Characterizing the Performance of Parallel Applications on Multi-Socket Virtual Machines

Khaled 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 management of scientific computing applications running in virtualized environments. NUMA locality on current solutions (KVM and Xen) is enforced by pinning virtual machines to CPUs and providing NUMA aware allocation in hypervisors. Our analysis shows that due to two-level memory management and lack of integration with page reclamation mechanisms, applications running on warm VMs suffer from a "leakage" of page locality. Our results using MPI, UPC and OpenMP implementations of the NAS Parallel Benchmarks, running on Intel and AMD NUMA systems, indicate that applications observe an overall average performance degradation of 55% when compared to native. Runs on "cold" VMs suffer an average performance degradation of 27%, while subsequent runs are roughly 30% slower than the cold runs. We quantify the impact of locality improvement techniques designed for full virtualization environments: hypervisor level page remapping and 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 re-produce native performance: in a partitioned system with one virtual machine per socket the average workload performance is 5% better than native.*

I. Introduction

Virtualization technologies are ubiquitously deployed in data centers and offer the benefit of resource consolidation [17], performance and fault isolation, flexible migration [20] and easy creation [6] 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 [16] project. In commercial workloads, the tasks are often independent and serve short lived requests; server tasks are started at virtual machine boot time and are 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 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. One facet of this semantic gap is exposed by the available NUMA support in existing open source solutions (Xen [4], KVM [11]), as well as in proprietary (VMware ESX [23] and hyperV) solutions.

Currently, NUMA affinity in virtualized environments is achieved by a combination of pinning guest VMs to CPUs and having the hypervisor memory management allocate memory with affinity to the faulting CPUs. NUMA support in the OSes running inside VMs 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 developers email lists indicates that there is widespread belief that this cooperation provides most of the NUMA support needed. Recently, "enlightenment" [18] 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.

In this paper we characterize the performance of HPC applications in virtualized NUMA environments and quantify the performance expectations of several Xen and KVM solutions designed to improve memory locality. For our evaluation we use a workload containing implementations of the NAS Parallel Benchmarks [3] in MPI, UPC and OpenMP.

In Section V we discuss the performance using existing virtualization technologies on AMD and Intel NUMA and UMA processors. In contrast with previous HPC studies [25], [26], [8] which report little or no impact from virtualization on UMA or NUMA architectures with four or less cores, our results indicate a significant performance degradation (up to 82% on KVM and 4x on Xen) when VMs span sockets in 16 core NUMA architectures. With virtualization, our analysis also indicates that programming models designed for cluster environments such as MPI or Partitioned Global Address Space languages provide better scalability and performance than shared memory programming models such as OpenMP.

Our analysis in Section VI shows that applications that start right after booting on a *cold* KVM 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 reasonably good upper bound for the expectations on performance gains when improving the NUMA support in virtualization technologies.

We then explore techniques designed to improve locality in full virtualization environments: 1) hypervisor only approaches (Section VII); and; 2) system partitioning (Section VIII) which is applicable for full virtualization but it requires support from the parallel programming models. Note that none of these approaches require exposing the hardware topology to the guests and cover the whole spectrum of possible solutions were "enlightenment" or other paravirtualization approaches deemed undesirable.

The analysis in Section VI indicates that up to 25% of the pages used in runs on warm VMs have bad locality. Up to 90% of the page translation activities on a warm VM are filtered by the guest OS. Hypervisor only schemes, while the most portable, have no performance potential: on average only 2% of the program page translation activities can be correctly handled at this level.

In Section VIII we explore an orthogonal technique for providing affinity: system partitioning using multiple guest VMs. In order to achieve good performance we had to extend the KVM/Xen support and implement shared memory bypass in the MPI and UPC runtimes. Partitioning [21], [24], [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 better 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.

The main contributions of this paper are the characterization of locality in NUMA environments and the quantification of the performance expectations for several KVM and Xen 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 direct interest to the proponents of techniques to provide contracts between virtual machines and hypervisors.

II. Related work

Memory translation in virtualized environments has been extensively studied. To accelerate virtualization, Intel provides the VT-X technology [22], while AMD adopted AMD-V [1]. Both provide hardware mechanisms for hypervisor level page table traversal used by the current solutions: VMware [23], Xen [4], and KVM [7].

