Vshmem: Shared-Memory OS-Support for Multicore-based HPC systems

Lamia Youseff[α,β]
lyouseff@csail.mit.edu
[α] MIT CSAIL
Cambridge, MA 02139

Rich Wolski[β]
rich@cs.ucsb.edu
[β] University of California, Santa Barbara,
Santa Barbara, CA 93106

***Abstract*—**

As a result of the huge performance potential of multi-core microprocessors, HPC infrastructures are rapidly integrating them into their architectures in order to expedite the performance growth of the next generation HPC systems. However, as the number of cores per processor increase to 100 or 1000s of cores, they are posing revolutionary challenges to the various aspects of the software stack. In our research, we endeavor to investigate novel solutions to the problem of extracting high-performance.

In this paper, we advocate for the use of virtualization as an alternative approach to the traditional operating systems for the next generation multicore-based HPC systems. In particular, we investigate an efficient mechanism for shared-memory communication between HPC applications executing within virtual machine (VM) instances that are *co-located* on the same hardware platform. This system, called *Vshmem*, implements low latency IPC communication mechanism that allows the programmer to selectively share memory regions between user-space processes residing in collocated virtual machines. Our contributions addressed

I. Introduction

Physical constraints like power and heat dissipation prevent hardware vendors from increasing the speed of the processor through increasing its frequency. Instead, the hardware industry has recently revisited the micro-processor design with the goal of finding alternative techniques to increase its performance growth. Multi-core microprocessor – i.e, configuring more cores per chip with shared memory between them– was the technique of their choice. As a result, multi-core became the norm in contemporary microprocessor architectures. Furthermore, the number of cores per chip is continuing to increase, such that it is even expected that the number of cores will double every year. Therefore, we expect to see chips with 100s or 1000s of cores in the next decade.

As a result of the huge performance potential of multi-core microprocessors, HPC infrastructures are rapidly integrating them into their architectures in order to expedite the performance growth of the next generation HPC systems. Starting in June 2007, the top500 supercomputers list became dominated with dual-core and quad-core microprocessors [1]. For example, the number of systems in the list deploying Quad-Core processors have grown between 19 systems in June 2007 to 336 systems in June 2009. Furthermore, it was expected that this trend is going to continue for the next decade, with core numbers in the range of one-hundred thousand to one million and more [1]. However, while multi-cores are offering unprecedented power and performance for HPC, they are posing revolutionary challenges to the various aspects of the software stack, including operating systems, compilers, tools, languages, runtime systems and applications.

The operating system is one of the most important components in the software stack, as it impacts all the other software components layered above it. Linux and Unix-like OS kernels have emerged as the operating system-of-choice for many HPC infrastructures as a result of its wide-range of readily available programming support tools and specialized libraries. Furthermore, it is becoming the preferred OS in academic and production scientific computing settings as a result of being an open-source freely-available and easy to use operating system. As a result, many of the scientific programmers are familiar with Linux as a development platform. However, Linux and Unix-like operating systems are monolithic kernels where all of the system tasks take place in the kernel space. To clarify, Figure 1 shows a simplified representation of a software stack based on monolithic OS-kernels. The operating system in monolithic kernels is responsible for managing and multiplexing the different processors /cores, hardware devices and components. It is also responsible for process management including process creation, scheduling, context-switching and termination, memory management, employing the protection mechanisms, and file-system management among other things. As a result, the monolithic-kernel operating system is a very complicated and huge software component. In addition, Linux systems have become a performance hurdle for high-performance computing applications because of their high OS-noise [2], [3], [4]. In fact, some studies have reported that this OS-noise is the primary bottleneck for application scalability in HPC systems [2]. Other studies have also shown that the huge memory-footprint and the decreasing cache efficiency of the Linux kernel is among the limiting performance bottlenecks for contemporary HPC applications [2], [3], [5], [**?**].

Given the increasing memory foot-print and system noise as well as the decreasing cache efficiency of the Linux kernels deployed in contemporary HPC systems, the HPC and OS community expect that these limitations will be magnified in the future [2], [3], [6], [7] as the number of cores in microprocessors increases. This becomes clear as we consider the diversity in the processes workloads deployed, served by

[0]This work was done while the author was at the University of California Santa Barbara and was funded in part by NSF Grants 0444412 and 0331645.

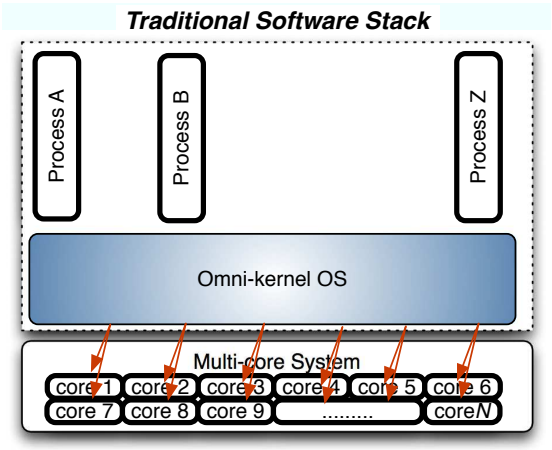

Fig. 1. This figure shows a simplified representation of the traditional software stack, demonstrating the deployment of one operating system which is managing all the cores and the processes.

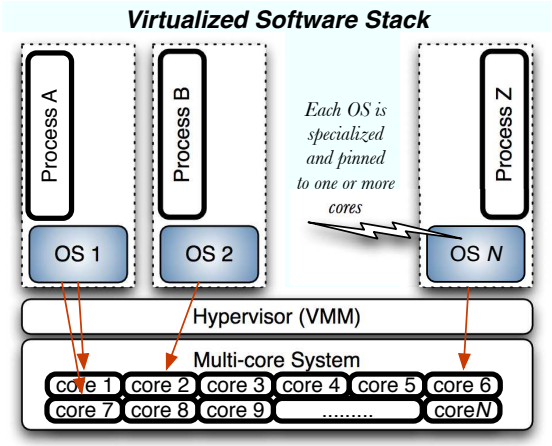

Fig. 2. This figure shows a simplified representation of the virtualized software stack, demonstrating the deployment of a hypervisor and several VMs, each of which is managing a subset of the cores and a subset of the processes.

the OS as well as the potential diversity of the architecture of the cores, frequency and memory hierarchy. Deploying one monolithic kernel like Linux to manage 10's to 1000's diverse cores and diverse processes in future multi-core microprocessor systems, will make the OS a significant performance bottleneck and will cause the overall performance of the system to deteriorate. That is to say that, alternative operating system approaches and software stacks *must* be explored to solve this performance quandary for these future multi-core microprocessors.

In the operating system community, several approaches were explored to address this problem and provide mechanisms to enhance the OS scalability, fault-tolerance, efficiency and performance, such as research in micro-kernels, single address-space operating systems, type-safe operating systems and virtualization. In our research, we chose to explore the virtualization approach because of the numerous benefits offered by virtualization to HPC infrastructures, such as proactive fault tolerance [8] and load-balancing through OS migration [9], [10] and VM consolidation for power-saving in HPC data centers [11], . Furthermore, virtualization ability to support specialized and customized OS kernels can provide enhanced scalability, reliability and low OS-noise [12], [13], [14], [15], [14], [16], [17] to HPC applications. Additionally, recent research in dynamic kernel adaptation [18] has uncovered potential performance benefits for applications deployed in the virtualized software stack.

Furthermore, the ability to continue to support UNIX programming in HPC has made the virtualized software stack appealing, as UNIX and UNIX-like OS kernels can be deployed within a virtual machine. Particularly, Linux has emerged as a nearly ubiquitous, open-source operating system with a wide-range of readily available programming support tools and specialized libraries. It is currently the system-of-choice in academic and production scientific computing settings and as a result, many – if not the majority of – scientific programmers, being trained today are familiar with Linux as a development platform. Therefore, virtualization will enable us to continue to support the familiar Linux API without possibly encountering the limitations of monolithic OS kernels. Furthermore, recent advances in OS research have addressed the performance issues – historically associated with virtualization– with novel techniques that reduce their performance overhead. One such technique is paravirtualization [19], [20] which is the process of strategically modifying a small segment of the interface that the VMM exports along with the OS that executes using it. Paravirtualization significantly simplifies the process of virtualization (at the cost of perfect hardware compatibility) by eliminating special hardware features and instructions that are difficult to efficiently virtualize. Paravirtualization systems thus, have the potential for improved scalability and performance over prior VMM implementations. Our own research [21], [22], as well as other groups [23], [24], [25], [26], [27] have advocated the benefits of virtualization for HPC and rigorously investigated its performance overhead for HPC micro-benchmarks, macro-benchmarks and common HPC applications. They also showed that various scientific and HPC workloads deployed on virtualized software stack demonstrated a statistically insignificant performance degradation. We have also carefully studied its impact on the different levels of the memory hierarchy for memory-intensive and memory-sensative application in [28], [29].

In sum, the virtualized software stack can offer many benefits to HPC community and enhance the reliability of the current infrastructures. However, in order for virtualization to be successful in multi-core microprocessor in HPC environments, *all* software components must be able to leverage the shared memory offered by the architecture in order to be able to extract the performance potential of the multi-core microprocessors.

In a virtualized environment, there is no such direct support of inter-VM shared memory although it is available at the hardware level. This is a result of the *perfect* memory isolation

dictated by the virtualization technology between the virtual machines. Therefore, user-space processes, which are part of the same HPC application and executing in collocated virtual machine on the same hardware, endure unnecessary communication overhead and higher latency than processes running within the same monolithic kernel. This aspect is currently one limitation to the efficient utilization of the virtualized software stack in HPC. In order to solve this performance quandary, we need a user-controlled memory sharing mechanism through which programmers can utilize the shared memory between the cores. Such mechanism should provide a programming interface to support user-level applications. It should also verifiably enhance the performance and programming efficiency of the machine.

In this paper, we present an efficient mechanism for shared-memory communication between HPC applications executing within virtual machine (VM) instances that are *co-located* on the same hardware platform. It also provides this mechanism through a familiar programming interface that can be utilized to can be utilized to control the degree of isolation between virtual machine to enable low-latency shared-memory communication This system, dubbed *Vshmem*, implements low latency Sys V [30] IPC-style shared-memory, synchronous and asynchronous communication mechanisms allows the programmer to selectively share memory regions between user-space processes residing in collocated virtual machines. We also verified the performance of our system using various HPC computational kernels and applications. Our results reveals that HPC can leverage the power of virtualization as technology trends drive multi-core architectures and heterogeneity forward.

The rest of this paper is organized as follows. Section II presents the motivation and background for our work. Section III describes the design and implementation of Vshmem, as well as details the xen-port we implemented for the system. Section V displays the performance evaluation of the Vshmem, using various scientific dwarfs and applications. We discuss the performance results and potential impact of our system in section VI. We compare our work to other research in section **??**. Finally, we conclude our paper and present our future research directions in section VII .

