual to physical memory mappings are aggressively cached within the OS. In contrast, page reclamation, swapping or removal, are done lazily by the OS depending on the amount of physical memory available: pages can be moved to an inactive state, cached or recycled. An asynchronous daemon is usually activated for reclamation whenever the number of the available pages drops below a certain threshold. In the exceptional case of not having enough pages to satisfy a request, an application may be synchronously blocked until enough pages are freed.

Hypervisor only approaches are most portable and generic since they do not require guest OS modifications. Several solutions are available to improve page locality: 1) pages can be migrated to the NUMA domain that has affinity with the faulting core and; 2) the mapping of pages inside the hypervisor can be completely reset when memory is no longer in use by the guest (upon application termination).

We consider first a brute force approach that forces swapping of the guest VM memory after each run in order to determine a new mapping for pages and achieve the effects of *cold* runs. We implemented a daemon that triggers page reclamation from the virtual machine, whenever an application terminates. Reclamation of pages depends on the page activity and it requires the page to age so that it can be swapped. For this experiment, we ignore the time needed to get the page to age and to swap it out and report only the effect on the performance of the next run. Unfortunately, this simple technique leads to a performance decrease of all runs. In KVM, the hypervisor sees only one address space for any VM and we are forced to swap both kernel and user guest OS pages.

We observed slowdowns compared with the cold-run performance measured as 17%, 50%, 17%, 26%, 10%, 42%, 10%, and 43% for SP, MG, BT, LU, EP, CG, FT, and IS, respectively. This performance decrease is larger than runs on a warm VM. This indicates that selective page remapping is a necessity, if the performance is to be improved.

A more specialized approach is to use migration to adjust the affinity of the pages whose faults have been propagated to the hypervisor. In this case, we can check if the page is in the right NUMA domain. To respect the first touch policy we avoid migrating pages used by other virtual CPUs. Migrating shared pages on faults implements a last touch policy and causes a hot-potato effect and further slowdown. We implemented this mechanism in the KVM driver using mechanisms similar to hotplug memory. As shown in Figure 4, while on average 30% of page faults are propagated to the hypervisor, only for a small fraction of pages ($\approx$ 1% labeled “Remote Single”) can the locality be improved. Overall, this approach did not improve the workload end-to-end performance. Detailed results are omitted for brevity.

VIII. Single Node VM Partitioning

Node confinement eliminates NUMA problems at the expense of scalability and generality. Using a software configuration with a separate VM on each socket requires a programming model able to run on distributed memory machines. OpenMP which requires shared memory and pthreads based implementations cannot span multiple VMs. On the other hand, programming models specifically designed for cluster based environments are well suited for this usage

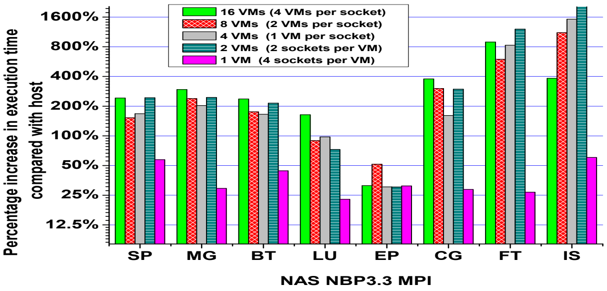

Fig. 7. *Performance of various VM configurations on the 4-socket Quad-core AMD Opteron 8350. Inter-VM communication uses virtio.*

scenario. These models include MPI, as well as the Partitioned Global Space Address (PGAS) languages illustrated here by UPC. All the implementations, experiments presented and inferences made for MPI in the rest of this section have been replicated using the Berkeley UPC implementation. Although for brevity we do not present any UPC results, our conclusions are valid for both implementations.

Figure 7 shows the performance of the MPI applications. For the configurations with multiple VMs, the MPI implementation uses the virtual network interface (loopback) and the IP stack for communication. Although we use the *virtio* driver which generally achieves up to 70% of the native hardware bandwidth, the performance degradation is large (up to 16 times) when increasing the number of VMs. Many other [21], [22], [27], [29] research activities tried to address communication problems in virtualized environments but we did not find any mature and usable solution for efficient communication between VMs using loopback.

A. Inter-VM Communication Using Shared Memory

Even when *virtio* performance reaches native hardware performance, inter-VM communication using loopback is less efficient than shared memory communication. Our shared memory communication in KVM uses the `ivshmem` [30] QEMU patch. `Ivshmem` exports shared memory on the host as a PCI device on the guest. Specifically, it creates a shared memory file on the host and memory-maps this device in the address space of the virtual machines. A device is created for the guest that is used to communicate information about the shared memory segment. On the guest OS, a kernel module is added to detect if the shared-memory device is exposed by the system emulator. As such, the module gets information about the address of the shared memory and tries to map it to the guest address space. It also initiates a device that can be used by applications, or runtime, to map the shared memory to their address space. The original implementation of `ivshmem` was based on 32 bit code and we had to extend it to 64 bit to allow a larger accessible address space.