The impact of virtualization for scientific workloads has received its fair share of attention. Xu et al. [25] study the performance of multiple programming paradigms on VMs. Youssef et al [27] evaluate the impact of Xen on MPI performance and report a low overhead of virtualization. They [26] also evaluate the impact of virtualization on multithreaded linear algebra software and report low overhead on UMA systems. These studies have been conducted on NUMA systems with four or less cores or UMA [26] architectures.

Huang et al present several Xen extensions to improve virtualization performance. They present Xen-IB [9] 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 [8] Inter-VM communication (IVC) using MVAPICH and Xen extensions to provide shared memory between virtual machines. 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 [28] for XenSocket, and by Kim et al[12], again at low core counts. Besides providing an OpenMPI implementation, we evaluate performance when completely bypassing the networking (IP) stack with shared memory.

Lange et al [13] 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 available per node and Palacios is not yet tuned for multicore systems.

In general, most of the cited studies [25], [26], [27], [9], [8], [13] 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. Barrelfish [21], `fos` [24] and Tessellation [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. The systems designed specifically for HPC 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 OSes (Linux) and commercial virtualization technologies (KVM,Xen) in configurations where a virtual machine spans all the available cores.

III. Memory Management in Virtual Machines

Understanding application performance in multi-socket virtual machines requires revisiting the memory management mechanisms. Without loss of generality we focus our discussion on KVM, which exploits the latest AMD and Intel virtualization technologies.

One of the most difficult problems addressed in virtualized environments is handling memory translation. 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 (process in KVM terminology) to the hypervisor and a two level page translation scheme is required. For protection, on any system only one trusted entity is allowed to manage physical paging. In shadow paging schemes, the hypervisor intercepts and handles paging whenever guests try to access the register affecting the memory mappings. This solution involves a high overhead to satisfy a memory fault: in the HPC realm, Lange et al [13] show that software translation introduces additional runtime overhead in most cases.

AMD and Intel provide hardware support for only one level of traversal of page tables: these extensions allow near native performance. KVM is designed only for processors with hardware support for virtualization, while other solutions

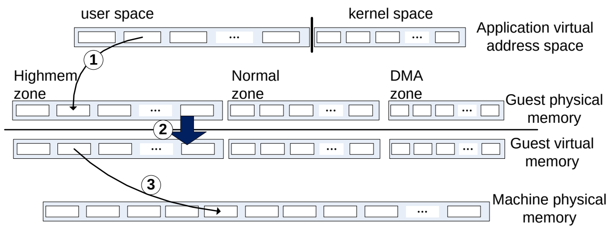

Fig. 1. *The three page-translation stages for guest application virtual addresses in KVM. The guest OS is responsible for the first translation phase. The KVM device driver manages the two other phases with the assistance of the host OS (hypervisor). Note, only stage two of the mapping is static.*

(VMware and Xen) provide additional support for software address translation.

A. The KVM Memory Management Unit

Most of the results presented in this paper are based on QEMU/KVM [7], [11]. 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 depend on the simulated architecture: some regions require direct mapping, e.g. IO or DMA, while most of the physical memory is virtualized. The QEMU/KVM solution allocates all the required 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 host virtual memory, performed by the KVM modules, if hardware acceleration is active; and finally, 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 this memory does not have a 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 from the host OS the actual physical memory for this particular host virtual address. 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 that some memory pages need to be reclaimed 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 needed for reclamation. The driver keeps a reverse mapping of all virtual CPUs address mappings that are using this page, and starts invalidating the page' PTEs in the guest before returning the page for reclamation by the hypervisor. Obviously, the hypervisor involvement is not needed if page translations get cached in the guest OS. Additionally, if the same host page is reused multiple times by the guest, the host translation also remains intact.

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 affinity to the physical CPU. 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. 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.

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 maintaining a good NUMA allocation provides only a best-effort guarantee.

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. Furthermore, due to licensing restrictions we do not present any VMware of hyperV results[1]. In KVM, when faults are propagated, the hypervisor provides memory affinity with the faulting CPU. In Xen, 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 identity of the faulting CPU.