II. Motivation and Background

The next generation HPC systems are adapting unprecedented hardware complexity in the form of increasing number of cores per processor in order to sustain the performance growth and to maintain the increase in the *MFLOPS* offered to the computationally-intensive HPC applications. However, the quest for an efficient software stack for such systems is a challenge facing the HPC community. Specifically, the "monolithic-kernel" traditional software stack approach presents serious impediments for next generation HPC systems as a result of its increasing memory foot-print and system noise and decreasing cache efficiency [2], [3], [4]. On the other hand, the virtualized software stack presents an attractive approach in addressing this complexity in the next generation HPC infrastructure. With virtualization, OS-specialization and customization can address those impediments and significantly reduce the OS-noise in the system.

To clarify, figure 2 demonstrates one example of the power of virtualization in leveraging the potential of multi-core systems through OS-specilization and customization. In this figure, each OS kernel can specialized for a specific core type or workload such that it extract all the computational power of the core without adding unnecessarily OS-overhead on other cores and applications. The software stack of this model is composed of a slim hypervisor deployed to manage the entire hardware and multiplex the resources between the different virtual machines. Each VM deploy an OS kernel which can be specialized, optimized and/or pinned to one or more type of cores such as computational cores and IO enabled cores. One recent virtualization technique that support this software stack without introducing significant overhead is paravirtualization. In paravirtualization, the interface exported by the hardware through the hypervisor is simplified in a way that eliminates hardware features that are difficult to virtualize. Examples of such features are *sensitive* instructions that must be intercepted and interpreted by the virtualization layer, introducing significant overhead. There are a small number of these instructions that the guest OS uses that must be replaced to enable execution of the OS over the VMM. Although the guest OS kernel has to be modified, no application code need to be changed to execute using a paravirtualizing system such as Xen. Although this software stack has a superb potential for extracting the computational power per core, the exchange of data between processes experience unnecessary communication latency as a result of perfect isolation between VMs. Specifically, the communication between user-space processes running in different OS kernels endure significantly higher latency than the processes executing within the same OS kernel.

Early on, the operating systems (OS) community recognized shared-memory Inter-Process Communications (IPC) capability as an essential OS service. UNIX, for example originally supported a number of process communication constructs including lock files, signals and pipes [30]. Today's Linux and UNIX-like systems support more communications constructs between processes. Arguably, the System V shared memory interface is one of the most popular IPC constructs among UNIX-programmers The System V shared memory interface provides abstractions that enable memory sharing of well-defined regions of a process address space, as well as synchronous and asynchronous communication between the processes in the form of semaphores and message queues. Therefore, user-space processes can establish communication using shared memory if they are executing within the same OS-kernel. Another IPC facility is sockets which allows user-space processes to establish a communication channel between themselves to facilitate data exchange, but which is usually used in a networked setting, to allow processes belonging to different hosts on the network to communicate.

Although IPC socket interface can be used for commu-

nication between user-space processes in collocated virtual machines, data exchange over the network protocols adds unnecessary communication latency. This is caused by IPC socket interface, which lacks the capability of providing direct (i.e. zero copy) shared memory between processes as well as the overhead introduced by the IP stack. In addition, it requires several cross address-spaces copies (i.e. user-space to kernel-space copy and kernel-space to user-space copy) that adds to the communication latency. In essence, this unnecessary latency is a consequence of the *perfect* memory isolation between virtual machines, which necessitates the use of communication protocols to transfer data between machines using standard network protocols.

To exemplify, Xen is an example of a hypervisor that is widely deployed. It enables user-space processes to communicate across VMs using TCP/IP sockets. Xen implements a split device driver architecture for the network subsystem. For that, a *netback* is deployed in the privileged VM, commonly referred to as Dom0 and a *netfront* is deployed in the unprivileged VMs, and they both interacts using high-level network device abstractions. When a user process needs to exchange data with another user process in a collocated VM, it can use socket network interface. The data to be exchanged is then copied to the kernel-space via *copy_from_user*, and uses the networking subsystem to encapsulate them in IP packets. Once the IP packets are ready, the DomU kernel a ring buffer that transfers the packets to Dom0. The latter domain which acts as the software bridge in this scenario. Upon the reception of the packets, Dom0 copies the packets to its own address space in order to process them and determine the receiver domain, and then adds them to the ring buffer of that receiver domain. Once the packets are transfered to the receiver domain, they gets copied to its own address-space which then determine the receiver process. The packets are then copied via *copy_to_user* to the user address space of the receiver process [20], [31], [32]. This communication pattern endures significant performance penalty since all the data exchange from the sender process to the receiver process is redirected through Dom0, and involves several copies across the address-spaces. The System V shared memory interface, on the other hand presents an efficient, low latency communication between the processes executing in the same user space through **direct zero-copy memory sharing** between them. However, it lacks the capability of enabling processes across VMs to communicate. Several projects, such as Xenloop [31], XenSocket [33], Xway [34] and MMNet [35] have looked at optimizing the socket communication path between user-space processes in collocated VMs by supporting them with shared memory. However, none of these systems can achieve comparable performance to low SYS V IPC latency of two communicating processes inside the same address space. This is a result of the IP stack overhead in some of these projects and the number of memory copies in the other projects.

Figure 3 illustrates this comparison between System V IPC via shared memory in the "monolithic-kernel" traditional software stack (on the left) versus the IPC socket interface in

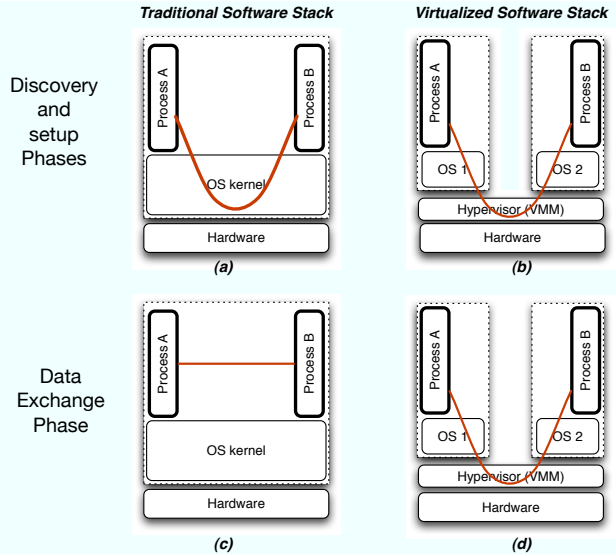

Fig. 3. This figure portrays the difference in the communication mechanisms between user-space processes collocated in the same OS, versus on different OS and using SYS V IPC versus socket programming respectively where (a) represents the discovery and setup phases in SYS V IPC in the traditional stack, (b) shows the same phases in the virtualized stack using socket programming, (c) represents the direct data-exchange using SYS V IPC in traditional stack while (d) shows the data-exchange in the virtualized stack using socket programming.

the virtualized software stack (on the right) through the three phases of inter-process communication: the discovery phase, the setup phase (both shown in the upper row) and the data exchange phase (shown in the bottom row). Note that, for simplicity, we consider Dom0 in this diagram as part of the hypervisor layer. In the traditional software stack, the shared memory discovery and setup for IPC is done via the operating system which is the common layer between the two user-space processes: the sender process and the receiver process. During the discovery and setup phases (i.e. subfigure (a)), the OS kernel registers the shared memory to its IPC facilities and makes it available for other processes to connect to by assigning it a key and shmem identifying number (aka *shmkey* and *shmid*). Once the receiver process identifies this shmem region via the shmkey and request to have it attached via a system call, the OS kernel maps it to the process's address-space. Once mapped, low-latency data exchange between the two user-space processes is facilitated as demonstrated in the lower left cartoon in figure 3 through direct zero-copy shared memory communication.

On the other hand, the virtualized software stack experiences much higher latency in inter-process communication between user-space processes in collocated virtual machines. The cartoon at the upper right corner of figure 3 demonstrates the discovery and the setup phases in the alternative software stack. The discovery mechanism in this scenario is done via Dom0 which acts as the software bridge between the different domains. As described earlier and demonstrated in the lower right cartoon (i.e. subfigure (d)), the actual exchange of data between the two processes uses socket programming which involves several copies from and to the user-space and

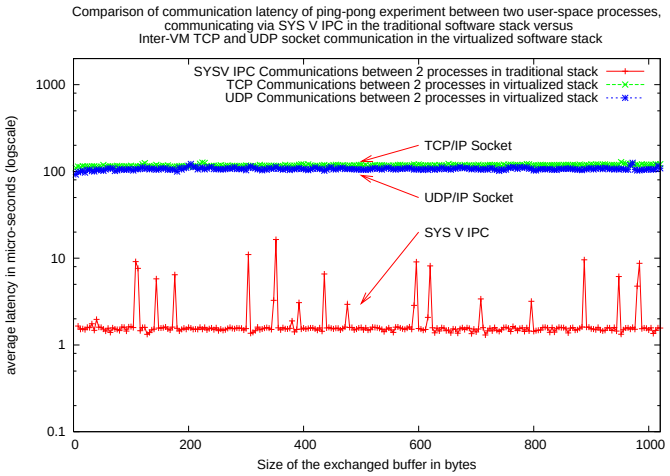

Fig. 4. This figure portrays the difference in the communication latency of user-space processes using SYS V IPC in the same operating system versus socket programming across virtual machines respectively.

the kernel-space, entails packet processing and the IP stack overhead. All these operations add a considerable overhead to the communication latency between the processes.

To characterize this overhead, we designed a simple ping-pong experiment of a buffer of data between two user-space processes, which send the buffer back and forth between them for 1000 times for each run. Figure 4 illustrates the communication latency in microseconds (on Y-axis) on log scale as a function of the exchanged buffer of data in bytes (on the X-axis). The figure compares between the latency of SYS V shared memory IPC in the traditional software stack versus Inter-VM communications using TCP/IP and UDP/IP sockets in the virtualized software stack. The numbers reported here are the averages of 10 runs. This experiment signifies the overhead introduced by socket programming in the virtualized software stack, and how it compares with SYS V IPC shared memory communications. The communication between virtual machines using socket programming introduces a much higher latency in communication in the order of 56 times more. For example, TCP and UDP average communication latency between user-space processes in virtualized environments are 117.66 and 106.94 microseconds respectively, while the average of the communication latency between user-space processes using SYS V IPC is 1.923 microseconds. This is a serious impediment for the new virtualized software stack since it will hinder the user-space from achieving near-peak performance of the machine. In turn, efficient low-latency inter-VM communication leveraging the shared memory in multi-core machines is indispensable for the virtualized software stack, and is more compelling now than before. The goal of this work is to address this limitation by providing an efficient low-latency mechanism between user-space processes in the virtualized software stack, which allows the programmer to *selectively* control the degree of isolation between the user-space processes in the virtualized software stack through a familiar and standard programming interface.

