

Transparent System-level Migration of PGAS Applications using Xen on InfiniBand

D.P. Scarpazza[#], P. Mullaney^{*}, O. Villa[#], F. Petri[#], V. Tipparaju[#], D.M.L. Brown Jr.[#] and J. Nieplocha[#]

[#]*Pacific Northwest National Laboratory*

MS K7-90, 902 Battelle Boulevard, Richland WA 99352, USA

{daniele.scarpazza, fabrizio.petrini, vinod, david.brown, jarek.nieplocha}@pnl.gov

^{*}*Novell Corp.,*

404 Wyman Street, Waltham, MA 02451, USA

pmullaney@novell.com

Abstract—Checkpoint-restart is considered one of the most natural approaches to achieving fault-tolerance in a high-performance cluster. While experiences has focused attention on user-level solutions, the advent of efficient system-level virtualization software, such as Xen and VMWare, has opened the door to the possibility of efficient and scalable cluster-level virtualization.

In this paper we present an innovative approach to cluster fault tolerance by integrating the Xen virtualization with the latest generation of the InfiniBand network. A major contribution of this approach is the automatic identification of global recovery lines to freeze the status of the machine. Our focus is on the partitioned global address space (PGAS) programming models. PGAS models has been receiving an increasing amount of attention in the recent years. We have developed a global coordination mechanism and deployed it in the Aggregate Remote Memory Copy Interface (ARMCI) one-sided communication library that has been used as a run-time system for several PGAS languages and libraries. The experimental results show that it is possible to virtualize communication and computation with minimal overhead and to provide seamless migration capabilities.

I. INTRODUCTION

The ever-increasing demand for processing power has led to the widespread diffusion of computing clusters and super-computers. Parallelism permeates these computing platforms, ranging from multiple processing cores within a single processor [1] to complex systems with tens of thousands of processors such as BlueGene/L [2].

Together with technological constraints, such as power consumption, integration technology, and software challenges to achieve the maximum level of performance, usability has become a major issue. A large-scale supercomputer (and now even a medium-scale cluster) is composed of many processing and communicating components that can have thousands of threads of concurrent activities with a proportional number of outstanding messages at any given time. The large total component count of these parallel systems makes any assumption of complete reliability entirely unrealistic. Although the mean time between failure (MTBF) for the individual components (e.g., processors, disks, memories, power supplies, and networks) and the physical connections between them may be very high, the large number of components in the system will inevitably lead to frequent individual failures.

Unfortunately, the current state of practice for fault tolerance is such that the failure of a single component usually causes a significant fraction of the system, and any application using that part of the system, to fail. In fact, the components of the system are strongly coupled; for example, the failure of a fan is likely to lead to other failures due to overheating. And application state is not stored redundantly, so loss of any state is catastrophic.

Over the years, users and system administrators have developed practical heuristics to achieve fault tolerance. The most common solution is to perform application-specific, user-level checkpoints at regular intervals. This error-prone approach puts the burden on the programmer, who needs to identify the *live* data structures, the specific points in the applications where it is possible to take a checkpoint, and the data format and the frequency of the checkpoint. The likelihood of a failure is reduced if a large-scale machine is logically divided into independent segments or when the user is forced to launch jobs that use a small subset of the available resources. While this is a straight forward solution, it seriously limits the computational capability, the very reason of existence of parallel computers.

A. Software Approaches to Fault Tolerance

Due to the prohibitive cost of using hardware redundancy [3] which can multiply by an integer factor the cost of a machine, several research projects have considered software solutions to this problem. The initial work of Plank and others [4], [5], [6], [7] has been reevaluated in the last few years in the context of cluster computing.

Software solutions can be classified based on the level of abstraction at which the checkpoint is taken. At the highest level of abstraction, we can take advantage of the algorithmic properties of some scientific applications that converge to the correct result even in the presence of system failures [8]. The practicality of this method is obviously limited, because only some applications meet its requirements.

Another approach is based on modifying the application's source code to perform checkpoint/recovery [9]. The programmer directly inserts checkpoint/recovery points into the programs source code with the support of a library that implements checkpointing primitives. A refinement of this

approach is to use the compiler to automatically insert the checkpoint code in a way that is nearly transparent to the programmer [10].

The emphasis of several recent research efforts is on achieving user-transparent, automatic, and efficient checkpoint and restart [11] [12] [13]. This requires the solution of several problems, including the virtualization of computation and communication, the identification a global recovery line to take a coordinated snapshot of the system, and the implementation of migration algorithms. To gain wide acceptance in the user community these solutions need to have very low overhead, both in fault-free mode and during a recovery, and they must be highly scalable.

DejaVu [14], a joint academic and industrial effort, is a transparent user-level framework. DejaVu is able to virtualize the operating system (OS) interface, making it transparent to both applications and any communication middleware, by intercepting all the system calls made by either the application or any middleware libraries it is linked against. Consistency on the recovery line is achieved through a transparent online logging protocol which relaxes the requirements of a distributed snapshot and implements a reliable communication, with explicit acknowledgments sent at the library level.

User-level processor virtualization is also the basic concept behind Charm++ and Adaptive MPI [15], which supports multiple checkpoint-based fault-tolerance protocols and a message logging-protocol.

This checkpoint mechanism can also be implemented inside the operating system kernel [16]. There are many details of a process state that are only known to the kernel or are otherwise difficult to re-create, such as the status of open files and signal handling. The main advantage of implementing checkpoint mechanism in OS kernel is that it is totally user-transparent and requires no changes to any application code. The downside of this approach is the increased complexity of working at the kernel level, with rapidly changing and often undocumented kernel versions, and the demanding constraint of porting the checkpoint/restart mechanisms to multiple processor and network architectures.