IV. Experimental Setup

We experiment with two open source virtualization technologies: KVM and Xen 4.0 using the hardware virtualization

[1]VMware requires legal approval when publishing performance results. Free licenses allow only 2 vcpus, while commercial licenses are limited to eight vcpus. Affinity management is delegated to guests in VMware. HyperV is restricted to 4 vcpus.

support provided by CPU vendors: Intel VT-X and AMD-V.

For the guest OS used inside the virtual machines 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 [3] 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 overall the memory footprint of the workload varies from tens of MBs to tens of GBs. Asanović et al [2] 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 low (less than 10% in all cases) and all the results and trends presented are statistically significant.

V. Performance on Multi-Socket Nodes

Figure 2 shows the comparison with native performance when running on KVM with virtual machines spanning an increasing number of sockets. When using a single socket, the performance on virtual 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 [26], [9], [13] 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, 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 results and omitted for brevity.

A summary of similar experiments using Xen-based virtualization is shown in Figure 3. In Xen, all CPUs are exposed as virtual CPUs in dom0—the only privileged domain that controls all virtualization activities. Unlike KVM, explicit control over the virtual to physical CPU mapping is not available. We ran the set of benchmarks, both on NUMA and UMA systems for 16-way parallelization. Unsurprisingly, we saw high degradation with NUMA runs, while UMA runs were at most 20% slower. Booting the hypervisor with explicit NUMA awareness, which is the default on Xen 4.0, does not cure the problem. Without delving into the battle of KVM versus Xen, Xen performance is generally better than KVM on the UMA architectures, while NUMA performance of Xen is much worse. Noting the $\log_2$ scale of the *y-axis* in Figure 3, we observe performance up to 4x slower than native. Therefore, we emphasize the KVM evaluation in the rest of this paper.

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 many configurations for virtualization including multiple pinning strategies, different paging granularities and emulated NUMA on the guest. 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 instrumented 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 4 and 5 show the paging activity observed at all translation levels, when running 16-way parallel MPI jobs (NPB class B) on the AMD 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 4 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" and the pages are correctly mapped on the NUMA node local to the faulting CPU. 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 5, 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

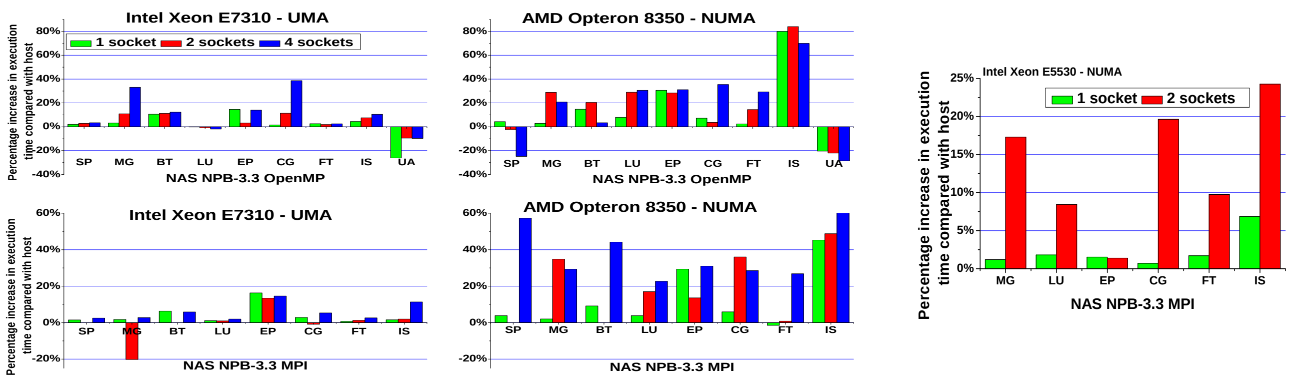

Fig. 2. *Performance of NAS NBP3.3 benchmarks MPI and OpenMP implementations running with KVM on guest virtual machines on two machines, each with 4 sockets, representing NUMA and UMA architectures.*

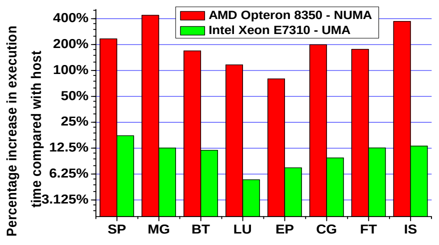

Fig. 3. *Performance relative to native when running on Xen 4.0 dom0. Each VMs is spanning 4 sockets and each application is 16-way parallel.*