III. Vshmem Design and Functional Requirements

Towards the goal of supporting an efficient low-latency communication methodology in the virtualized software stack, we implemented *Vshmem*. *Vshmem* extends the System V IPC to enable zero-copy communication between user-space processes running in distinct VMs deployed on the same physical host. *Vshmem*, furthermore harnesses the familiarity of the UNIX programmers with System V IPC syntax and semantics for shared-memory, synchronous and asynchronous communication without disabling the isolation boundary between virtual machines. In this section, we describe the general design goals of Vshmem, its principle operations, the necessary extensions to the SYS V IPC.

Vshmem extends the existing SYS V IPC semantics [30] to enable processes executing in separate VMs to discover SYS V IPC constructs in other VMs collocated on the same physical node.

In order to enable user-space inter-VM direct shared memory, synchronous and asynchronous communication, Vshmem *must* support three principle operations in order to allow seemingly transparent SYS V IPC communication across virtual machines. The three main required operations are:

1) **Inter-VM IPC Constructs Discovery:** This operation refers to the conceptual procedure that equip any user-space process executing in a Vshmem-enabled virtual machine to *find* other Vshmem IPC constructs through the Vshmem system in collocated Vshmem-enabled virtual machines.
2) **Inter-VM IPC Constructs Setup:** This operation refers to the conceptual procedure that equip any user-space process executing in a Vshmem-enabled virtual machine to *share* some or all of its IPC constructs with some or all of the user-space processes executing in collocated Vshmem-enabled virtual machines. This abstraction also refers to the conceptual procedure of equipping any user-space process executing in a Vshmem-enabled virtual machine to *map* some of the existing Vshmem IPC constructs through the Vshmem system in collocated Vshmem-enabled virtual machines to its own address space.
3) **Inter-VM IPC Constructs Tear-down:** This operation refers to the conceptual procedure that equip any user-space process executing in a Vshmem-enabled virtual machine to *unmap* previously-mapped Vshmem IPC constructs from its own address space. Furthermore, it enables any user-space process executing in a Vshmem-enabled virtual machine to *un-share* IPC-constructs with other user-space processes in collocated Vshmem-enabled virtual machines, which it previously had shared.

These three operations demonstrate the necessary functional requirements for the Vshmem system to support inter-VM IPC constructs. In this respect, the implementation of these operations is the basic spine of the *Vshmem* system. Furthermore, their implementation constitutes the main mechanisms for

Vshmem port to any hypervisor or virtualization technology, as we exemplify later in this section. By providing a different implementation to each of these operations, *Vshmem* can be simply ported to various hypervisors and virtual machine monitors. For that reason, we ensured in our current *Vshmem* implementation to separate the functional requirements from their implementations in order to facilitate future development of *Vshmem* ports to other virtualization technologies. Furthermore, by separating the *Vshmem* requirements from their implementations, SYS V IPC Shared memory applications can be portably deployed across different traditional and Vshmem-enabled virtualized systems with minimal code modifications, since the API remains unchanged.

In addition to the required principle operations, a number of extensions to the current SYS V IPC are necessary to support inter-VM communications. We designed these extensions to the semantics of SYS V IPC with the goal of allowing discovery, setup and tear-down of shared-memory communication channels between user-space processes executing in collocated virtual machines [1]. The three different IPC extensions to support our efficient low-latency communication goal are:

1) **V-shmem**: This first extension implements a partial address-space sharing between user space processes in order to facilitate zero-copy communication between them. Having the communicating processes executing on the same physical node, *V-shmem* allows one process to extent its virtual memory and permit other processes in the same or different VMs to map this range of memory addresses to their own address space. The interfaces used for these procedures are the familiar SYS V *shmget()*, and *shmat()* system calls. User-space processes identify shared memory offered by other process through an IPC key, which is passed to the *shmget()* to obtain a *shmid*. Shmid is then used to map the memory region to the address space of the communicating process through the *shmat()* API. Thereafter, communication between the processes is a zero-copy communication with no latency. The *shmctl()* and *shmdt()* system call is used to perform control operations on the *V-Shmem*, including the removal of the shared memory constructs between processes.
2) **V-sem:** The second extension *Vshmem* presents is *V-sem*, a virtual semaphore implementation which enables synchronous communication between processes. *V-sem* also extends the SYS V semaphore implementation to work across virtual machines. It implements the same APIs, and syntax as SYS V. Also, it uses an IPC key to identify sets of semaphore and attach to them through the *semget()* system call. It further uses *semop()* and *semctl()* to set, get and test-and-set the semaphore between the processes.
3) **V-msgq:** The third extension is *V-msgq*, which implements a virtual message queue between different processes residing in different virtual machines. We also extend SYS V APIs to support the *V-msgq* abstraction. As a result, a *V-msgq* can be created and used by two processes using the *msgget(), msgctl(), msgrcv() and msgsnd()* system calls. IPC keys are further used to support identifying message queues across virtual machines running on the same physical node. *V-msgq* presents the asynchronous communication methodology between process in *Vshmem*. Reasonably, *V-msgq* was implemented using *V-shmem* and *V-sem*, and support the same syntax and semantics of the SYS V message queues.

In sum, *Vshmem* is intended to transparently broaden the SYS V IPC calls to extend the IPC capability across virtual machines on the same physical host. As described above, it extends the SYS V IPC shared memory API *shmget(), shmat(), shmdt(), shmctl(),* synchronous communication via semaphore API *semget(), semctl(), semop()* and asynchronous communication via message queues API *msgget(), msgctl(), msgrcv(), msgsnd()* to allow efficient inter-VM communication between user-space processes. We also designed several new kernel data structures to support Vshmem constructs. However, due to space limitations, we refrain from discussing these data structures. Interested readers in more detailed discussion of our implementation and the data structures should consult [**?**].

Several design considerations had shaped also our implementation decisions. The first consideration deals with the degree of the hypervisor modifications our implementation should include. In a virtualized environment, like in virtual clusters and cloud computing, modifications to the hypervisor are strongly discouraged. This is because of their prospective impact on the stability of the cluster system, the OS kernels and applications executing on this system. Therefore, the first design principle is to avoid any hypervisor modifications or extensions. In addition, for fear that we introduce performance or functional disturbance to the Linux kernel, we confined our Vshmem extensions to a Linux device driver, which is designed to be dynamically loadable and removable at runtime. Furthermore, in the hope that we support a portable implementation of Vshmem, we refrained from any hardware-specific codes in our implementation. In designing Vshmem, we gave special focus on the separation between the system abstractions and their mechanisms. In turn, this will facilitate the process of porting the system to other hypervisor and virtualization technologies by reusing several parts of the current Vshmem port. Specifically, we designed our implementation to an upper half which contain the Vshmem-generic codes, and a bottom-half which is hypervisor-technology dependent. This way, our system can be simply extended to a wide-range of IPC systems and to a wide-range of virtualization technologies.

IV. Implementation of Vshmem Xen port

We implemented the first *Vshmem* port for the Xen hypervisor [20]. Xen is a popular open-source hypervisor developed originally for the X86 architectures and implements paravirtualization to run virtual machines without requiring hardware

[1]We also commonly refer to these three Vshmem extensions: V-shmem, V-sem and V-msgq as the Vshmem constructs.

virtualization support. Recently, Xen has been ported to many OS kernels, and hardware architectures. Accordingly, we chose to first port the *Vshmem* IPC system to Xen because if its wide-adoption, its performance characteristics which we empirically evaluated in [28], [21], [22], in addition to its recent merge to the Linux kernel main-tree. In this section, we detail the implementation specifies of our Vshmem system.

Given the implementation considerations and principles we outlined in previous section, we implemented the Vshmem-Xen system as a Linux device driver. We deployed the concept of Xen split device driver in our implementation of Vshmem, where the driver is divided into two ends: front-end and back-end. Furthermore, each of the two ends is divided into two further parts: an upper half and a bottom half. Figure 5 overview the architecture of our Vshmem-Xen split device driver.

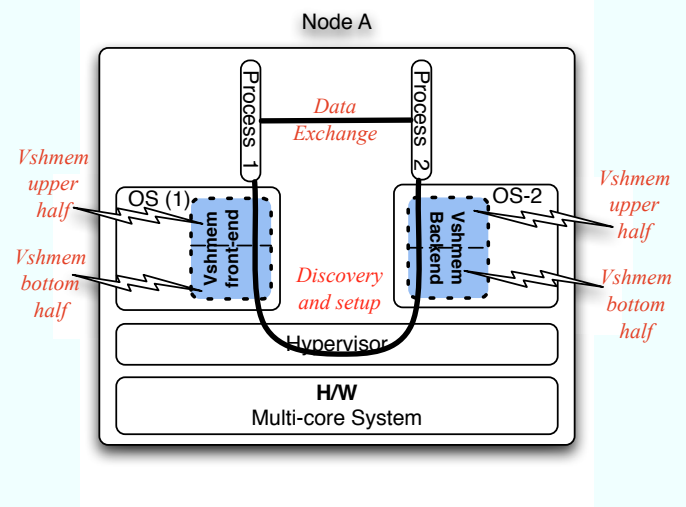

Fig. 5. The figure demonstrates the architecture of the Vshmem-Xen split device driver, with its two halves: the upper and bottom halves as well as its front-end and its back-end.

The Vshmem-Xen split device driver, as shown, consists of the front and back device ends. The back-end is the device-end which is responsible for creating and offering the Vshmem SYS V IPC constructs. On the other hand, the front-end is the device connecting to an existing SYS V IPC constructs. This distinction is necessarily since the procedure of creating and registering the new SYS V IPC construct to Vshmem system is different and requires more complicated procedures than the process of discovering and connecting to an existing construct. Normally –although not necessarily–, the device back-end is loaded to the Dom0 kernel, since it usually has more hypervisor-privileges which would facilitate efficient creation, registration and destruction of Vshmem SYS V IPC constructs. On the other hand, the front-end is loaded in the less-privileged virtual machines: DomUs. This is one use-scenario for the Vshmem device driver that supports efficiency and performance effectiveness. However, the virtual machine loading the front-end can also create and register SYS V IPC constructs. This is supported by having the front-end forward Vshmem constructs creations and registration calls to the back-end, which creates them on its behalf. This way, we allow both the front-end and the back-end of the Vshmem device driver to create, register and destruct Vshmem SYS V IPC constructs. It is important to mention that only the owner/creater of a Vshmem SYS V IPC constructs can register, set the ownership, set the access flags, and destructs its Vshmem constructs.

We also divided the implementation of each end of the device driver to an upper-half and a bottom half. The upper half of each device driver implements Vshmem extensions to the syntax and semantics of the SYS V IPC, and is hypervisor-technology independent. It implements the mechanisms of the shared memory, semaphore and message queue constructs through the Vshmem kernel data structure. It is this half of the device driver that interacts with the user-space processes. It also communicates with the bottom half through a set of function calls and well defined API.