In contrast, Xen-based virtualization [6] allows sharing pages between only two VMs [21], [29], [22] using the *GrantTable* and it imposes severe restrictions on the amount of memory allowed.

In general, inter-VM shared memory support poses design and security issues that caused the virtualization vendors and implementors to restrict it. First, sharing memory between VMs establishes a tight coupling and complicates VM migration. Second, the security and stability of the system is only as good as the protection mechanisms associated with the shared memory.

B. OpenMPI Extensions for Shared Memory Bypass

The OpenMPI implementation uses the modular component architecture (MCA) to integrate its various runtime components. The implementation uses several layers; at the bottom it uses an architecture dependent layer while the topmost layer provides the high level MPI functionality. There is also a glue layer between these two layers. Porting to a new architecture requires implementing a new byte transport layer (BTL). At startup, the MPI processes select the software components that can be used to communicate with any other process. If a process is reachable using multiple components, selection logic is used to decide the best component for communication. Components register themselves and declare their relative priority (exclusivity in the OpenMPI jargon). For instance, a process may be able to reach another using either TCP or shared memory BTLs, but because the shared memory BTL has a higher priority, the process then selects it. Each process maintains a list of BTLs that can be used for communication, one per destination process. To add a new communication BTL that exploits shared memory between VMs, we developed the following components:

1) A BTL that provides all the interfaces needed for inter-VM communication. This BTL has a lower priority than the native shared memory BTL and higher than any network BTL.
2) A memory pool component to handle shared memory allocation for the new communication BTL.
3) A memory-mapping component that handles the device responsible for shared memory.
4) A component that uses the special shared memory between the VMs for MPI collective operations: barrier, broadcast, *etc.*

We also implemented the logic to determine inter-VM reachability using shared memory: all VMs sharing a node are assigned a unique node identifier. Finally, the logic to choose different BTLs was modified to make sure that shared memory communication can coexist with other BTLs without conflicts.

Figure 8 shows the communication layers used in distributed memory systems. The communication between processes in a node uses shared memory and a networking layer is used for processes outside the node. Depending on message size and type, data may be queued without the need to block sender, or blocking may be needed to synchronize with the receiver. Two message queues are provided: a shared queue with a fixed size and an "eager" queue with size proportional to the number of senders. Because receiver queues are shared between senders, accessing them requires holding a lock to serialize the queue updates.

Our extension implemented between VMs sharing a node adds a new layer of communication, as shown in Figure 9. Two of the layers shown in the figure use shared memory;

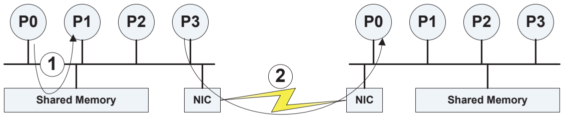

Fig. 8. *MPI Communication between processes without virtualization: 1- MPI communication within a SMP node uses shared memory; 2- MPI communication across SMP nodes uses the fastest available network card (using one of the tcp, IB, ..., etc BTLs).*

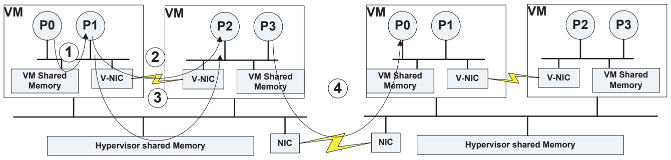

Fig. 9. *MPI communication with inter-VM bypass: 1- communication within a VM using a shared memory BTL; 2- Communication between VMs using virtual NIC ; 3-* newly introduced shared-memory communication BTL for communication between VMs*; 4- communication between VMs across nodes using NIC interfaces.*

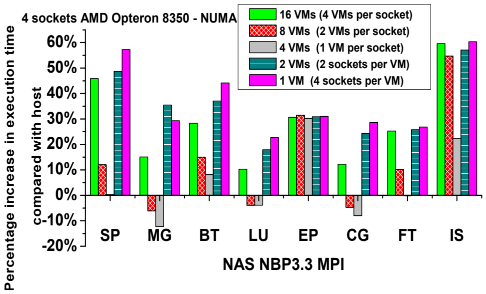

Fig. 10. *The performance of MPI NAS benchmarks with different virtual machine configurations on the 4-socket Quad-core AMD Opteron E8350. Inter-VM communication uses shared memory.*