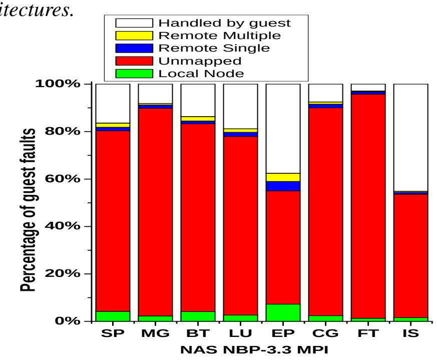

Fig. 4. *The page translation activities for a run on a cold VM: first application running after booting the VM*

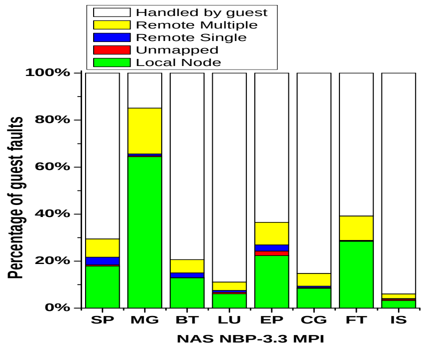

Fig. 5. *The page translation activities as seen by the KVM device driver for the second run of an application on the guest OS.*

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 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 6. 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. Thus all results are statistically significant and indicate that improving the NUMA support in virtualized environments is likely to produce performance benefits when compared to the existing solutions.

For the OpenMP benchmarks cold runs are indistinguishable from warm runs, the difference is explained in the next section.

The temporal distribution of page faults is determined by the application's memory footprint and access pattern. In this study 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 6: 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 7 presents the distribution of the page faults intercepted by the hypervisor with respect to the NUMA nodes

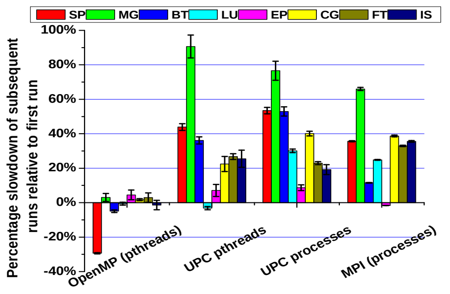

Fig. 6. *Execution time increase of runs on warm VMs compared to the first run after booting the VM. MPI and UPC applications become slower.*

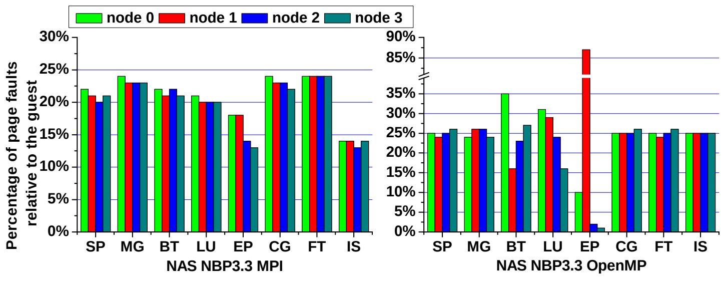

Fig. 7. *NUMA node distribution of page faults served by hypervisor for MPI and OpenMP cold runs. Faults are separated by NUMA nodes, evenly allocated pages show better NUMA allocation.*

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* with either processes or `pthreads` exhibit indistinguishable performance and we observe a pronounced difference between *cold* and *warm* runs, as shown in Figure 6. 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

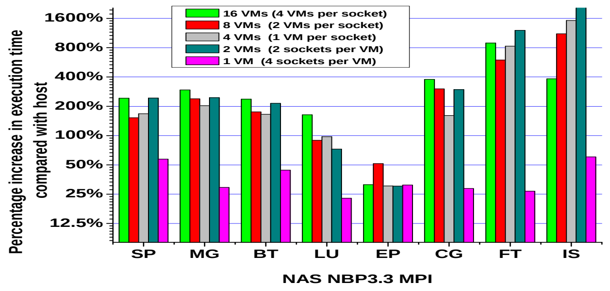

Fig. 8. *Performance of various VM configurations on the 4-socket Quad-core AMD Opteron 8350. Inter-VM communication uses virtio.*

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 5, 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. Resource 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 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 8 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 about 70% of the native hardware bandwidth, the performance degradation is large (up to 16 times) when increasing the number of VMs. Many other [28], [12], [19], [5] 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` [10] 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 [4] allows sharing pages between only two VMs [28], [5], [12] using the *Grant-Table* and it imposes severe restrictions on the amount of memory allowed. We have also modified Xen to provide multi-VM sharing and increase the amount of memory shared. The results on Xen show identical trends to the KVM results and we will not discuss further Xen details.