The early experience of Transparent Incremental Checkpointing at Kernel-level (TICK) [17] proves that kernel-level checkpointing can be remarkably fast, with minimal overhead and very high responsiveness which are important features when multiple nodes in a cluster need to take a coordinated checkpoint.

An integrated solution to checkpoint/restart is presented in [18]. The proposed software infrastructure is based on LAM/MPI and BLCR (Berkeley Labs Checkpoint Restart). The emphasis of this work is on the development of a resource management software layer and the coordination between the job scheduler and various local daemons to achieve transparent migration.

Virtual machines, such as VMware and Xen [19], have recently gained popularity in the academic and industrial communities. A key component of a virtual machine (VM) is the virtual machine monitor (VMM), also called hypervisor

or host, which is implemented directly on top of the available hardware and provides virtualized hardware interfaces to the guest VMs. A virtual machine is an ideal building block to implement checkpoint/restart algorithms, and to virtualize the resources in a cluster. For example, VM can simplify cluster management, by allowing the installation of customized versions of the operating system with different levels of security, instrumentation, services, etc.

B. Contribution of this Paper

The experience of the last few years has clearly shown that achieving fault tolerance in a large-scale supercomputer is an elusive goal. Software-level solutions need to be completely user-transparent, minimally intrusive, and economically feasible from a software engineering point of view.

This paper provides a proof of concept on how some of these demanding design goals can be achieved. More specifically, these are the main contributions of this paper.

- 1) *Integration of Xen and InfiniBand.* As already discussed, the resources of a computational node can be virtualized at different levels, using libraries such as DejaVu or kernel modules such as TICK or BLCR. The use of a virtual machine, such as Xen, is a very attractive solution because it leverages the efforts of a large community of developers. We have enhanced Xen's kernel modules to fully support the user-level InfiniBand protocols and IP over IB with minimal overhead. We also provide a seamless mechanism to migrate a Xen image from one node to another, extending the work presented in [20], [21] and [22].
- 2) *Support for Partitioned Global Address Space (PGAS) programming models.* Programming models that provide the visibility of a single global address space across distributed memory are gaining popularity in the academic, scientific and industrial communities. While most of the existing work in checkpoint/restart has been focused on the MPI communication library, in this paper we support the ARMCI library-based PGAS programming model.
- 3) *Automatic Detection of a Global Recovery Line and Coordinated Migration.* PGA programming models greatly simplify the run-time software of a communication library –for example they do not require explicit tag matching to send or receive a message. In the paper we present a novel algorithm to issue a coordinated checkpoint, distribute the checkpoint request across the processing nodes and the VM hypervisors, induce a global recovery line and perform a live migration without any change to the user application.
- 4) *Experimental Evaluation.* Finally, the experimental evaluation shows that cluster virtualization and image migration can be achieved with minimal overhead.

We believe the the results presented in this paper can also be used in a wide class of applications that go beyond cluster fault tolerance. Virtualization is a very powerful means of simplifying cluster management; for example, to easily roll back different types of software installations. Another

possible application is to use migration in combination with a resource manager to increase cluster use with a more flexible scheduling algorithm. Finally, migration can be an effective tool to implement proactive fault tolerance, as shown in [23].

II. GLOBAL SOFTWARE DESIGN

This section provides an overview of our software architecture, with emphasis on the Xen hypervisor, the InfiniBand device driver and the communication libraries that we use in the experimental evaluation. We integrate several components, including new and enhanced InfiniBand kernel drivers and a global coordination mechanism, to take fully transparent checkpoints across a computing cluster. We consider homogeneous high-performance clusters, i.e. each machine has the same number and type of CPUs, the same amount of physical memory, and the same network adapters. The various components of the software stack are outlined in Figure 1.

A. Xen Hypervisor

Xen [19] allows the system administrator to create, on each physical machine, one or more non-privileged VMs (also called “domU” domains), plus one privileged virtual machine (called the “dom0” domain) that is intended to configure and invoke the hypervisor [24]. Xen provides the ability to pause, un-pause, checkpoint, and resume domU VMs. Additionally, Xen can migrate domU domains across physical machines: proactive fault tolerance strategies [25] exploit these features to move tasks from physical machines showing distress signals to spare machines, thus providing computation continuity with an acceptable overhead. Moreover, the cost of migration can be significantly reduced by using the live migration capability, which performs most of the VM memory image transfer while the VM is still active and fully operational, thus achieving down times as short as 60 ms [26].

By employing Xen it is possible to achieve process mobility by running the application tasks, together with all their required libraries and user-mode operating system components in a single domU virtual machine. When the tasks need to be migrated, the entire VM state is migrated with them. This approach exhibits a number of advantages with respect to more traditional approaches based on process-level migration [18]; for example, applications do not need to have explicit knowledge of synchronization points.

In this paper, each physical machine hosts exactly one dom0 and one domU domain. We use the privileged domain only for management purposes, while all the payload computation is carried out in the unprivileged para-virtualized domain. Each virtual machine is configured with the same number of CPUs.

Other choices are possible, e.g., a separate domU VM per CPU. This solution provides finer migration control, allowing each process to be checkpointed, restored and moved separately. But the finer control allowed by this solution is not useful in our scenario, where physical nodes are homogeneous. Rather, performance is hindered, because tasks running on the same physical machines will appear now in distinct VMs,

therefore being unable to use fast inter-process communication via shared memory.