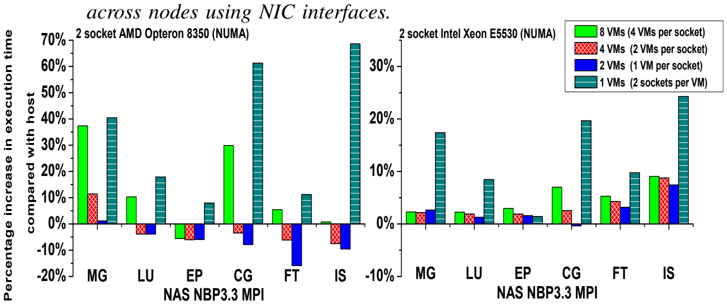

Fig. 11. *The performance of MPI NAS NBP benchmarks with different virtual machine configuration on 2-socket Quad-core AMD Opteron E8350 and Intel Xeon E5530.*

one within the VM and the other within a node. A third path is used to communicate across nodes. The effect of this additional layer is to create more localized communication queues within the VM and other queues for communication between VMs. These communication queues are protected by separate locks, thus less conflicts are expected in the new environment. Using the OMB [31] benchmarks to measure the bandwidth and latency for 16 MPI tasks split into 8 pairs on the AMD system, we measure three orders of magnitude improvement over IP for small messages, with the smallest difference of 66x associated with large messages.

OpenMPI does not currently support the ability to switch between BTLs at runtime. Without hot switching of components, there are restrictions on migrating VMs while applications are running. For instance, an application will need to switch from the inter-VM shared memory BTL to the TCP BTL if one of the participant VM migrates to a remote node. Adding hot-swapping capability to the OpenMPI runtime component architecture will provide a full solution for virtualized environments when socket partitioning is desired.

C. Single Node MPI Performance with VM Partitioning

Figure 10 presents the performance on the quad-socket, quad-core AMD NUMA system when partitioning the cores between virtual machines. Three VM configurations are node confined, while in the others (1 and 2) VMs span four and two NUMA nodes respectively. As shown, the performance varies with the VM configuration and the best performance is always attained by the configuration with one VM per socket. Node confinement with one or two VMs per domain always produces better performance than a single wide VM. In five cases, the best performance with partitioning matches or exceeds the native performance.

Configurations with more than two VMs per domain provide lower performance than the default of one VM per system. Our conjecture is that having multiple VMs per socket unnecessarily stresses the memory subsystem by having multiple OS images serving few processes, which leads to less effective caching and less allocated time slots. Kernel SamePage Merging (KSM) is a recent Linux kernel feature which combines identical memory pages from multiple processes into one copy-on-write memory region. Note that these experiments were run with KSM enabled.

Virtualization introduces two-level locking on data structures used to manage shared resources, such as memory. In addition to enforcing NUMA affinity, partitioning also reduces lock contention in the system and ultimately provides better memory management scalability. The faults on inter-VM shared memory occur during the application initialization phase, which is not used when reporting NPB performance. Thus, the performance of UPC implementations behaves identically to the MPI performance. When measuring application initialization time, we observe about 50% increase for *cold* runs compared with *warm* runs.

Figure 11 shows the same experiment (class B) conducted on two-socket quad-core AMD Opteron and Intel Nehalem systems. In this case, all applications noticeably benefit from partitioning: using one VM per socket matches *at least* native performance and even improves the performance by up to 15% in five out of six benchmarks. Without partitioning the performance degrades by up 70% on the AMD system. Comparing the results in Figure 11 with the results in Figure 10, notice that the performance improvement for the two-socket experiments is larger than for the four-socket ones. One would expect the impact of lack of NUMA support to grow with the number of sockets/domains, but in this case this is mitigated by better caching behavior. Better cache behavior reduces the frequency of visiting the memory subsystem asking for lines, thus reducing the impact of bad NUMA locality.

Figure 12 shows the behavior of classes B and C and it illustrates the effect of increasing the dataset size. More runtime page faults increase the page locality leakage during the application execution. Increasing the dataset size has also the side effect of reducing the effectiveness of the cache to mask the NUMA allocation problem. Applications with a larger footprint (class C) observe a higher average degradation on a single VM, 54% compared with 39% for class B. For the partitioned [2] system the degradation increases from 3% for class B to 10% for class C. The results also suggest that without partitioning, relying on runs on cold VMs as a cure becomes less optimal, as the first run slowdown compared to native increases from 9% for class B to 27% for class C.

A detailed analysis of page faults shows that in the partitioned case the correct locality is preserved, except for the inter-VM shared memory regions used inside MPI for communication. The remote NUMA accesses for communication are unavoidable in a parallel application and are an intrinsic characteristic of such applications: data has to move between cooperating tasks. In the MPI case, the communication buffers are used pairwise by tasks. Although MPI applications are usually optimized to minimize communication, a particular concern is when the communication buffers do not have affinity with any of the endpoints. Since our implementation of inter-VM shared memory is persistent and it has a sticky mapping between runs, this situation can be easily avoided by extending the MPI communication buffer allocation with awareness of the inter-VM shared memory layout.

With partitioning, both cold and warm VMs are able to provide the same level of performance, as shown in Figure 12. Furthermore, any run on a partitioned system matches or exceeds (in two cases) the performance of runs on a single cold VM spanning all the cores.