The bottom half is the hypervisor-dependent implementation and uses the hypervisor-specific API in order to share memory pages with other virtual machines, to register Vshmem constructs or to destruct existing ones. By isolating hypervisor-specific implementation in the bottom-half of the device driver, we simplify the procedure of extending Vshmem to other virtualization technology since each hypervisor technology has its own memory management mechanisms and API. Should the open-source community decide to extend the Vshmem implementation to other hypervisor, they only need to re-implement the set of function calls of the bottom half.

In order to describe the bottom half device driver implementation, we need to highlight some Xen specific tools and subsystems. Xen originally provides a basic mechanism for memory sharing and data transfers between kernel-space processes in virtualized environments, which is called grant table (a.k.a. gnttab). The grant table mechanism mainly allows sharing of memory pages between **kernel spaces** of the different virtual machines. However, the granularity of memory sharing is typically coarse (e.g. 4096 bytes in x86 architecture) and is only allowed between kernel space processes. Despite the recent introduction of gntdev [36] to the Xen tree, which is a device to allow user-space processes to gain access to grant table pages, user-space processes continue to not be allowed to create, manage and remove grant tables pages. In addition, the discovery of the different grant table entries is not supported by the basic functionality of grant table or gntdev. Furthermore, gntdev does not provide a programming interface that the user-space processes can utilize. In this respect, the gntdev and gnttab are very limited subsystems for our purposes.

In order to discover and setup the different Vshmem IPC constructs between the domain, some form of basic configurations communication is needed between the Xen domains. Fortunately, Xen provides a key-value store – dubbed *XenStore* – which is shared between the VMs, and used to negotiate general device settings and configurations.

We use both Xen grant table and *Xenstore* to implement the Vshmem-Xen port. Vshmem is deployed on Xen in order to enable sharing and coordination between virtual machines. For

that, it deploys the Vshmem back-end in a privileged virtual machine, i.e. *Dom0*. Other virtual machines run in unprivileged mode, i.e. *DomU*s and deploys the Vshmem front-end. For the bottom half in both the front-end and the back-end, an implementation of the three Vshmem principle operations: Inter-VM IPC constructs discovery, Inter-VM IPC constructs setup and Inter-VM IPC constructs tear-down is provided.

The bottom-half uses *Xenstore* to announce its participation and availability for the *Vshmem* system, a state which we dub *VshmemReady*. Thereafter, whenever *shmget(), semget()* or *msgget()* are called with an *IPC_CREAT* flag set, the bottom half is responsible to request a new page to be shared by the requesting domain via the gnttab, a new *Vshmem* construct is created, and the new IPC key and its *Vshmem* construct gnttab reference(s) is/are announced through the *Xenstore* to the other *VshmemReady* domUs. When a new process executes a *shmget(), semget(),* or *msgget()* with the IPC_CREAT flag **un**set, the bottom-half first looks for the requested IPC-key in the XenStore, and connect to it if it exist. Otherwise, it returns an error. If the *Vshmem* IPC key is found, the bottom-half maps the shared memory region to the calling user-space process address space. In order not to break the isolation between the VMs, Dom0 has the sole capability to create new *Vshmem* constructs. However, *Vshmem* deploys a methodology through which a DomU may request from Dom0 to create a *Vshmem* construct on its behalf. This methodology allows user space processes in DomUs to use the *Vshmem* system without breaking the isolation barrier between the virtual machines.

Once the key is configured/found in the XenStore, the bottom-half configures the shared memory, semaphores and message queues can to be mapped to the address-space of the user-space process. However, a normal kernel-space to user-space mapping through *mmap()* does not work in this case, since the memory is owned by another domain. Every time the user-space process tries to access this memory region, a page fault is caused on this address although it owns all the permissions needed and the page is in memory. The solution to this problem is by injecting the shared page to the domain address space through a *pfn_remap_range*. The final Vshmem tear-down operation is done by removing the Vshmem keys from the XenStore, and un-sharing the memory between the domains through the grant table interface. All that procedures is transparent to the user-space process, and are hidden in the bottom-half.

In terms of performance overhead, the creation of any one *Vshmem* constructs costs the *Vshmem-backend* one hypercall to map the shared grant table page, and costs the *Vshmem-frontend* one hypercall to map the page into its address space. To avoid this overhead on the execution time of the user-space applications, a mechanism which anticipates the number of pages to be shared is needed such that the pages can be requested ahead of time and is part of our future work. The same number of hypercalls is needed for shared memory tear-down. Discovery does not require invocations to any hypercalls, but interacts with the Xenstore through its API. Once the shared memory is mapped to the address-space of the user processes, no further interaction is needed between the *Vshmem* driver and the hypervisor or Xen tools. Thereafter, a zero-copy methodology is in place which achieves the aimed-for efficiency and low latency data exchange between the user-space processes. In this regard, the overhead of interacting with the hypervisor, as reflected by the number of hyper-calls is kept to a minimum in our implementation. Furthermore, Vshmem constructs discovery, setup and tear down functionalities are not on the critical path of the data exchange and therefore, should not impact the performance and latency of the direct zero-memory copy between the user-space processes.

V. Vshmem Performance Evaluation

In this section, we describe the performance profile of the Vshmem system. In addition, we compare the performance of SysV/Vshmem to two other popular parallel programming systems: message-passing programming using MPI and shared-memory programming using OpenMP. For this comparison, we have selected four widely-used scientific dwarfs.

A. Methodology and Hardware Platform

In this performance evaluation, we used a dual-core, 2.8-GHz Intel Pentium D with an 800-MHz processor bus and 2 MB of L2 cache. The machine's memory system uses a 533-MHz bus with 1 GB of dual interleaved DDR2 SDRAM.

As we outlined before, we compared the performance of two software stacks: the traditional and the virtualized software stacks. The traditional software stack consists of a $Linux$ OS-kernel version 2.6.18 with SMP support. We considered this OS-kernel our base performance kernel. In addition, we used GCC version 4.1.2 and GNU make version 3.81 in compiling this OS-kernel and the different benchmarks and codes. We furthermore deployed fedora core-8 tools for file-system and software packages management. For the virtualized software stack, we employed Xen 3.3.0 hypervisor and *xenified* $Linux$ OS-kernels version 2.6.18 for the virtual machines. The kernels for the virtual machines were configured with a single processor, i.e. non-SMP kernels. Both the hypervisor and the OS-kernels were compiled from scratch with GCC version 4.1.2 and GNU make version 3.8.

In order to compare the performance of the different parallel programming models, we installed MPICH 1.2.7 runtime system [37] with P4 channel interface and configured it to utilize openSSH version 4.7 and openSSL version 0.9.8b for establishing secure connections between the MPI threads. The OMP benchmark executables were compiled and linked using GCC version 4.1.2 with the *-fomp* flag which supports the OpenMP v3.0 API and executable directives [38].

All the results of the experiments and benchmarks were collected in run level *one*. A run-level in Unix is a mode of operation that employs a specific set of services. Run level *one* is an intermediate state which starts single-user mode and does not start the heavy $Linux$ daemons. Offering only minimal OS services, a $Linux$ kernel operating in run-level *one* is usually more efficient than higher run-levels. In order to be able run MPI benchmarks at this run-level, we configured

the OS kernel to start the networking and sshd daemon Run level *five*, on the other hand is a full mode operational kernel with multi-user support, a display manager and console logins. In this evaluation, we employ run-level *one* for all of our experiments since we intend to compare the raw performance of the individual programming models. Using this run-level, we avoid both overhead and performance variation that OS daemons might introduce into the performance results.

B. Benchmarks

For our empirical evaluation of Vshmem, we deployed the benchmarks summarized in table I. This set of programs can be categorized into three groups .

a) Communication-latency Micro-codes: The overarching objective of Vshmem is to provide a low-latency communication mechanism between user-space processes in collocated VMs in virtualized HPC environments. Therefore, the first set of evaluation codes we used was focused on assessing this communication latency in the virtual software stack. Specifically, the objective of this set of micro-benchmarks is to contrast the communication-latency characterization of the current Vshmem implementation to other IPC communication methodologies between user-processes in different collocated virtual machines. Two micro-benchmarks were crafted in this subcategory to measure the latency of process-to-process communication across different mechanisms.

Our intent is to measure the overhead of socket connection management in comparison to the shared-memory mechanisms for communicating short chunks of data. The first benchmark, denoted *Sync*, passes control back and forth between two threads of execution for 100,000 turns by modifying data in a memory region shared by the threads. The benchmark was written in C relying neither on existing synchronization libraries nor on hardware support for synchronization. Instead, Sync uses classic P/V semaphores to impose the pattern of alternating accesses to the shared region, which is modified at each turn, as shown in figure 6. The benchmark uses busy waiting to minimize context switches.

The second micro-benchmark, denoted ***Socket***, uses a network socket to pass back and forth a chunk of data between two processes for *100,000* iterations. Socket is configurable to use UDP or TCP network protocols for the connection.

```
sem[0] = sem_init(0);
sem[1] = sem_init(1);
for (i = 0; i < iter; i++) {
       sem_P(sem[my_id]);
       memset (region, my_id, size);
       sem_V(sem[!my_id]);
}
```

Fig. 6. Simplified pseudo-code for *Sync* micro-benchmark. The same code runs in two threads, with the value of `my_id` being the only difference (it is 0 in one thread and 1 in the other). Variable `turn` and the semaphore structures are in shared memory.

b) Scientific Dwarfs: This group of benchmarks represents algorithmic methods and communication-computation patterns that are commonly exhibited in high performance computing applications as well as other areas such as embedded systems, database systems, machine learning and graphics. To contrast the performance of Vshmem to the performance of popular parallel programming models, we have implemented several computational codes and methods in MPI and OpenMP in addition to SYSV/ Vshmem.

We chose four different dwarfs to use in the evaluation of Vshmem and we selected a simple representative problem for each dwarf and studied its communication/computational pattern and data memory-layout as well as its parallel implementation considerations.

1) **Dense Linear Algebra***:* This dwarf represents the classical dense matrix and vector operations which is commonly occurring in many applications in high performance computing. Vector-vector multiplication is normally referred to as BLAS level 1 while matrix-vector and matrix-matrix multiplications are refereed to as BLAS level 2 and BLAS level 3 respectively. Linpack benchmark [39], for example is one application of this dwarf which is customarily used to rank the different HPC infrastructures for the semi-annual top 500 list [1].

We have adapted a simple MPI implementation of the matrix-matrix multiplication problem which is available publicly [40]. We modified this basic code to implement BLAS level 1 and 2. In this code, the work is distributed among the different MPI threads by rows.As a result, the memory of the applications in this dwarf is normally accessed in strides of rows. In addition, we substantially modified this code to convert it to the other two programming models: OMP and SysV/ Vshmem. Although we kept the same work distribution among the threads, we replaced the MPI calls and data structures with OMP *'for-loop'* directives to obtain the OMP version. We further placed matrix A, B and C in OMP shared memory. The same work distribution and data layout was also adapted for the SysV/ Vshmem version of the code. All the BLAS codes repeat the multiplication for 1000 iterations and output the average execution time for different matrices' sizes.