Xen is gaining increasing popularity [27], [21] because it makes these features available with an acceptable performance overhead [28], [29]. Unlike full virtualization, Xen uses an approach called para-virtualization, where non-privileged domains run a modified operating system featuring *guest device drivers*. Guest drivers do not access the hardware directly, rather their requests are forwarded to the native device drivers running in the privileged dom0 domain, according to a *split driver* model.

B. InfiniBand Device Driver

Many high-performance interconnects capable of Remote Direct Memory Access (RDMA) semantics, like InfiniBand, Quadrics and Myrinet, offer an OS-bypass feature, which avoids many OS-induced latencies and multiple copies without sacrificing the isolation provided by the split driver model. Our design relies on the InfiniBand interconnect, which is increasingly gaining popularity in the High Performance Computing arena [30], [31]. In a traditional InfiniBand setup (without virtualization), the hardware adapter is capable of accessing directly the user-level communication buffers, without any OS-induced overhead (OS-bypass). When virtualization comes into play, all the software components can be arranged to bypass not only the guest OS but also the dom0 OS and the virtual machine manager itself. This technique, called VMM-bypass [20] allows VMs to achieve nearly the same raw performance as the original InfiniBand drivers running in a non-virtualized system. In an HPC context like the one we present in this work, where each physical machine hosts exactly one virtual machine, the advantages of the physical isolation of VMs are not very important, whereas the performance benefits of VMM bypass are crucial.

Our VMM bypass driver extends the original concept of OS-bypass [20] to allow individual VMs and applications on those VMs to bypass the OS layers as well as the hypervisor itself. The driver is implemented in two sections, a paravirtualized section for *slow path* control operations (q-pair creation for example) and a direct access section for *fast path* data operations (transmit/receive). Together, these sections provide the implementation of the verbs interface for guest VM access. This approach achieves optimal execution in a virtualized environment by allowing direct access to the Host Channel Adapter (HCA) for performance-critical data path operations.

Our work is based on a proof-of-concept implementation that was produced at IBM and Ohio State University. The original driver was extended and modularized so that it could support additional CPU architectures as well as additional InfiniBand adapters. In addition, a proxy layer was added to enable InfiniBand Subnet Management and Connection Management protocols to operate from guest VMs. InfiniBand HCA ports can become members of IB multicast groups. Because membership is at the port level and the port is shared between VMs in a virtualized environment, a multicast group reference counting mechanism was added so that each VM

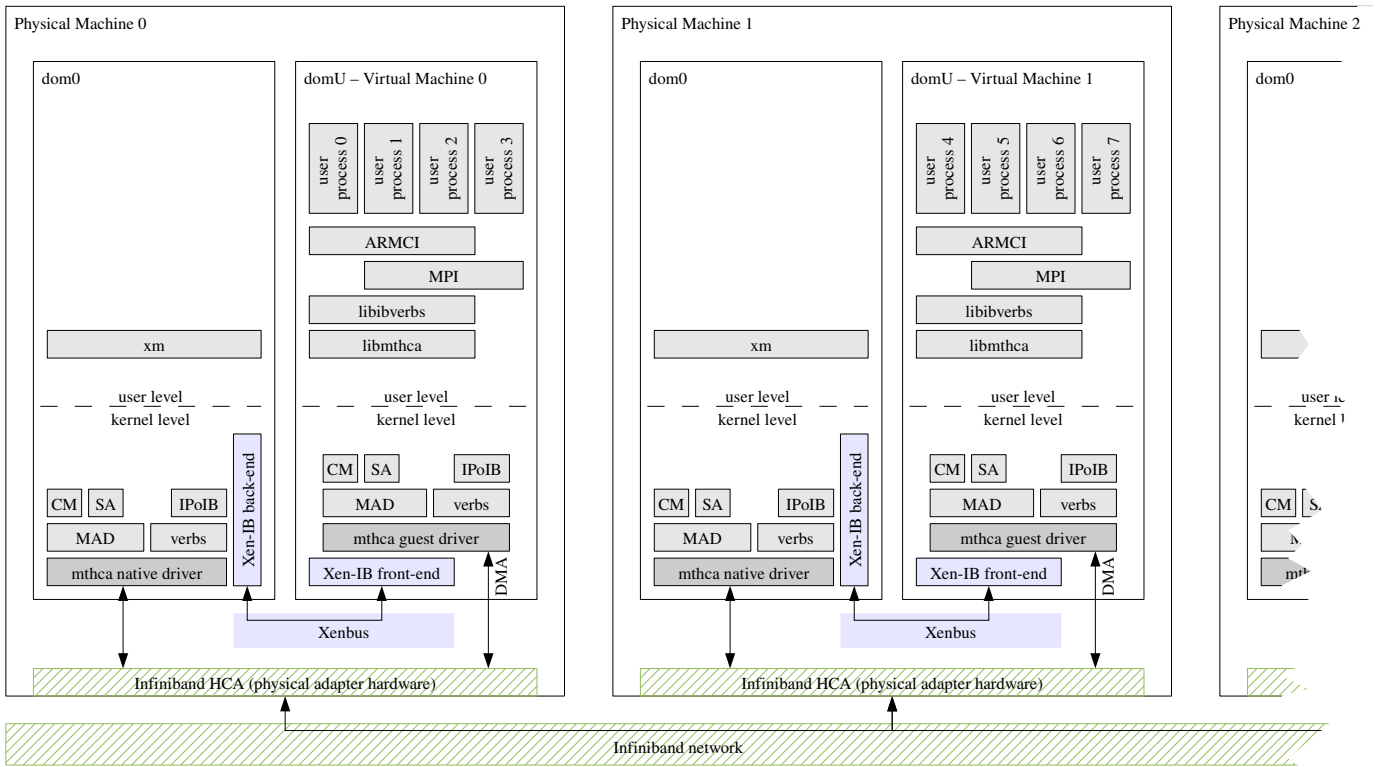


Fig. 1. The software stack adopted in our reference experimental setup.

could join and leave a multicast group, but only the first join and the last delete would remove the physical port from the multicast group. The proxy and multicast reference counting functions are implemented as additions to the Open Fabric Enterprise Distribution (OFED) stack for guest VMs.