For lack of space, we do not include detailed results for partitioning on UMA systems. For our quad-socket quad-core system, performance compared to native is slower by an average of 2.2%, while performance with one VM spanning all 16 cores is on average within 6% of native. We attribute the better behavior with partitioning to less contention on shared data structures.

When running in a cluster environment using a two node UMA system (32 cores), part of the communication is done across low performance virtualized IO cards. In the next section we present how partitioning is able reduce the average performance impact from 63% with one machine per node to 12%.

IX. Multinode VM Partitioning

On clusters, virtualization also affects networking and I/O performance. Figure 13 shows the performance of the MPI implementation of the NAS benchmarks class C on the two node quad-socket quad-core Intel Tigerton cluster, already described in Section IV. We present the performance with partitioning for fully-virtualized and para-virtualized I/O configurations. With full virtualization, the RTL8139 Ethernet adapter is software emulated by the VM and the guest OS does not require modifications. For para-virtualization we have used the *virtio* [27] driver.

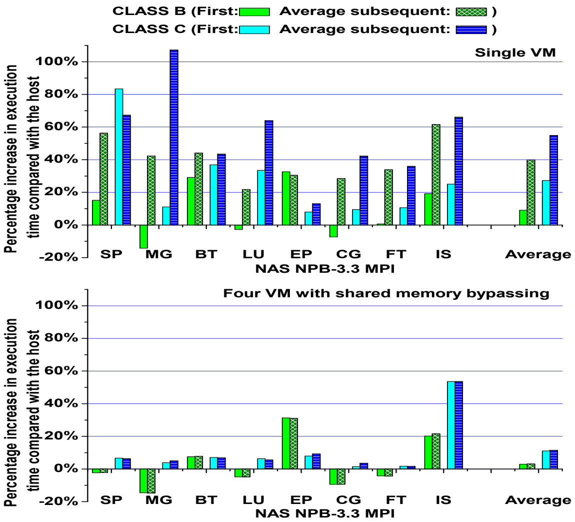

Fig. 12. *Performance of class B* vs. *class C for NPB MPI 3.3 for the base system (1 VM per 4 sockets) compared with partitioned system with shared memory bypassing (1 VM per socket). Architecture is based on AMD Opteron E8350.*

As shown, partitioning cores in independent VMs does improve the performance (I/O) on clusters. For *virtio*, the performance degradation is reduced from 63% with a single VM per node (4 socket per VM) to 34% with 2 VMs per socket (2 cores per VM). The impact of partitioning on full virtualization is more evident as the slowdown is decreased from 224% with a single VM per node to 17% when using 2 VMs per socket.

As partitioning provides better concurrent access to the NIC, these results (with shared inter-VM memory support) were intuitively expected: we discuss the impact of I/O concurrency in Section IX-A. The surprising result is that partitioned fully virtualized configurations provide comparable or better performance than partitioned para-virtualized configurations: the consensus [32], [33] is that para-virtualization always offers better I/O performance than full-virtualization. We analyze this performance inversion in Section IX-B.

A. Impact of I/O Concurrency

Partitioning creates multiple independent system images on a node, where each system provides its "own" NIC: all I/O requests within a guest proceed independently and concurrently to the hardware. KVM offers the capability to configure a VM with multiple NICs: in this case all I/O requests addressed to *different* NICs proceed independently and concurrently to the hardware. Intuitively, partitioning increases concurrency at the guest VM and at the hypervisor level, while multi-NIC configurations increase I/O concurrency only at the guest level.

Figure 14 shows the performance when using multiple NICs (8 NICs is the maximum supported by `qemu`). The 8-NIC

[2] IS performance is caused by un-tuned collective operations in the partitioned system.

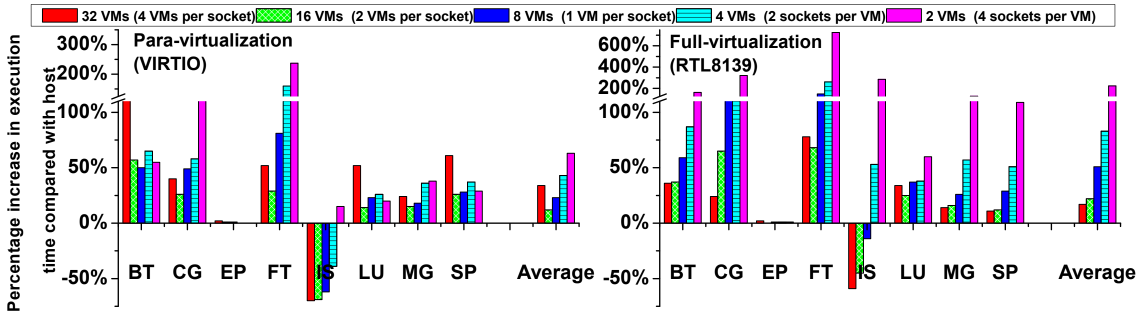

Fig. 13. *Performance with different partitioning layout for virtio and rtl8139.*