2) **Structured Grids***:* The structured girds dwarf represents a group of scientific problems which consist of data points laid on n-dimensions grids. A sequence of time-steps, during which each data point is updated using values of neighboring points is needed to solve these problems. Parallel implementations of this dwarf are normally realized by dividing the problem into subgrids. Each thread keeps a local copy of one subgrid and updates the original grid at the end of every step. Therefore, there is a significant data-exchange phase between the boundary points of the neighboring subgrids, which occurs at the end of every time step. This pattern of communication is a common pattern in fluid dynamics applications, finite elements methods, adaptive Mesh Refinement (AMR) applications, Partial differential equations (PDE) solvers and weather modeling.

We used one simplified example of the structured grids

Category	Type	Code	Code Description
Micro-codes	Communication latency	Sync	ping-pong of a buffer via Vshmem memory.
		Socket	ping-pong of a buffer via TCP & UDP sockets.
Scientific Dwarfs	Dense Linear Algebra	BLAS 1	Vector-Vector multiplication operations.
		BLAS 2	Matrix-Vector multiplication operations.
		BLAS 3	Matrix-Matrix multiplication operations.
	Structured Grids	Laplace Solver	A Laplace solver using Jacobi method.
	MapReduce	Parallel π Calculation	Calculates π using *Simpson's Integration Rule*.
	N-Body Methods	Molecular Dynamics	MD simulations using *Verlet scheme*.

TABLE I
AN OVERVIEW OF THE CODES AND BENCHMARKS USED IN EVALUATING THE VSHMEM SYSTEM

dwarf, which is the *Jacobi* method for solving the *Laplace* Equation. The Laplace equation is a PDE system whose solution is important in many scientific computations such as heat conduction and fluid dynamics as well as areas like astronomy and electromagnetism. *Jacobi method* is a relaxation method generally used to find an approximation of the solution of a linear system which we deploy here to find the solution of the Laplace PDE. During each time-step, each point in the Laplace mesh data-structure is updated by the average value of the 4 neighboring points.

we adapted a publicly-available MPI implementation of this problem in 2 dimensions [41], and modified the original code to simulate an increasing number of mesh points on any given number of processors. Furthermore, we substantially altered the code to convert it to the two other programming models, i.e. OMP and SysV/Vshmem. The work distribution for the three programming models is the same; each thread is processing a subgrid and is communicating its own boundary points with the neighboring threads. The data layout is, however different between the three implementations. In MPI, the points are distributed equally among the MPI threads, and the boundary points are exchanged at each time step using *MPI_Send()* and *MPI_Recv()*. *MPI_Reduce()* is used to check for the convergence of the Jacobi method. In the OMP implementation, the original grid is placed in the OMP shared memory. Each OMP thread keeps a local copy of the grid and updates the original grid at the end of each time step. SysV/Vshmem deploys the same data layout as the OMP implementation.

3) **MapReduce***:* The MapReduce dwarf presents an embarrassingly parallel algorithmic model that involves a minimal amount of communication between the parallel threads. Basically, it represents a pattern where repeated independent tasks compute a certain function on a large data-set, and the final result of the parallel method is an aggregation of the local computations.

In addition to its wide-deployment in large-scale search applications, this dwarf has several applications in HPC [42]. In scientific computing, Grid computing applications can arguably be considered one variation of this dwarf [42]. Monte Carlo methods, perhaps one of the most widely-deployed computational methods in scientific applications, are considered by some HPC scientists [42] a special case of the MapReduce dwarf. Furthermore, a recent research project [43] has stressed the potential of MapReduce methods for e-science and identified several scientific applications of this important algorithmic model such as High Energy Physics (HEP) data analysis and K-means Clustering.

We chose a simple problem to mimic the computation-communication pattern of the MapReduce dwarf. For that, we utilized a MPI implementation of π-calculation using *Simpson's Discrete Integration Rule*. In this application, π is evaluated as the value of the integral of $4/(1 + x * x)$ between 0 and 1. Since the value of an integral is the area under the curve of the function, π is approximated by the summation of n rectangles occupying the area under the curve. This approximation scheme is known as Simpson's discrete integration rule. The local computation in this problem comprises of an evaluation of the function at point x for the different x-values along the intervals assigned to each thread and a summation of the partial area under the curve for the local intervals. Undoubtedly, the bigger the number of intervals, the more accurate the approximation is but the larger the computation will be. Once the local computation is completed, the approximated value of π is acquired by aggregating the different local summations from the different threads. To summarize, the map operation for this problem is achieved by dividing the area into smaller intervals among the different threads and the reduce operation is done through the summation of the local calculations.

We have adapted a publicly-available MPI code that implements this problem [44]. In this MPI version, the work distribution is achieved by allocating equal numbers of intervals to each MPI thread. The communication between the threads occurs during the map phase, when the master thread broadcast the total number of intervals to all the MPI thread via *MPI_Bcast()*, and during the reduce phase when the local calculated values are all summed through *MPI_Reduce()*. We ported this MPI implementation to OMP and SysV/Vshmem and the same work distribution scheme is followed in the two new ports. In OMP and SysV/Vshmem implementations, the local values of the π calculation are placed in shared memory. The OMP implementation utilizes *OMP reduce(+:local_pi)* directive to perform the summation of the π value. In the

SysV/Vshmem implementations, we developed a reduction function that allows the master thread to aggregate the local summations of partial π values to realize the approximated overall π value.

4) **N-Body Methods***:* The N-Body dwarf represents the classical algorithmic methods that rely on the interaction of many points, where every point depends on all other points to update its status. These methods normally take an O(N^2) computational complexity and appear in a wide-range of scientific applications in astrophysics, molecular dynamics and computer graphics. Traditionally, the workloads and the communication scheme of the applications in these areas change dramatically with time as a result of the dynamic nature of the problems [45].

As an application of the N-Body dwarf, we used a publicly available [46] simple molecular dynamics simulation that was developed in OMP.In this simulation, the commencing positions and velocities of N interacting particles are initialized to random values. The simulation, then calculates the interaction between the particles and computes their new positions, velocities and accelerations using the *verlet time integration scheme*. The Verlet Time Integration Scheme is a numerical method used to calculate the integral of Newton's laws of motion at a reduced error level. It employs two Taylor expansions – one forward and one backward in time– of the position vector.

We adapted this code in our evaluation and modified the OMP version to simulate a varying number of particles (*np*), a varying number of dimensions (*nd*) for any number of time steps (*num_steps*). Furthermore, we exported the code to SysV/Vshmem. The data layout for both versions of the code is similar where the arrays accessed by all the threads are allocated in the shared memory. The work distribution between the threads in OMP and SysV/Vshmem is also similar, where the points are divided equally between the threads.

C. Communication Latency Evaluation

Achieving a low communication latency between the user-space processes in collocated VMs is one of our primary goals. Therefore, we crafted two benchmarks, *Sync* and *Socket*, to evaluate this latency. These benchmarks measure the time for data exchange between the user-space process and does not include the setup or overhead associated with establishing connections.

Figure 7 depicts the results of running the two benchmarks using traditional and virtualized software stacks. In this figure, the x-axis represents the size of the buffer communicated in bytes while the y-axis represents the average latency of 100,000 iterations in micro-seconds. In this experiment, *Sync* uses SysV shared memory to send the data buffer back and forth using zero-copy between the two user-space processes running in the same OS kernel for the SysV case.

The latency for sync using SysV is in the range of 1-2 microseconds, with a small number of outliers. In case of

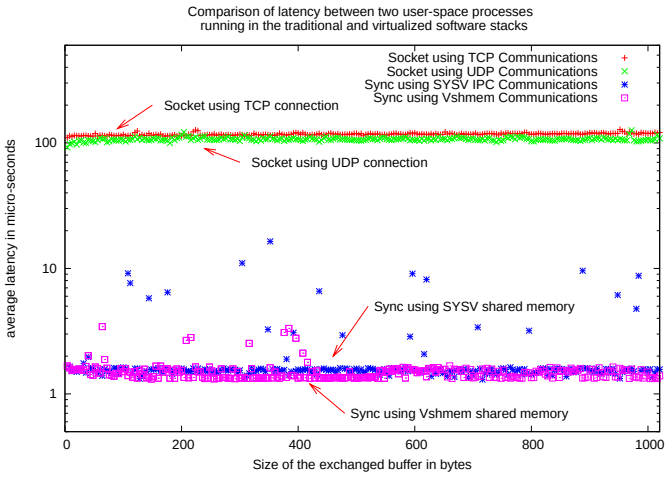

Fig. 7. This figure depicts the communication latency in the traditional and the virtualized software stacks. In this figure, the *Sync* benchmark is using the SYSV shared memory in the traditional stack and the Vshmem is using Vshmem shared memory in the virtualized stack. The *Socket* benchmark is using TCP and UDP connections, both in the virtualized stack.

Vshmem, *Sync* exchanges a buffer of data between two user-space processes running in different virtual machines. The VMs are collocated on the same physical node, but each VM is pinned to a different core to minimize the impact of VM context switching. As Figure 7 illustrates, *Vshmem* has a comparable performance to SysV IPC as both methodologies use zero-memory copy. In addition, the Vshmem curve does not exhibit the same variability as SYS V. Specifically, the variance for SysV results is 2.7 with standard deviation of 1.66 while the variance for Vshmem measurements is 0.007 with standard deviation of 0.086. We believe that this is due to less OS noise in the virtualized software stack caused by the lack of SMP support and unnecessary context switches. Other research projects [3], [4] have also encountered the impact of SMP support on OS-noise and presented its characterization in their work.

Additionally, figure 7 demonstrates the outcome of the *Socket* benchmark, which was run in the virtualized software stack between two user-space processes residing in two distinct collocated VMs. The results show that the latency of TCP and UDP communication is higher than shared memory. Our analysis of the collected data also shows that the latency of TCP and UDP is more variable than that of the shared memory communication methods, although it does not show in this figure. Specifically, the variance of the TCP measurements is 6.28 with standard deviation of 2.5 while the variance for UDP measurements is 11.2 with standard deviation of 3.3. Furthermore, there was no network noise throughout the experiment that would cause the high variability evident in the TCP and UDP data points. We believe that this difference is due to context switching and the network stack overhead. From these results, we conclude that Vshmem across VMs is an order of magnitude faster than socket-based communication. Furthermore, it achieves more reliable and less variable latency

in communication. As a result, Vshmem achieves a low-latency communication between user-space processes in the virtualized software stack.

D. Performance Evaluation using the Scientific Dwarfs

1) Dense Linear Algebra: BLAS routines were the first codes we used in comparing the performance across the different programming models. Figure 8 shows the average execution times of the different matrix-vector multiplication operations for varying matrices' sizes. Due to space limitations, we present in this paper the performance results for one level of $BLAS$ operations. However, the interested readers should consult [**?**] for the complete set of results. For the two of the subfigures, the X-axis represents the matrix dimension while the y-axis represents the average execution time in μseconds. In order to be able to capture the variability in the execution times, we ran each experiment for ten times and plotted the average execution times of all the runs, as shown by the scattered graph. We also plotted the execution time of the ten runs, as portrayed by the solid line in the subfigures.