VM migration is also supported by the VMM bypass driver. Because the VMM bypass driver provides direct guest access to hardware resources, migration involves the tear-down and re-establishment of those resources. This is in contrast to traditional device emulation or fully paravirtualized approaches where more software state can be preserved. Another limitation was that current virtualization environments only provide for notification of VM suspension and resume to be sent to kernel modules. Since the VMM bypass driver extends HCA access to the application level, suspend and resume messages must be propagated to applications. Applications must register for these messages and must indicate when they have completed their associated processing of these message so that migration may be completed. Guest access to the HCA I/O space and HCA DMA operations open guest-to-guest security concerns. One concern is the registration of guest memory regions with the HCA. This is addressed by a memory management ownership check in the back-end part of the VMM bypass driver. Another concern encountered was direct guest access to user access regions. These regions of HCA I/O space are allocated to applications and are used to signal and complete HCA data path operations. The system must ensure that access is only possible to the allocating guest

VM. This is being addressed with the memory management policy extension that allows the back-end driver to allocate and grant exclusive access on a per guest VM basis.

C. Communication Libraries

We consider applications that rely on both MPI and ARMCI [32], [33]. In particular, ARMCI has been used as a communication runtime layer in the implementation of several global address space programming models such as Global Arrays [34], Co-Array Fortran [35], SHMEM [36], and recently X10 [37]. By implementing coordination and a global recovery line at ARMCI level combined with the Xen virtualization, our goal is to provide a fundamental fault tolerant solution for the programming models that rely on ARMCI. As an MPI library, in our setup, we use *mvapich* version 0.9.7, with Mellanox extensions version 2.2.0.

The implementations of partitioned global address space programming models require efficient one-sided access to remote data. The Aggregate Remote Memory Copy Interface (ARMCI) library was developed to provide these functionalities. ARMCI offers an API that is simple to use, and at the same time it avoids performance degradation when applications which require multiple non-contiguous data transfers run on systems with high latency networks. Frequent multiple non-contiguous data transfers are often found in sections of code operating on dense multidimensional arrays or using scatter/gather operations. With ARMCI, the communication startup costs are incurred only once, whereas simpler APIs often

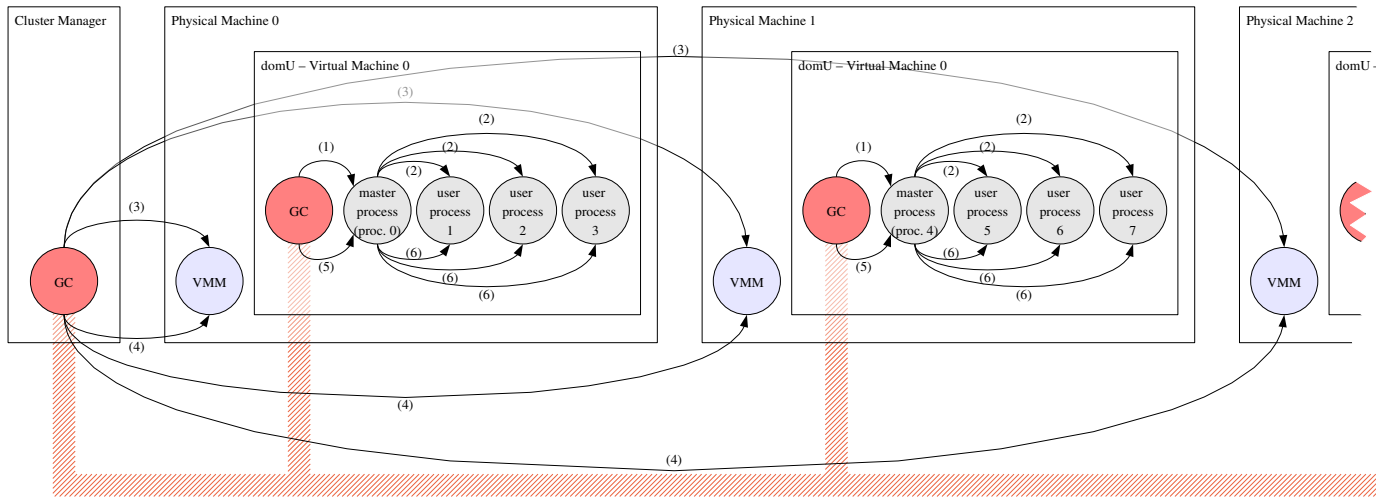


Fig. 2. Timing of a checkpoint/restart.

handle each contiguous portion of data as a separate message and incur setup costs each time. ARMCI has been optimized to use high-speed networks deployed in modern high speed interconnects [38]. For example, it has been optimized to use InfiniBand Verbs on the InfiniBand networks [39].

One of the fundamental contributions of this paper is an enhanced version of ARMCI which enables global coordination. Its internal design is described in detail in the next section. We have extended ARMCI to enable it to accept requests to perform global recovery lines at arbitrary times. In our design, we employ it to completely drain communication traffic and reach a globally consistent state before checkpointing or migration are performed.

III. GLOBAL RECOVERY LINES

This section describes in detail how we perform a Global Recovery Line (GRL) in the software design we propose. The objective of our GRL is to bring the entire HPC cluster to a quiescent, globally consistent state, which allows safe checkpointing or migration. The operations involved in the GRL must be implemented in a scalable way.