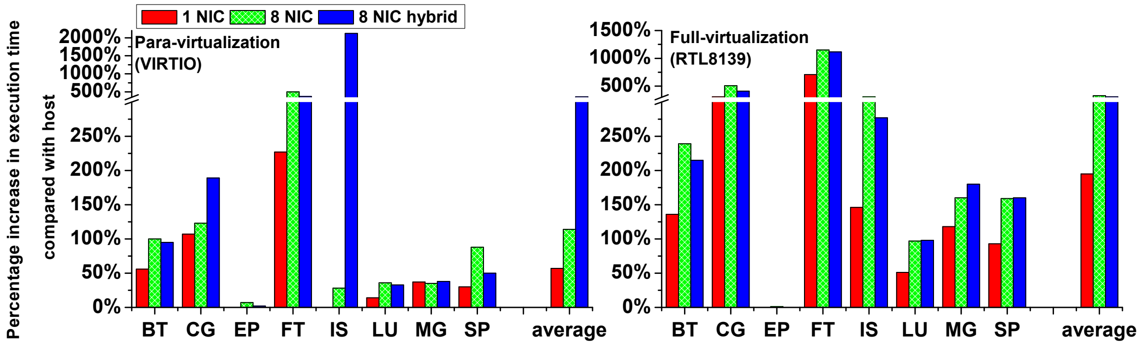

Fig. 14. *Performance when using multiple NICs (concurrent virtual channels) for virtio and rtl8139.*

hybrid mode presents a configuration where processes within a node explicitly bypass the network and use shared memory: surprisingly this was not automatically done by MPI and we had to extend the `mpich2` implementation.

As shown, using multiple NICs degrades the performance of all applications when compared to using a single adapter. This impact is higher for para-virtualization (3.62X) than for full-virtualization (3.02X).

Part of this behavior was expected, part is surprising. We expected multiple NICs to provide similar or better performance than a single adapter and lower performance than partitioning. Current solutions (KVM and VMware) use a service thread for networking requests: partitioning provides multiple threads (one per VM) and it was expected to provide a better concurrent access to network resources. We expected single and multi-NIC configurations to exhibit similar contention for the service thread and comparable performance. As the current runtime implementations deal poorly[3] with multi-NIC configurations we were not able to determine the causes of the performance loss in this case. Adding multiple network service threads to a VM might improve the performance of configurations using multiple NICs. Overall, this experiment clearly shows that the improvement achieved by VM partitioning cannot be achieved by multiple-emulated NICs.

B. Para- and Full Virtualization of Network I/O

In full-virtualization schemes, the guest OS is unaware of running in a virtualized environment: the hypervisor intercepts any I/O request and traps to emulate the network device. While the guest OS does not require any modifications, the overhead of communication is expected to be high. In para-virtualization, the guest and the hypervisor cooperate to service network requests. The guest is supplemented with an additional driver that communicates with the underlying hypervisor through hypercalls.

With full-virtualization, I/O requests are usually serviced eagerly. Para-virtualized environments usually perform throughput optimizations by buffering and coalescing requests: *virtio* provides queues to buffer requests. The driver is split into a front-end within the guest and a back-end within the hypervisor. These ends share a buffer that allows doing scatter-gather operations. Flushing these buffers, through calling a `kick` routine to start the communication, is tuned to improve overall throughput. In general, para-virtualization is considered to improve I/O performance for commercial applications.

Eager and lazy service policies for communication requests lead to systems with different latency and bandwidth characteristics. We measure the system bandwidth using the OSU 3.5 MPI performance [26] suite. Figure 15 shows that para-virtualization and buffering leads to a high latency system: eager service equates low latency. Figure 16 shows that para-virtualization leads to a high throughput system: eager service equates low bandwidth. Note that full-virtualization still provides better bandwidth for small messages.

[3] OpenMPI does not handle it correctly or hangs, mpich2 supports it with significant overhead and we had to implement the "hybrid" mode for shared memory bypass.

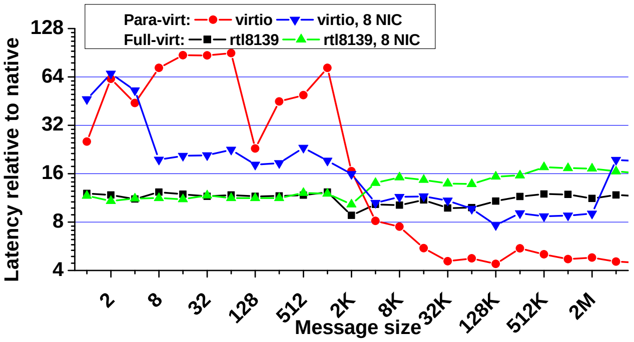

Fig. 15. *Multi vs. single NIC multi-latency between two nodes, each running a single 16 core VM.*