Given the above results, we observed that the SysV/Vshmem achieves the shortest execution timing for all the different $BLAS$ levels and for the different data-types (i.e, the single and double precisions). Furthermore, we observed that OMP achieves the second fastest execution time for almost all the different $BLAS$ levels and for the different data-types. MPI implementation exhibits the slowest execution times among the three programming models. We also observed that the gap in execution times between the three programming models diminishes as the BLAS level and/or the matrix size increases.

These results are due to the communication mechanisms deployed by each programming model as well as the computational workload of each BLAS problem size. MPI uses the P4 channel, which establishes the communication between the different MPI threads using regular UNIX sockets. This adds an unnecessarily overhead to the execution of the application, especially that there is no data exchanged over the network. However, the IP stack adds this performance overhead, which we characterized in section V-C. OMP model, however uses shared memory to exchange data between the OMP threads which improves the overall execution times of the OMP application in comparison with the MPI application. At the same time, the OMP model places an unnecessarily overhead on the execution due to its advanced mechanisms for thread creation, synchronization and destruction. As a result, the SysV/Vshmem model achieve faster execution than OMP since it does not suffer from this overhead. We also observed that the performance gap – evident by the spacing between the different curves representing the different programming models – diminishes as the $BLAS$ level increases or the matrix size grows. In fact, as the computational workload of the problem increases, the ratio of the communication to computation changes. As the $BLAS$ level or matrix size increases, the overall execution times become dominated by the computations. As a result of the tremendous increase in the computational load, the communication to computations ratio will decrease. In turn, the communication overhead becomes insignificant on the overall processing time as the BLAS level and the matrix size increases. As a result, the performance gap between the different programming models diminishes.

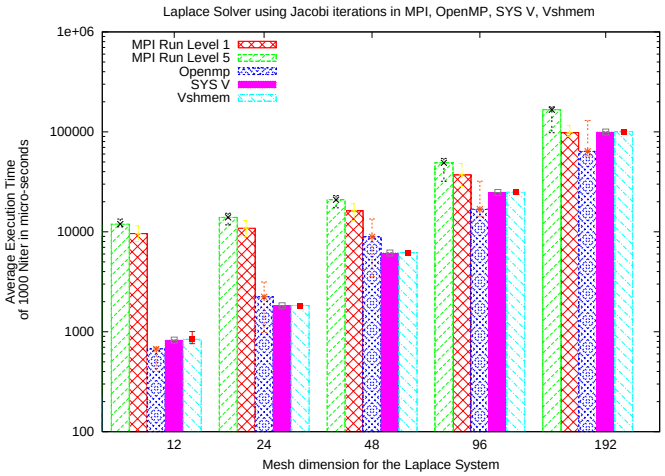

Fig. 9. This figure portrays the execution time of the Laplace solver using Jacobi iterations. The number of mesh points in the system are shown on the x axis while the y axis present the average execution time of 10 runs, where each run has 1000 iterations.

2) Structured Grids: In this category, we chose an implementation of a structured-grids problem for a Laplace solver using Jacobi methods in order to simulate the communication and computation patterns of this scientific dwarf. We configured the codes for an increasing number of mesh points and measured their execution times while calculating the average of 50 runs for each mesh size.

Figure 9 shows the average execution time of the Laplace solver for 50 runs using the different programming models. The number of mesh points in the system is shown on the x-axis while the y-axis presents the execution time in μseconds.

We first observed that the Laplace SysV and Vshmem execution times are not significantly different. We further noticed that the execution time gap between the different programming models diminishes as the mesh size increases, which is the same effect we detected in the BLAS results. This is again results from the increase in the number of mesh points which caused computation to dominate communication.

Surprisingly, the OMP Laplace code execution was the fastest of the three programming models, although it has the highest variability. We believe the reason for this difference is our implementation of the reduction operator. Early on, many studies have looked at optimizing the OMP reduction directive [47] as it was a main source of performance bottleneck in parallel programs for shared memory architectures. Some studies have shown this bottleneck may cost large-scale OMP parallel applications as much as half of their execution time [47]. These efforts have resulted in a highly-optimized reduction primitive that minimized the wait time of the other threads.

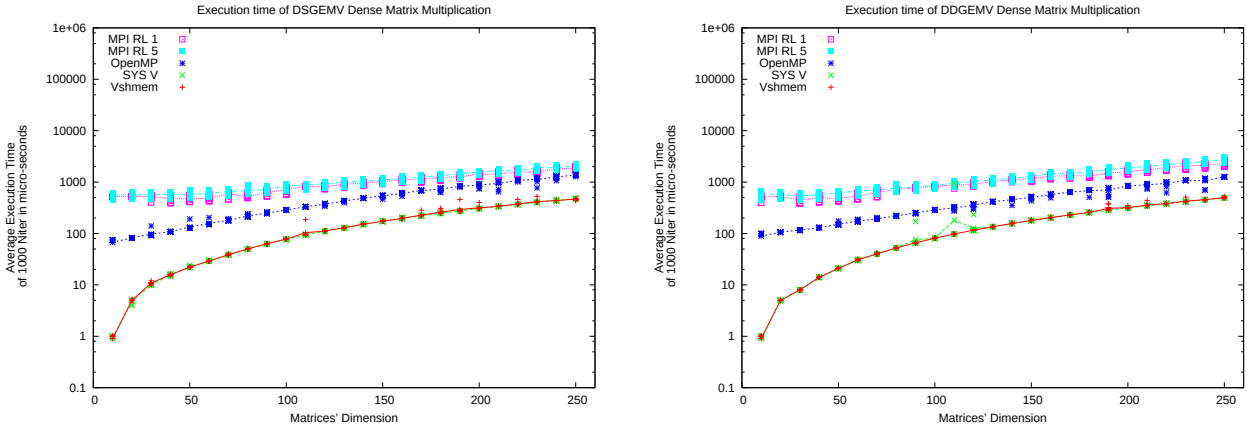

Fig. 8. The two figures depict the execution time for different programming models of the $matrix - vector$ multiplication codes for single (left subfigure) and double (right subfigure) dense matrix multiplication, respectively.

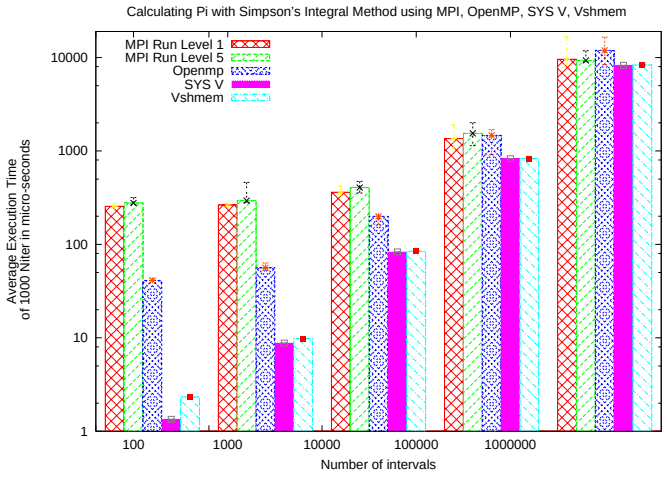

Fig. 10. This figure depicts the execution times for the calculation of π using the different programming models. The x-axis represents the number of intervals that was used for the *Simpson's discrete integration rule* while the y-axis represents the average execution time in μseconds.

3) MapReduce: The next scientific dwarf we used in our evaluation is the MapReduce dwarf. We deployed a very simple problem that reflects this dwarf's embarrassingly parallel algorithmic method, which is a π-calculation implementation using *Simpson's Discrete Integration Rule*. The MPI communication pattern for this problem consist of *a* broadcast operation at the beginning of execution to transmit the number of intervals to the different threads and *a* reduction operation at the end of execution to aggregate the local summation of the π values in order to calculate the total π. Each experiment in this code consists of 1000 iterations and the average execution time is reported.

Figure 10 presents the average execution times of 50 runs of the MapReduce experiment using the different programming models. The x-axis represents the number of intervals that was used in the π calculation while the y axis represents the average execution time in μseconds.

The results in this figure are similar to the performance profile of the previous dwarfs. Therefore, the SysV/Vshmem model exhibits the fastest execution times, followed by OMP, followed by MPI in run-level *one* followed by MPI in run level *five*. Despite the similarity between the performance ranking of the different models, the reasoning behind this ranking is different. For the MapReduce dwarf, there is minimal communication between the threads since this is an embarrassingly parallel method. However, we notice that the overhead of the runtime system of the different programming models is dominating the performance, especially at lower intervals count. In case of the MPI implementation, the MPICH runtime system is a middleware layer that slowed down the overall execution of this code. OMP runtime overhead was smaller, but it still imposed an unnecessary overhead on the threads execution. SysV/Vshmem, on the other hand, is a light-weight implementation that *selectively* shares memory regions between the processes without imposing a continuous overhead on their execution.

Furthermore, we noticed that Vshmem performance is slower than the SysV execution times for the 100 intervals, which is caused by the Vshmem operations' overhead described in section **??**. In addition, we also observed the same diminishing performance gap in the execution times between the programming models as the problem size increases. This is due to the fact that, as the problem size increases, it becomes more dominated by the computations, and the overhead of the runtime system becomes negligible relative to the overall execution time.

4) N-Body Methods: The last scientific dwarf we used in our evaluation is the N-body methods. In our own previous work [48], [49], we have studied an actual computational biology N-body problem and implemented its simulation in MPI. In this evaluation, we deploy an implementation of a molecular dynamics simulation. This simulation was written in OMP by John Burkardt in Florida State University [46] and we ported it to SysV/Vshmem. We modified the OpenMP and SysV/Vshmem codes to simulate a varying number of particles in a varying number of dimensions and time-steps.

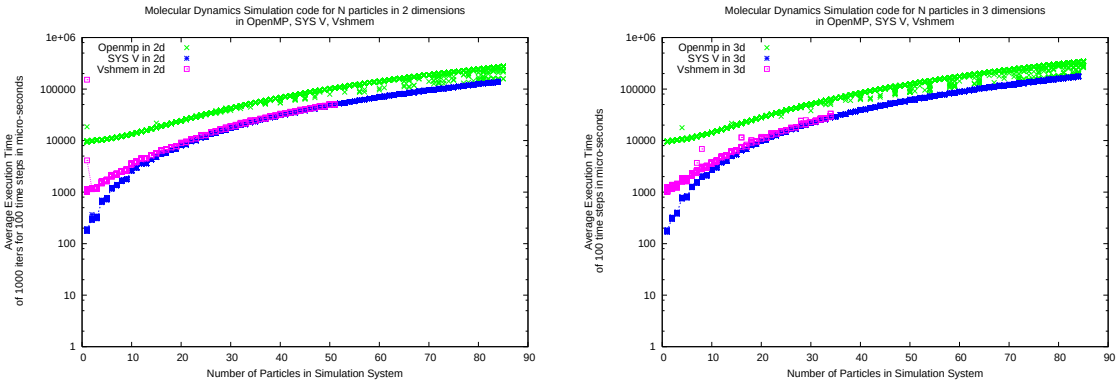

Fig. 11. These figures represent the execution times of the molecular dynamics code for an increasing number of particles in 2d (left subfigure) and 3d (right subfigure) for 100 time-steps simulations.