GRLs are crucial to implementing scalable fault-tolerance strategies, either based on checkpointing or proactive migration. In the first case, a GRL is required immediately before each checkpoint and checkpoints are invoked at regular time intervals. In the second case, a GRL is required before a migration is performed. A migration is usually requested when a distress signal is raised by one of the physical nodes because of an indicator of hardware failure, or risk of future hardware failure. In either case, the GRL ensures that the all the system (application, libraries, kernel components, virtualized and physical hardware) is in a state that allows a VM to be suspended and later resumed, possibly after a migration to another physical node. At this point, the reason that triggered the GRL is not relevant and has no effect on how the GRL is performed.

A GRL is needed with an InfiniBand network for two reasons:

- InfiniBand does not allow location-independent layer 2 and layer 3 addresses.
- InfiniBand hardware maintains stateful connections which are not accessible by software.

In the traditional case of IP over Ethernet, migration of VMs with their MAC and IP addresses unchanged is possible, because Ethernet devices allow new MAC and IP addresses to be associated with them dynamically. On the other hand, addresses in an InfiniBand network (local identifiers or LIDs) are managed by a dedicated service, the Subnet Manager (SM). The SM binds LIDs to physical hardware ports. Consequently, migrating a VM that is using a given LID to another location effectively disrupts its communications. Additionally, in TCP/IP connections are maintained at the OS level. In InfiniBand connections are maintained and managed in hardware. Connections are not location-independent and they are not directly accessible to the software.

A GRL is composed of a *drain* phase, which completes any ongoing communication, the *global silence* where it is possible to move node images and perform node migration or take a full checkpoint, and the *resume* phase in which the processing nodes acquire knowledge of the new network topology and continue their execution.

Significant parts of the GRL are the following: a Global Coordinator (GC), the Master Processes (MP), the non-master processes and the Virtual Machine Managers (VMM):

- The Global Coordinator (GC) is the application provided to the administrator to invoke a checkpoint or a proactive migration. It includes a stand-alone MPI application that performs a drain operation inside each VM, followed by an invocation to the VMM to checkpoint or migrate domains as necessary, and by a resume operation. Our GC is realized as a stand-alone tool for simplicity, but we envision future-generation clusters with their GC functions integrated in resource managers.

- In high-performance applications on SMP systems, it is common to have as many user processes as the number of available CPUs. Internally in ARMCI, one of these processes is chosen to be a Master Process to carry out on behalf of all tasks on that SMP node some activities like allocation of global memory etc. In the context of the Master Process, ARMCI creates a Server Thread to execute some one-sided operations that the underlying network protocols might not support, e.g., global mutexes and locks [40]. In our implementation, the GC sends drain/resume signals to MPs, which in turn propagate them to their non-master siblings running on the same SMP nodes.
- VMM here represents all the Xen components (the hypervisor, the user-mode command interface which allows to control it, etc...) which allow to pause, unpause, save, restore and migrate a virtual machine.

Figure 2 illustrates how the GC, processes and VMMs cooperate to perform a GRL. All the interactions are acknowledged by an appropriate communication mechanism (a message, a signal or a barrier), but this acknowledgment interactions are not represented in the figure for sake of simplicity. The hatched red area across the machines symbolizes that the GC is a parallel application, with components running on each VM and on an arbitrary machine used as a management console.

A sequence diagram showing the temporal dependencies and the barriers involved in the GRL mechanism we presented is given in Figure 3. In the figure, time proceeds vertically downwards (length of operations is not to scale); slant arrows represent signals, together with their propagation time; horizontal dashed lines represent synchronization barriers; hatched gray areas represent time in which a virtual machine is stopped, and the associated processes are not running.

Interactions occur in the following order as represented by the numbers in the Figure 3. 1) the local components of the GC in each VM send a drain signal to the MP associated to that VM. 2) MPs forward the same signal to their non-master siblings. The signal handler for the drain signal sets a *drain requested* flag to true in each user process. 3) All processes detect this flag, “drain their communication” and hit a barrier. Complete details of this draining operation are given below. When they leave the barrier, global quiescence is achieved; after hitting the barrier, all of them pause (in the POSIX sense) waiting for the resume signal. So, the GC can safely pause all the needed virtual machines.

At this time, checkpointing or migration can happen as desired (not represented in figure). 4) When computation needs to be resumed, the GC instructs the VMMs to unpause the respective VMs. 5) The GC then sends the MPs the resume signal. MP’s forward the resume signal to their non-master siblings. The resume signals wakes up all the user processes, which re-establish their communications, and hit a second barrier. When they leave the barrier, normal execution can continue.

The drain and resume signals have actually been implemented as the POSIX signals SIGUSR1 and SIGUSR2. The