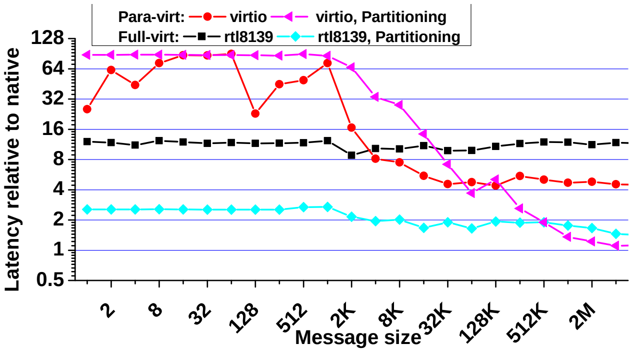

Fig. 17. *Partitioning vs. single VM message latency between two 16 core nodes.*

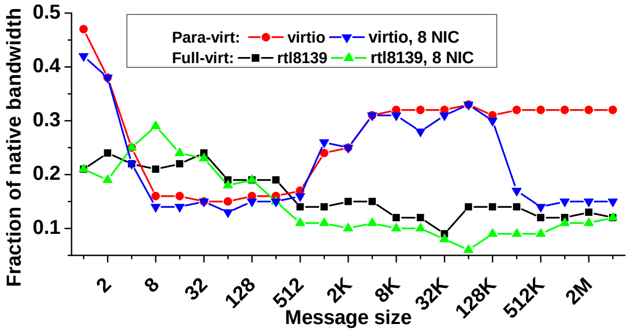

Fig. 16. *Multi vs. single NIC multi-bandwidth between two nodes, each running a single 16 core VM.*

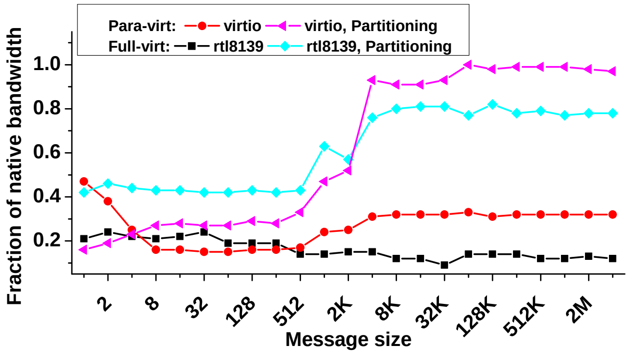

Fig. 18. *Partitioning vs. single VM bandwidth between two 16 core nodes.*

Trying to achieve the performance gains of partitioning through the use of multiple NICs is not possible. As shown in Figure 15, the latency of using 8 NICs (maximum allowed by qemu) is higher than using single NIC. For large messages, using multiple NICs causes 2X increase in latency on average. For bandwidth, as shown in Figure 16 the bandwidth is reduced with multiple NICs for almost all message sizes. The impact on performance is more profound for para-virtualized *virtio* because of the overhead emulating multiple *virtio* devices sharing a bus.

These results suggest that partitioning performance cannot be achieved simply by introducing concurrency in the guest. The next section discusses the interaction between partitioning and virtualization IO.

C. Partitioning and Virtualized I/O

Partitioning improves both the latency and the bandwidth provided by virtualized environments, especially for full-virtualization. As illustrated in Figure 17, partitioning improves the latency of *virtio* by as much as 4X for small messages, albeit in a nonuniform manner with respect to the message size. For messages larger than 128K, partitioning hurts the latency of *virtio*. In this case partitioning introduces multiple VM buffers and changes (delays) the timing of the invocation of the `kick` routine to deliver the messages. It is possible that tuning the VM buffer size can alleviate this behavior. The latency of "Full-virt" is consistently improved by at least 5X and by as much as 8X.

The bandwidth improvements are shown in Figure 18: *virtio* observes improvements as high as 3X, while "Full-virt" observes improvements as high as 5X. In particular, note that para-virtualization combined with partitioning achieves 97% of the native bandwidth.

As shown before, in application settings partitioning reduces the performance degradation with *virtio* from 63% to 34% of native. The impact of partitioning on full-virtualization is more evident as the slowdown is decreased from 224% to 17% of native. This trend is explained by the microbenchmark results which indicate that the performance of fully-virtualized I/O is improved more than the performance of para-virtualized I/O.

Partitioning supplements the parallelism at the guest OS level with multiple service threads interacting with the hypervisor (qemu/KVM in our case). In contrast, the multi NIC configuration has only one service thread. In addition, multi NIC emulation introduces additional overhead when emulating the shared bus. This is why multi NIC configurations provide lower performance than single NIC configurations.

Overall, we conclude that the sub-optimal throughput associated with full-virtualization can be eliminated by the parallelization introduced through VM partitioning. Partitioning with *rtl8139* provides best throughput for small messages and best latency for most message sizes. This lead to observing the best overall performance for the studied applications. Furthermore, partitioning enables native throughput (97%) for para-virtualized environments. In contrast, the best throughput reported to date for *virtio* is around 70% of native.