Our experiments simulated between 1 and 50 particles in two and three dimensions for 100 time-steps.

Figure 11 portray the execution times for the OMP and SysV/Vshmem programming models in two and three dimensions. The figures run in two dimensions. All the subfigures in the first row represent the simulations in two dimensions, while the second row represents the simulations in three dimensions. The second vertical dimension represent the number of time-steps in the simulation; the first column of subfigures represents simulations for 10 time-steps while the second column represents simulations for 100 time-steps. The x-axis in all subfigures represents the number of particles in the system while the y axis presents the execution times in μseconds. In order to detect the potential variability in the execution times, we plot the different times with the points in a scatter graph mode while the lines represent the average of the 50 execution times for each simulation.

The results show that OpenMP always exhibits the slowest execution time for the different dimensions, number of steps and number of particles in the simulation. This is a result that the OMP model places an unnecessarily overhead on the execution due to its advanced mechanisms for thread creation, synchronization and destruction. The results also show that for small computational loads, such that in subfigure (a) and (c) for simulating 35 or less number of particles and subfigures (b) and (d) for simulating 10 or less number of particles, Vshmem is slower than SysV implementation. This is a result of the high overhead of the Vshmem setup operations that we measured in section **??**. However, as the computations dominate the total execution time of the simulation, this Vshmem setup operations overhead is amortized and the performance difference between Vshmem and SysV becomes insignificant. We also observe the same effect of decreasing performance gap between the models. This performance gap evident by the spacing between the different curves representing the different programming models diminishes as the computations dominates the overall execution time.

VI. Discussion

Next-generation HPC infrastructures will deploy a massive number of cores per microprocessor in order to sustain their performance growth. Although this model has a huge performance potential, managing those systems and building their software stack can be very complex. The alternative approach we advocate in this paper is to utilize the isolation offered by the OS-virtualization technology to encapsulate and manage the complexity in those systems. In virtualization, a slim hypervisor is deployed to provide the multiplexing between the basic resources in the system. Several slim VMs would run on the hypervisor, each to manage a small count of cores as well as to manage their user-space processes. By isolating the management of the cores, the OS-noise can also be contained within each VM and the extraction of the performance potential of those system can be simplified. Arguably, this alternative model is promising to deliver the near-peak potential performance of those machines by customizing the VM kernels, and decreasing the OS noise, i.e. by avoiding posing the OS as the performance bottleneck.

Despite the attractive benefits of virtualization in the form of complexity isolation, it comes at the cost of complicating the programmability of those powerful HPC systems. Due to the lack of a simple and efficient communication mechanism between virtual machines (VMs) to allow user-space process to communicate and co-ordinate their progress, the programmability of this alternative software stack is a difficulty. Previous studies had utilized network-based communications between the virtual machines for process coordination and data exchange. UDP-socket and TCP-socket communications have been popular for communication between processes running on different physical machines. However, using socket-passed communications between virtual machines deployed on the same physical machine can be inefficient, since it will experience high latency and lower bandwidth. That is because, for every packet send, it will endure the unnecessary overhead of the IP stack and several memory copies across privilege levels. Therefore, a more efficient communication methodology between VMs collocated on the same physical

machine is needed.

In our experimentation, we used the Xen hypervisor as the building block for the alternative software stack approach. In order to provide an efficient communication mechanism between user-space processes in collocated VMs, we implemented Vshmem. Vshmem provides zero-copy communication (i.e, shared-memory, asynchronous and synchronous channels) between user-space processes running in different Xen domains (VMs), using the SYS V IPC-interface. SYS V IPC interface is a familiar and efficient communication and memory sharing mechanism between user-space processes running inside the same OS. Vshmem leverage the simplicity of SYS-V IPC interface to provide communication between user-space processes running in different collocated VMs by enabling SYSV IPC constructs like shared memory, message queues and semaphores to be deployed across distinct VMs.

Our results have exhibited that Vshmem can be very efficient and much lower in latency in communication overhead between the domains, in comparison with other inter-VM communication mechanisms. For a variety of scientific dwarfs, we have shown that Vshmem in the virtualized software stack can achieve comparable performance to Sys V IPC in the traditional software stack. We also compared Vshmem with other network-based communication mechanisms. The basic latency-inducing factor in network communication is the IP stack overhead and memory copies. In Vshmem, the memory is *directly* shared between the user-space processes, which can use the memory region at zero-copy communication. Xensocket [33], Xway [34] and Xenloop [31] are other Xen inter-VM methodologies that leverage grant-tables to support network communications between Xen VMs on the same physical machines. Vshmem is more efficient than those systems, since Vshmem does not involve any kernel-to-user space crossing neither copy data between the buffers in the data-path. Those methods mentioned above have two to 4 copies overhead, which in turns increase the data exchange latency between the communicating user-space processes. Furthermore, Vshmem gives the programmers a familiar programming interface that they can utilize to flexibly co-ordinate and program several processes to work together and accomplish the general program goal.

We also compared the Vshmem programming support to two familiar programming models: MPI and OpenMP based on their relative total execution time in solving specific common scientific dwarfs. However, the three models: MPI, OpenMP and SysV/Vshmem differ in other aspects as well. It is important before discussing these aspects to draw the distinction between a programming model, like MPI and OpenMP and programming OS-support, like Vshmem. A programming model is a complete set of API, library functions and/or runtime system that allow the programmer to implement a particular algorithmic method to solve a particular problem. On the other hand, a programming OS-support is a set of OS-extensions that allows the programmers and the various programming models to efficiently utilize the underlying infrastructures.

Vshmem provide an OS-support for multi-core microprocessor that enables shared-memory zero-copy communication as well as synchronous and asynchronous communication mechanisms between user-space processes in collocated VMs. However, it does not implemented higher-level programming constructs that are needed in expressing parallel algorithmic methods. For example, barrier and reduction operations, which are common parallel operations provided in most parallel programming models, are not provided by the Vshmem programming support. However, in order to implemented the different scientific dwarfs for our Vshmem performance evaluation, we used a simple implementation of those operations based on shared memory, which we include its implementation details in the appendices. The need to implement these operations portrays the lower-level abstraction that Vshmem presents to the programmers, in comparison with other higher level programming models. Although this might add a extra programming effort on the programmers, it gives them the flexibility to implement the most efficient form of the barrier and reduction to their problem. Higher-level programming languages can also have the flexibility to add their own implementation of these operations using the Vshmem shared memory, should they decided to add support of the Vshmem to their runtime systems.

Another distinction between the MPI and the OpenMP programming models and the SysV/Vshmem programming support is the style of parallelism. Vshmem parallel model deploys several independent processes, running in isolated virtual machines. In this model, the address space of each process is separate from the other processes. However, the programmer can *selectively* share memory regions between the different processes using the Vshmem facility. The OpenMP model has, however a different style of parallelism. In the OpenMP model, the main thread starts the other threads when it enters a parallel region and exits them when it exit this region. This model is commonly referred to as fork-and-join. The MPI model is similar to the OpenMP model, in that the main thread forks the other threads. However, the parallel region in MPI model cover the entire application execution time and MPI threads only exits at the end of the application execution.

The Vshmem parallelism model in implementing independent processes can offer a number of performance optimization and simplifies the development of certain categories of scientific codes. Specifically, Vshmem processes' affinity to cores can be specified through setting the affinity of the VM-to-core, which can be simply supported through virtualization capabilities. For example, in Xen, the programmer can use a simple API like "*xm pin Vcpu*" to set the affiliation of VMs to certain cores. Furthermore, the Vshmem shared memory can also be affiliated to physical memory through system calls like "*madvise()*". In addition, this flexibility in providing parallelism through independent processes can also facilitate the implementation of irregular scientific applications.

We believe our Vshmem system will have an impact on promoting mixed parallelism for next-generation high

performance computing infrastructures. In this mixed parallelism model, communication between processes deployed on collocated VMs can occur over Vshmem shared memory, which provides a lower-latency medium for data exchange. On the other hand, the inter-node communication can occur via message passing, which can provide a better scalability for the system. Mixed parallelism can have a very efficient programming model to extract the harness the performance power of the multi-core clusters, i.e. clusters of nodes where each node has one or more multi-core microprocessor. By using two types of parallelism, the programmers are given the flexibility to optimize their code to the specific architecture. However, it might also come at the cost of extra programming effort.

Vshmem can also support heterogeneity in modern multicore-based HPC systems. Hardware vendors have also adopted heterogeneous processor-design architectures, whether by incorporating varying processor architectures within the same chip, by diversifying the capabilities of the different cores, or their memory hierarchy, or merely their clock frequency. FPGAs, GPGPUs and hardware accelerators are becoming classic examples of the additional heterogeneity and complexity in the hardware in HPC infrastructures. One quintessential example of modern heterogeneous multi-cores is the cell processor, with its one general purpose processor (i.e. PPE) and eight special-purpose co-processors (i.e. SPP) [50]. Additionally, the High Performance Computing (HPC) community is harnessing the potential performance power of heterogeneous multi-core design in building its next generation computing infrastructure. Some examples are Los Alomos National Lab (LANL) Road-Runner machines [51], and Japan's TSUBAME that is composed of 655 Opteron Dual cores and 648 ClearSpeed accelerators [52]. Consequently, this HPC hardware complexity is being propagated up to the software stack and is posing several challenges to programming these HPC infrastructures and extracting their performance.

For that, the virtualized software stack can have a significant impact in simplifying the management and programmability of these heterogeneous multi-core processors. As each OS is encapsulated in its own VM, customizing the OS for each core-type become an attractive method to optimize the performance of the entire system. Furthermore, this model can also facilitate the affiliating memory regions to particular physical memory, and affiliating particular processes to specific types of cores, which in turn can present performance benefits for the applications.

There are several future potential directions for Vshmem. As Vshmem enables zero-copy memory communication between user-space processes, it has the potential of optimizing the performance of several other systems to extract the performance from the complex multi-core HPC systems. One prospective future direction to Vshmem is to add its support to modern parallel programming models. For example, by adding a new Vshmem channel to the MPI parallel programming model, MPI can leverage the power of the virtualized software stack while hiding the programming complexity of shared memory under the widely-used MPI interface. PGAS languages are another category of parallel programming models that can also leverage the Vshmem programming support. By adding Vshmem support to PGAS languages, PGAS programming can also leverage the virtualized software stack while continuing to support the distributed memory programming model. It also will extend the capability of the PGAS languages by allowing them to affiliate the PGAS processes with particular cores and affiliate shared-memory regions with particular physical memory addresses, which are currently not supported in PGAS languages.