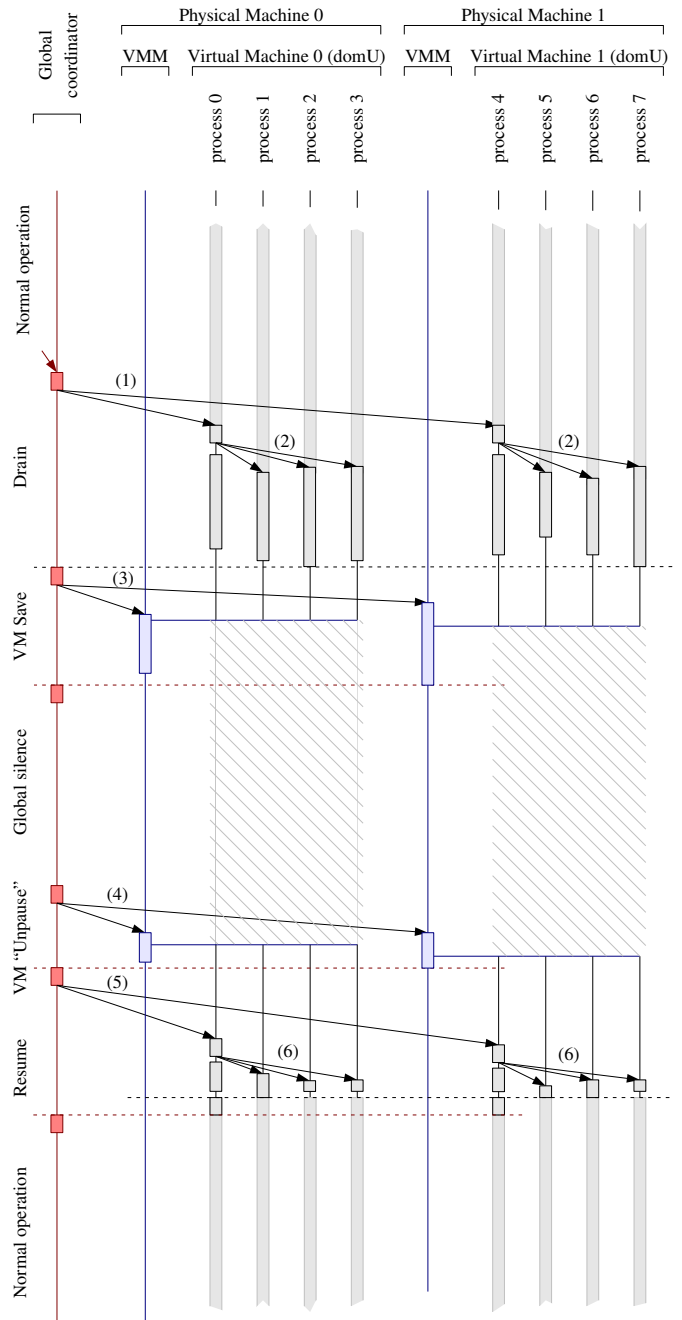


Fig. 3. Sequence diagram illustrating a global recovery line.

distributed portion of the GC is implemented as an MPI job, with a process running on each virtual machine plus the management machine.

The drain operation performed by the ARMCI library within each process completes the outstanding data transmissions (RDMA or InfiniBand send operations) it initiated and gracefully closes the InfiniBand connections that the process had opened. To guarantee that RDMA transfers do not violate memory isolation across processes, InfiniBand requires that all communication buffers be registered before use. In case of ARMCI, the user memory allocated through ARMCI memory

allocators in the application is registered so that zero-copy one-sided communication is possible directly through the RDMA read/write calls. Some non-contiguous ARMCI calls require assistance from the remote thread running in the Master Process to issue matching scatter/gather calls. This so-called host-assisted protocol is described in [39].

An InfiniBand connection involves a Queue Pair (QP) at each side. QPs are a send queue and a receive queue that hold data-transfer work descriptors. During a drain phase, completion and acknowledgment are awaited for each of the outstanding transfers. Finally, QPs are closed. At this point, in all the software and hardware components of the cluster system there is no valid connection state data which is related to the parallel processes. The InfiniBand and IP-over-InfiniBand (IPOIB) connections required to manage the processes and transfer the domains are untouched.

When all processes have completed their drain, they pause, waiting for the resume signal. At this point there are no running processes in the VMs. The GC receives an acknowledgment that the drain was globally successful and it issues a pause order to all the VMMs residing at each physical machine.

At this point, no user computation or communication is happening, and all the VMs are stopped. We call this phase *global silence*. During this phase, system tasks can be carried out safely. This is the right time to checkpoint the cluster by saving to disk the image of the VMs. Or, virtual machines running on faulty physical machines or machines which show distress indicators can be migrated to spare nodes. Other maintenance tasks can be carried out as well, including tasks which may require replacing hardware components and/or rebooting the physical machines.

When this operation is completed, the entire cluster must be brought in a fully operational state again. To do this, the effects of steps undertaken in the drain phase must be undone in reverse order.

First, the GC sends the wake-up command to the VMMs which are hosting the VMs (which is not necessarily the same set of VMMs where the VM were paused, because of the possibility of migrations). At this point, VMs are active but all the parallel processes inside them are still waiting for the resume signal.

The GC issues a resume signal to each master process, which forwards it to all of the non-master ones. All processes initiate a resume phase, in which they re-register their buffers, they re-establish QPs and perform a global exchange of the handles which allow the initiation of RDMA transfers. At the end of this resume operation, the communication is ready.

All the processes synchronize with a system barrier. When they leave the barrier, communication is fully operational. The portion of code that detected and managed the GRL returns to its caller, i.e. an invoked ARMCI communication primitive.

IV. RESULTS

In this experimental section we provide insight on the various components of our checkpoint/restart algorithm. We

emphasize two important results. First the overhead is minimal, less than 100 ms in most cases. And second, most of the components of the latency breakdown are application independent and only sensitive to the number of Xen images in the cluster configuration. For this reason, we focus our attention on the latency breakdown already detailed in the previous section.

Our experimental testbed consists of a cluster composed of 8 Dell PowerEdge 1950 computation nodes. Each node has two dual-core Intel Xeon Woodcrest 5160 processors running at 3.0 GHz, with 4 MByte L2 cache each (shared between the cores) and a 1.33 GHz front side bus. Each node has 8 GByte of DDR2 fully buffered RAM running at 667 MHz. The machines are connected by Mellanox Technologies MT25208 InfiniHost III Ex dual port 4X HCA adapters through a MT47396 Infiniscale III 24-port InfiniBand Switch. The HCA adapters are connected to a PCI-Express x8 bus.