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 9 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

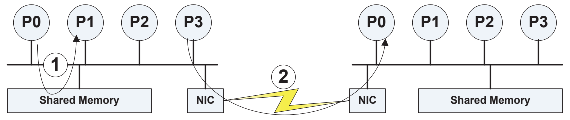

Fig. 9. *MPI Communication between processes without virtualization: 1- MPI communication within a SMP node uses shared memory; 2- MPI communication across SMP nodes uses the fastest available network card (using one of the tcp, IB, ..., etc BTLs).*

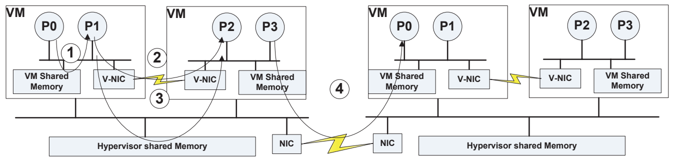

Fig. 10. *MPI communication with inter-VM bypass: 1- communication within a VM using a shared memory BTL; 2- Communication between VMs using virtual NIC ; 3-* newly introduced shared-memory communication BTL for communication between VMs*; 4- communication between VMs across nodes using NIC interfaces.*

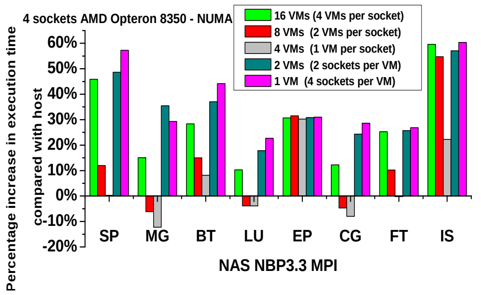

Fig. 11. *The performance of MPI NAS benchmarks with different virtual machine configurations on the 4-socket Quad-core AMD Opteron E8350. Inter-VM communication uses shared memory.*

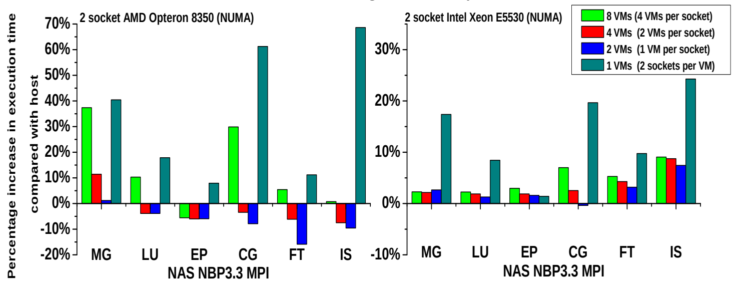

Fig. 12. *The performance of MPI NAS NBP benchmarks with different virtual machine configuration on 2-socket Quad-core AMD Opteron E8350 and Intel Xeon E5530.*

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 10. Two of the layers shown in the figure use shared memory; 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 [14] 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. MPI Performance in a Partitioned Virtual Environment

Figure 11 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. Figure 13 shows the impact of partitioning on page fault latency measured with `lmbench`. We have extended `lmbench` to take into account the various types of memory used in the implementations of parallel programming languages: 1) regular memory obtained using `malloc`; 2) device based `mmap` memory used for the KVM inter-VM communication; and 3) anonymous `mmap` memory traditionally used for shared memory inter-process communication in MPI or within one VM. As illustrated, partitioning reduces page fault latency. Note the high latency for faults on device mmaped memory necessary for inter-VM communication: in the MPI case this is used only for bounce buffers, while in the PGAS (UPC) case a significant fraction of the heap memory is obtained using this mechanism. 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 12 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