Another prospective direction for Vshmem is to add new ports for other hypervisor and virtualization technologies, such as hardware-assisted virtualization technologies by Intel and AMD. Many recent advances have improved the performance ramifications of this model, bringing it close to the native performance. Several hypervisors and VMMs harness the power of the hardware-assisted virtualization, such as KVM [53] and VMware [54]. Therefore, one prospective extension to the current Vshmem implementation is to support these hypervisors. Vshmem was designed to simplify its porting to other hypervisor, by confining the hypervisor-specific implementation to the bottom half of the driver. By implementing the discovery, setup and tear-down functions in the bottom-half of the Vshmem driver for the new hypervisor, Vshmem port will simply work for other virtualization techniques.

The last prospective direction for this work is in the OS customization and specialization area. The OS is a complex software layer that is designed to provide a wide-range of services for diverse set of applications. However, OS-customization can be deployed to slim down the operating system and specialize it for a particular application as well as a particular core-type. This model has the potential of delivering the performance power of the HPC systems deploying multi-core microprocessor and minimizing the OS-noise that might interfere with the application execution.

VII. Conclusions

We investigated an novel approach to simplify the management and programmability of the multi-core processors and to extract their performance power in HPC. In particular, we have advocated virtualization as an alternative approach to the traditional over-featured OS kernel approach; the later characterized by its huge memory foot-print, low cache efficiency and high OS-noise. In our alternative approach, a slim hypervisor and several light-weight OS kernels are deployed to manage exclusive subsets of cores and hardware devices. Despite the attractiveness of this software stack model as a result of its enhanced scalability, reliability and low OS-noise, it poses a significant limitation associated with the Inter-VM communication which endures high-latency as a result of the perfect memory isolation between the VMs. We addressed these two limitations as follows.

To address this communication limitation, we implemented a OS-level support that allows a programmer to *selectively* relax memory isolation between the virtual machines. In

addition, our system's programming interface was provided as an extension to the widely-used SysV [30] IPC interface, whose familiarity significantly simplifies the programmability of our model. Through this system, we offer a shared memory, synchronous and asynchronous zero-copy communication channel between user-space processes running in distinct virtual machines running on the same chip. We also evaluated the efficiency of our system using micro-codes as well as common applications of widely-used scientific dwarfs.

In conclusion, our research outcomes have displayed that HPC can leverage the power of virtualization as technology trends drive heterogeneity and multicore forward. Our work potentially has outreaching and impactful benefits for the bigger HPC community, with the numerous advantages that virtualization offers to the HPC infrastutres.

Acknowledgements: The authors would like to acknowledge Chandra Krintz and Dmitrii Zagorodnov for the useful discussions on this topic and their inputs and feedback.

REFERENCES

[1] Top500, "Top500 supercomputing sites," *http://www.top500.org/*. [Online]. Available: http://www.top500.org/

[2] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[3] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, no. 1, pp. 3–16, March 2008. [Online]. Available: http://dx.doi.org/10.1007/s10586-007-0047-2

[4] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 303–312.

[5] R. Brightwell, R. Riesen, K. D. Underwood, T. Hudson, P. G. Bridges, and A. B. Maccabe, "A performance comparison of linux and a lightweight kernel," in *CLUSTER*. IEEE Computer Society, 2003, pp. 251–258.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania, "The multikernel: A new OS architecture for scalable multicore system," in *Symposium on Operating systems principles (SOSP)*, October 2009.

[7] 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, pp. 76–85, 2009.

[8] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive Fault Tolerance for HPC with Xen Virtualization," in *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 23–32.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.

[10] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2009, pp. 101–110.

[11] R. Nathuji and K. Schwan, "Virtualpower: coordinated power management in virtualized enterprise systems," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 265–278.

[12] E. V. Hensbergen, "P.r.o.s.e.: partitioned reliable operating system environment," *Operating Systems Review*, vol. 40, no. 2, pp. 12–15, 2006.

[13] M. A. Butrico, D. D. Silva, O. Krieger, M. Ostrowski, B. S. Rosenburg, D. Tsafrir, E. V. Hensbergen, R. W. Wisniewski, and J. Xenidis, "Specialized execution environments," *Operating Systems Review*, vol. 42, no. 1, pp. 106–107, 2008.

[14] S. Thibault and T. Deegan, "Improving performance by embedding hpc applications in lightweight xen domains," in *HPCVirt '08: Proceedings of the 2nd workshop on System-level virtualization for high performance computing*. New York, NY, USA: ACM, 2008, pp. 9–15.

[15] G. Back and D. S. Nikolopoulos, "Application-Specific Customization on Many-Core Platforms: The VT-ASOS Framework," in *Proceedings of the Second Workshop on Software and Tools for Multi-Core Systems*, March 2007.

[16] L. Youseff, R. Wolski, and C. Krintz, "Linux Kernel Specialization for Scientific Application Performance," Univ. of California, Santa Barbara, Tech. Rep. UCSB Technical Report 2005-29, Nov 2005.

[17] T. Anderson, "The case for application-specific operating systems," in *Workstation Operating Systems, 1992. Proceedings., Third Workshop on*, Apr 1992, pp. 92–94.

[18] T. Naughton, G. Vallee, and S. Scott, "Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure," in *First Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007)*, Mar 2007.

[19] A. Whitaker, M. Shaw, and S. Gribble, "Scale and performance in the Denali isolation kernel," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2002, "http://denali.cs.washington.edu/".

[20] P. Barham and B. Dragovic and K. Fraser and S. Hand and T. Harris and A. Ho and R. Neugebauer, "Virtual Machine Monitors: Xen and the Art of Virtualization," in *Symposium on Operating Systems Principles (SOSP)*, 2003.

[21] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Paravirtualization for HPC Systems," in *ISPA Workshops*, ser. Lecture Notes in Computer Science, G. Min, B. D. Martino, L. T. Yang, M. Guo, and G. Rünger, Eds., vol. 4331. Springer, 2006, pp. 474–486.

[22] ——, "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*, 2006.

[23] 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.

[24] A. Gavrilovska, S. Kumar, K. Schwan, H. Raj, V. Gupta, R. Nathuji, A. Ranadive, R. Niranjan, and P. Saraiya., "High-performance hypervisor architectures: Virtualization in hpc systems," in *Proceedings of 1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007*, 2007.

[25] A. Tikotekar, H. Ong, S. Alam, G. Vallée, T. Naughton, C. Engelmann, and S. L. Scott, "Performance comparison of two virtual machine scenarios using an hpc application: a case study using molecular dynamics simulations," in *HPCVirt '09: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. New York, NY, USA: ACM, 2009, pp. 33–40.

[26] A. Tikotekar, G. Vallée, T. Naughton, H. Ong, C. Engelmann, and S. L. Scott, "An analysis of hpc benchmarks in virtual machine environments," pp. 63–71, 2009.

[27] A. Tikotekar, G. Vallée, T. Naughton, H. Ong, C. Engelmann, S. L. Scott, and A. M. Filippi, "Effects of virtualization on a scientific application running a hyperspectral radiative transfer code on virtual machines," in *HPCVirt '08: Proceedings of the 2nd workshop on System-level virtualization for high performance computing*. New York, NY, USA: ACM, 2008, pp. 16–23.

[28] L. Youseff, K. Seymour, H. You, J. Dongarra, and R. Wolski, "The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software," in *HPDC*, M. Parashar, K. Schwan, J. B. Weissman, and D. Laforenza, Eds. ACM, 2008, pp. 141–152.

[29] 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*. [Online]. Available: http://dx.doi.org/10.1007/s10586-009-0080-4

[30] M. J. Bach, *The design of the UNIX operating system*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[31] J. Wang, K.-L. Wright, and K. Gopalan, "Xenloop: a transparent high performance inter-vm network loopback," in *HPDC '08: Proceedings*

of the 17th international symposium on High performance distributed computing. New York, NY, USA: ACM, 2008, pp. 109–118.

[32] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in xen," in *USENIX Annual Technical Conference*, May 2006, pp. 15–28.

[33] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin, "Xensocket: A high-throughput interdomain transport for vms," IBM Research Technical Report RC24247, Tech. Rep., 2007.

[34] T. X. Team, "Xway: Lightweight communication between domains in a single machine," 2007, http://sourceforge.net/project/platformdownload.php?group_id=191553.

[35] P. Radhakrishnan and K. Srinivasan, "Mmnet: An efficient inter-vm communication mechanism," june 2008.

[36] D. G. Murray, G. Milos, and S. Hand, "Improving xen security through disaggregation," in *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2008, pp. 151–160.

[37] W. D. Gropp and E. Lusk, *Installation Guide for* `mpich`, *a Portable Implementation of MPI*, Mathematics and Computer Science Division, Argonne National Laboratory, 1996, aNL-96/5.

[38] Openmp, "Openmp," http://openmp.org/wp/. [Online]. Available: http://openmp.org/wp/

[39] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 1–18, 2003.

[40] R. Leibensperger, B. Barney, and G. L. Gusciora, "MPI Matrix Multiply - C Version," http://www.hku.hk/cc/sp2/workshop/samples/mpi/C/mpi_mm.c.

[41] "Laplace Solver using Jacobi iterations," http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/jacobi/C/main.html.

[42] 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

[43] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 277–284.

[44] "Calculating π using Simpson Integration Rule," http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/pi/C/main.html.

[45] D. C. Rapaport, *The Art of Molecular Dynamics Simulation*. New York, NY, USA: Cambridge University Press, 1996.

[46] J. Burkardt, "An OpenMP Molecular Dynamics Simulation using verlet time integration scheme," http://people.sc.fsu.edu/~burkardt/c_src/md_open_mp/md_open_mp.html.

[47] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh, "Evaluating synchronization on shared address space multiprocessors: methodology and performance," in *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1999, pp. 23–34.

[48] L. Youseff, A. Barbaro, P. Trethewey, B. Birnir, and J. R. Gilbert, "Parallel modeling of fish interaction," in *CSE '08: Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 234–241.

[49] A. Barbaro, K. Taylor, P. F. Trethewey, L. Youseff, , and B. Birnir, "Discrete and continuous models of the dynamics of pelagic fish: application to the capelin," *the journal of Mathematics and Computers in Simulation (MATCOM-D-08-00022 journal)*, 2009.

[50] M. W. Riley, J. D. Warnock, and D. F. Wendel, "Cell broadband engine processor: Design and implementation," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 545–558, 2007.

[51] K. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the Petaflop Era: The Architecture and Performance of Roadrunner"," in *IEEE/ACM Supercomputing (SC08)*, November 2008.

[52] S. Matsuoka, "The tsubame cluster experience a year later, and onto petascale tsubame 2.0," in *PVM/MPI*, ser. Lecture Notes in Computer Science, F. Cappello, T. Hérault, and J. Dongarra, Eds., vol. 4757. Springer, 2007, pp. 8–9.

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

[54] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.