The computational nodes use SUSE Linux Enterprise Server 10.0. Each physical machine runs a copy of the Xen hypervisor (version 3.0.2) and hosts exactly one dom0 domain and one domU domain. We use the privileged domain only for management purposes, while all the payload computation is carried out in the unprivileged paravirtualized domain. Since the physical machines we employ in our experiments are 4-way SMP (two dual-core processors), we configured the VMs as 4-way SMP, and subscribe exactly one task per CPU, i.e. 4 tasks per each VM. Further details on the software used in the experiments are given in Section II.

A production cluster would ensure high-performance, location-independent access to the virtual hard disk inside each VM through the use of a networked storage server and an appropriate protocol stack, such as SRP (SCSI Remote Direct Protocol) or iSCSI and iSER (iSCSI Extensions for RDMA). In our case the marginal relevance of storage performance makes such a solution overkill. We instead use a simpler and inexpensive (although less efficient) solution, based on an NFS server, which runs on a Silicon Mechanics 1U server, with 2 Intel Xeon 3.2 GHz processors, 8 GByte of RAM and a RAID 5 system containing 4 SATA disks, each 400 MByte in size; it runs Red Hat Enterprise Linux 4 as an operating system. A separate Gigabit Ethernet connects the computation nodes to the NFS server. The computation nodes employ an on-board Broadcom NetXtreme II BCM5708 Gigabit Ethernet adapter. We use a NetGear GSM7248 Ethernet switch.

As a benchmark application, we used the standard LU decomposition example which comes shipped with ARMCI. The benchmark decomposes large sparse matrices.

Given below is the analysis of delays involved in a global checkpoint ($D_{\text{checkpoint}}$) and in a proactive migration ($D_{\text{proactive}}$) operation respectively. In both cases we assume that the global coordinator issues a resume order immediately after the checkpoint or the save is complete. The delays can be expressed as follows:

$$D_{\text{checkpoint}} = 2 \cdot D_{\text{signal}} + D_{\text{drain}} + D_{\text{save}} + D_{\text{resume}}$$

$$D_{\text{proactive}} = 2 \cdot D_{\text{signal}} + D_{\text{drain}} + D_{\text{migrate}} + D_{\text{resume}}$$

where the delay terms have the following meaning:

- D_{signal} is the time required to forward a signal to each process in the parallel job and obtain an acknowledgment; this includes the network latencies due to barriers and operating system signal service latency. This term appears twice because there are two signals (drain and resume) to deliver.
- D_{drain} is the time required to drain the outstanding transfers. This depends on the performance of the network, and on the average amount and size of outstanding transfers generated by the application considered.
- D_{save} is the time required to save the image to disk. This depends on the VM memory size and on the write bandwidth provided by the storage subsystem.
- D_{migrate} is the time required to perform a VM migration with Xen. This depends on the VM memory size and the network bandwidth;
- D_{resume} is the time required to re-establish communication. This depends on the network latency and on the number of nodes in the cluster.

In case a checkpointing approach is adopted, the restart delay also is meaningful:

$$D_{\text{restart}} = D_{\text{signal}} + D_{\text{restore}} + D_{\text{resume}}$$

This includes the delay required to restore the image snapshot from storage (which is usually shorter than D_{save}).

Figure 4 illustrates how terms composing the D_{drain} delay vary when the number of processes is increased. Draining time is the only application-dependent term and it accounts for the time spent waiting for the outstanding transfers to complete.

Figure 5 illustrates how terms composing the D_{resume} delay vary when the number of processes is increased. There are no application-dependent terms in D_{resume} .

Figure 6 shows the delay measured when a VM with different memory footprints (between 64 Mbytes and 4 Gbytes) is saved and restored by Xen. Save is performed by writing the VM image on the local disk, hot restore is performed by reading the VM from the local disk when the entire VM image is cached in the main memory, while cold restore reads the VM image directly from local disk.

Data show that the save/restore delay is well modeled as a function of the VM memory size s , an initial setup delay d_{save} and the storage subsystem bandwidth B_{save} :

$$D_{\text{save}} = d_{\text{save}} + s/B_{\text{save}}.$$

For the experimental data represented in Figure 6, linear regression estimates $d_{\text{save}} = 1061.1$ ms, and $B_{\text{save}} = 192.9$ Mbyte/s. For a hot restore, $d_{\text{restore}} = 330.5$ ms, and $B_{\text{restore}} = 603.9$ Mbyte/s. For a cold restore, $d_{\text{restore}} = 1083.1$ ms, and $B_{\text{restore}} = 85.4$ Mbyte/s.

Figure 7 shows the delay due to a VM migration across cluster nodes. We assumed a worst-case scenario in which a non-live migration is performed, and all of the pages of the VM memory are considered dirty, and transferred. Data show