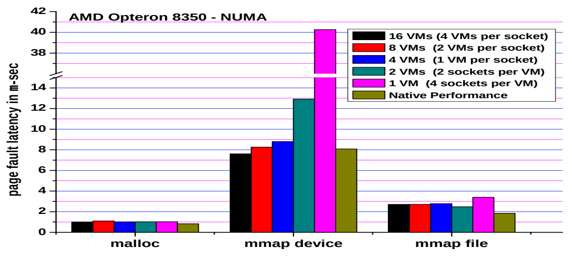

Fig. 13. *Page fault latency in μs for different VM configurations.*

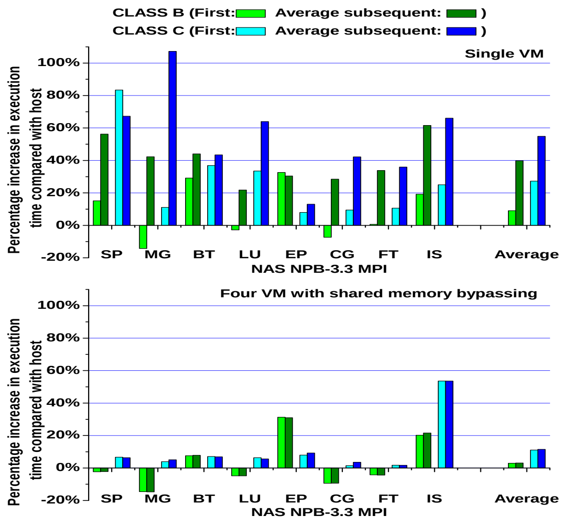

Fig. 14. *Performance of class B* vs. *class C for NPB MPI 3.3 for the base system (1 VM per 4 sockets) compared with partitioned system with shared memory bypassing (1 VM per socket). Architecture is based on AMD Opteron E8350.*

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 12 with the results in Figure 11, 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 14 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 14. 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 system (32 cores) and the network, partitioning is also able to lower the average performance impact from 63% with one machine per node to 12%.

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

IX. To Paravirtualize or Not?

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. The analysis presented in Section VI indicates that in these runs NUMA affinity is provided. Runs on *warm* VMs observe an average performance degradation of 54% caused by locality leakage and hypervisor only approaches have little potential for improving this behavior.

Paravirtualization is required to improve the performance of *warm* runs. The biggest drawback of paravirtualization 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 5, 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.

X. Conclusions

In this paper we evaluate the impact of virtualization on the performance of parallel scientific applications on multi-socket multicore systems. As a workload we use implementations of the NAS Parallel Benchmarks in MPI, UPC and OpenMP. Our results on UMA systems confirm previous results and we find an average slowdown of 6% compared to native for our workload. 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: 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, 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 and breaks the central tenet of virtualization: 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 worth considering as an orthogonal approach to improve performance. Besides cloud computing environments, we believe that we provide compelling evidence in favor of adding shared memory support for inter-VM communication in solutions specifically designed for high performance computing.

References

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

[2] 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. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] 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.

[4] 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*, pages 164–177, 2003.

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

[6] 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*, page 7, 2006.

[7] I. Habib. Virtualization with kvm. *Linux Journal*, 2008(166):8, 2008.

[8] 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*, pages 1–12, New York, NY, USA, 2007. ACM.

[9] 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*, pages 125–134, New York, NY, USA, 2006. ACM.

[10] 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*, pages 768–774, 2010.

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

[12] 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*, pages 11–20, 2008.

[13] 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.

[14] 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*, 24(1):42–51, 2004.

[15] 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.

[16] 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.

[17] 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*, pages 25–30, 2009.

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

[19] R. Russell. VIRTIO: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[20] 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.*, 36(SI):377–390, 2002.

[21] 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.

[22] 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*, 38(5):48–56, 2005.

[23] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[24] D. Wentzlaff and A. Agarwal. Factored Operating Systems (fos): The Case For A Scalable Operating System For Multicores. *SIGOPS Oper. Syst. Rev.*, 43(2), 2009.

[25] 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*, pages 140–147, 2009.

[26] 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*, 12(2):101–122, 2009.

[27] 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*, page 1, Washington, DC, USA, 2006. IEEE Computer Society.

[28] 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*, pages 184–203, 2007.