X. To Paravirtualize or Not?

As virtualized environments have to bridge the *semantic gap*, they offer degraded memory locality and I/O performance. Partitioning clearly provides performance improvements in para-, as well as fully virtualized environments. For I/O in particular, partitioning enables the full virtualization approach to attain performance comparable to para-virtualization: this contradicts the consensus that para-virtualization is required for best performance. As partitioning also improves memory locality and performance on NUMA architectures, the question remains whether engineering complex para-virtualization software can provide significant performance improvements when compared to fully virtualized memory management.

When "preserving" memory locality, our experimental evaluation for the MPI workload on KVM shows that *cold* runs observe a 27% average performance degradation on an quad-core quad-socket AMD NUMA system. Runs on *warm* VMs observe an average performance degradation of 54% caused by locality leakage and full virtualization approaches have little potential for improving this behavior.

Para-virtualization is required to improve the performance of *warm* runs: its biggest drawback is that it requires modifications in any guest OS, e.g. Linux and Windows. Furthermore, based on our understanding of the Linux kernel code, these modifications will require a significant if not complete re-implementation of the memory management code. As shown in Figure 4, for runs on a warm VM, most of the page faults are filtered by the guest OS and are not observed at the hypervisor level. Inside the guest OS, we can determine when a page is no longer needed, e.g. at application termination. As the actual NUMA nodes are available only at the host, the guest needs to communicate this page (or page group) for checks and possible reclamation. Currently, it is not possible to synchronously communicate these pages, as no traps are supported for page freeing and pages might not get reclaimed immediately inside the host in the KVM case.

We perform a simple experiment to disable caching of the page mappings inside the guest OS and cause the propagation of faults to the hypervisor. Page mappings are kept inside kernel memory which in Linux is managed by the *slab* allocator. The *slab* allocator provides per-core page caches, as well as a global page cache. Linux is configurable and provides the option of disabling the per-core caches and using only the global cache: this is referred to as the *slub* allocator. When using the *slub* allocator a higher percentage of faults is propagated but the overall result is that performance is lower than in any of the *slab* runs.

This indicates that a more specialized approach to provide selective page unmapping is required. This could be implemented using a hypervisor daemon coordinating with the guest kernel but we consider such an approach way beyond the scope of this paper. Beyond the challenges posed by the software architecture of the current Linux memory management code, an asynchronous daemon approach faces the "semantic gap" challenges (lack of information about the guest activities): it cannot tell if a page is used by an application if the application is not scheduled on any guest virtual CPU; if a page is in use, it cannot determine its desired locality or whether it is shared. This also has the potential of significantly slowing down the system for the common case of pages that have the "right" NUMA affinity. A solution based on "enlightenment", while still breaking the virtualization abstractions is more tractable due to better contained software changes to Linux.

Furthermore, we expect a selective unmapping approach to provide a level of performance situated between the performance of runs on warm VMs and the performance of runs on cold VMs. We consider the performance of *cold* runs as a good indicator of performance expectations for a paravirtualized selective unmapping approach.

Overall, our experiments indicate that partitioning is able to provide good performance in fully virtualized environments and it does eliminate most of the need for paravirtualization.

XI. Conclusions

In this paper we evaluate the impact of virtualization on the performance of parallel scientific applications on multi-socket multicore systems, and we advocate the adoption of resource partitioning. As a workload we use implementations of the NAS Parallel Benchmarks in MPI, UPC and OpenMP. Our results on single node UMA systems confirm previous results and we find an average slowdown of 6% when comparing to native performance. The NUMA support in current virtualization solutions is incomplete and this translates into an average performance degradation of 47% for the whole NPB workload (B and C), when compared to native. This impact is much higher than that previously reported: the difference is attributed to the higher node core count currently available.

We further evaluate techniques to improve locality in full virtualization environments: hypervisor level page migration and system partitioning. We also provide a thorough discussion of the interaction between the implementations of programming models and virtualized environments. Our results indicate that were NUMA support improved in current implementations, the average slowdown compared to native is still at 27%. Using partitioning with efficient inter-VM communication, the average performance on the NUMA system is within 3% and 11% of native for class B and C respectively, while on the UMA system is within 2.2% of native.

For the NPB workload, our analysis of paging behavior indicates that improving the NUMA support only at hypervisor level is unlikely to mitigate most of the performance impact of virtualization. A more complete solution requires both hypervisor and guest OS modifications (para-virtualization) and it breaks the central tenet of hiding the system resource management from guests. Thus, this approach is likely to face resistance from commercial implementors whose target applications are not HPC centric.

When the programming model allows, *e.g.* MPI or Partitioned Global Address Space languages or hybrid approaches (MPI+OpenMP, PGAS+OpenMP), partitioning is a simple and robust approach to improve locality and I/O performance: this translates directly into end-to-end application performance. With partitioning the average impact of virtualization on a two-node quad-socket quad-core cluster is as low as 17%.

Besides cloud computing environments, we believe that we provide compelling evidence in favor of partitioning and adding shared memory support for inter-VM communication in other solutions specifically designed for high performance computing.

References

[1] J. Matthews, T. Garfinkel, C. Hoff, and J. Wheeler, "Virtual Machine Contracts For Datacenter And Cloud Computing Environments," *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pp. 25–30, 2009.

[2] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the Migration of Virtual Computers," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 377–390, 2002.

[3] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase, "Virtual Machine Hosting for Networked Clusters: Building the Foundations for "Autonomic" Orchestration," *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, p. 7, 2006.

[4] "National Impact Series: Scientists Look To The Clouds To Solve Complex Questions," Available at http://www.er.doe.gov/News_Information/-News_Room/2009/Oct%2014_ComplexQuestions.html, 2009.

[5] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *Technical Report NAS-95-010, NASA Ames Research Center*, 1995.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 164–177, 2003.

[7] Kernel Based Virtual Machine, "http://www.linux-kvm.org/," 2008.

[8] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[9] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda, "Virtual Machine Aware Communication Libraries For High Performance Computing," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[10] C. Xu, Y. Bai, and C. Luo, "Performance Evaluation of Parallel Programming in Virtual Machine Environment," *NPC '09: Proceedings of the 2009 Sixth IFIP International Conference on Network and Parallel Computing*, pp. 140–147, 2009.

[11] L. Youseff, K. Seymour, H. You, D. Zagorodnov, J. Dongarra, and R. Wolski, "Paravirtualization Effect On Single- And Multi-Threaded Memory-Intensive Linear Algebra Software," *Cluster Computing*, vol. 12, no. 2, pp. 101–122, 2009.

[12] D. Rao and J. Nakajima, "Guest NUMA Support (PV) and (HVM) ," Xen Summit North America 2010.

[13] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz, "Tessellation: Space-Time Partitioning in a Manycore Client OS," in *Proc. of the first USENIX Conference on Hot Topics in Parallism (HotPAR*, 2009.

[14] A. Schpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing Diversity In The Barrelfish Manycore Operating System," in *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[15] D. Wentzlaff and A. Agarwal, "Factored Operating Systems (fos): The Case For A Scalable Operating System For Multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, 2009.

[16] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel Virtualization Technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[17] AMD-Vł Nested Paging, "developer.amd.com/assets/npt-wp-1%201-final-tm.pdf," 2008.

[18] I. Habib, "Virtualization with kvm," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[19] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems," in *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2006, p. 1.

[20] W. Huang, J. Liu, B. Abali, and D. K. Panda, "A Case For High Performance Computing With Virtual Machines," in *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2006, pp. 125–134.

[21] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin, "Xensocket: A High-Throughput Interdomain Transport For Virtual Machines," *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pp. 184–203, 2007.

[22] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-Domain Socket Communications Supporting High Performance And Full Binary Compatibility On Xen," *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 11–20, 2008.

[23] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, , and R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing," in *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[24] B. M. Posey, "Virtualization: Optimizing Hyper-V Memory Usage," *Microsoft TechNet Magazine http://technet.microsoft.com/en-us/magazine/-hh709739.aspx*, Dec 2011.

[25] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[26] "OSU MPI benchmarks, OMB 3.5," Network-Based Computing Laboratory, Ohio State University, http://mvapich.cse.ohio-state.edu/benchmarks/.

[27] R. Russell, "VIRTIO: Towards a De-facto Standard for Virtual I/O Devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.

[28] K. Z. Ibrahim, S. A. Hofmeyr, and C. Iancu, "Characterizing the Performance of Parallel Applications on Multi-socket Virtual Machines," in *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011, Newport Beach, CA, USA, May 23-26, 2011*.

[29] F. Diakhaté, M. Perache, R. Namyst, and H. Jourdren, "Efficient Shared Memory Message Passing for Inter-VM Communications," *Euro-Par 2008 Workshops - Parallel Processing*, pp. 53–62, 2009.

[30] V. S. Junior, L. C. Lung, M. Correia, J. da Silva Fraga, and J. Lau, "Intrusion Tolerant Services Through Virtualization: A Shared Memory Approach," *Advanced Information Networking and Applications, International Conference on*, pp. 768–774, 2010.

[31] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D. K. Panda, and P. Wyckoff, "Microbenchmark Performance Comparison of High-Speed Cluster Interconnects," *IEEE Micro*, vol. 24, no. 1, pp. 42–51, 2004.

[32] B. Zhang, X. Wang, R. Lai, L. Yang, Y. Luo, X. Li, and Z. Wang, "A Survey on I/O Virtualization and Optimization," *ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual*, pp. 117 –123, july 2010.

[33] P. Muditha Perera and C. Keppitiyagama, "A performance comparison of hypervisors," *The 2011 International Conference on Advances in ICT for Emerging Regions (ICTer)*, p. 120, Sept. 2011.