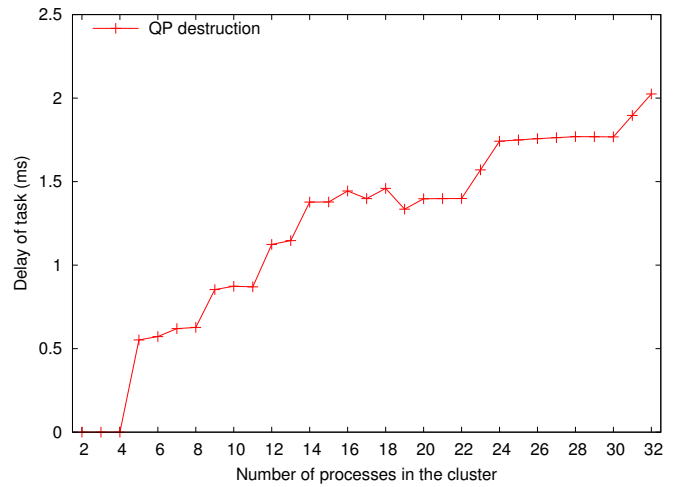
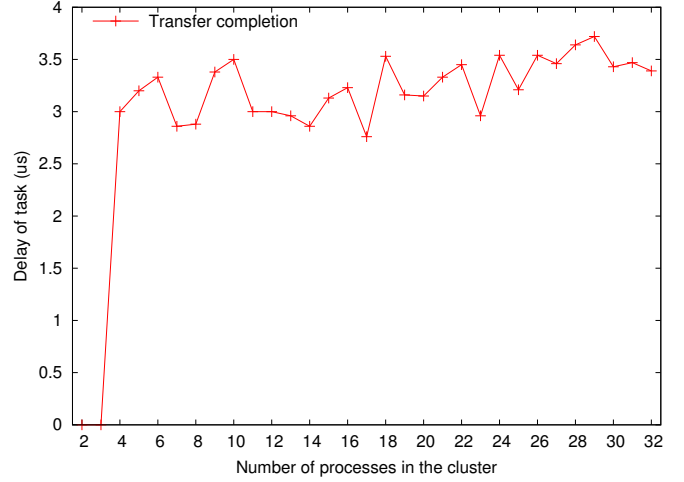


Fig. 4. Scalability of the D_{drain} delay sub-terms broken down into time for transfer completion and time for QP destruction.

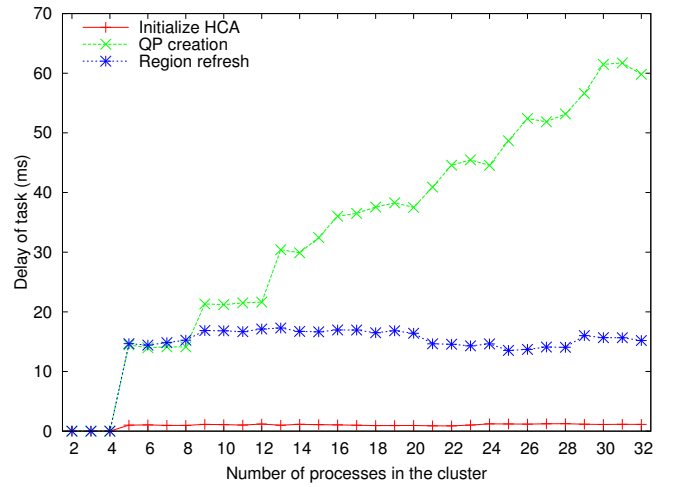


Fig. 5. Scalability of the D_{resume} delay sub-terms.

that the migration delay is a function of the VM memory

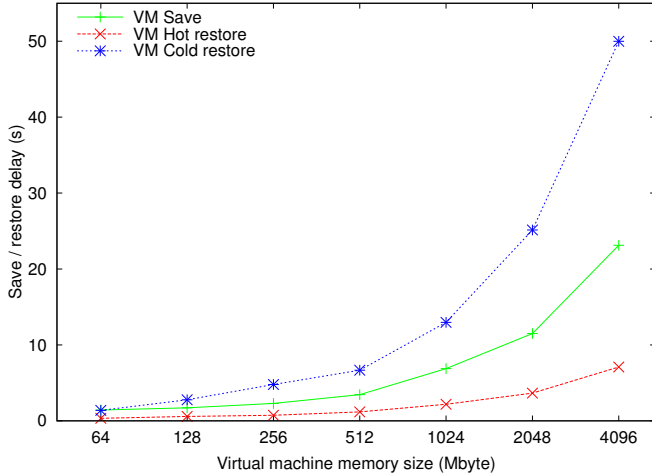


Fig. 6. Save and restore delays measured under our experimental conditions.

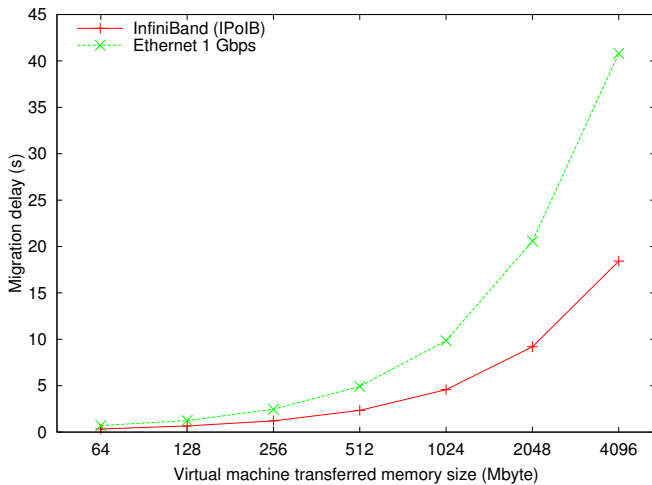


Fig. 7. Migration delays using IP over InfiniBand (IPoIB) and Ethernet.

size s , an initial setup delay d_{migrate} and the available network bandwidth B_{migrate} :

$$D_{\text{migrate}} = d_{\text{migrate}} + s/B_{\text{migrate}}.$$

The available network bandwidth depends on a number of factors including the signaling rate and the transfer protocol used. For the sake of simplicity, in our experiments we used IP over InfiniBand (IPoIB), which yields quite a low throughput and is significantly suboptimal (1/4) with respect to the maximum performance InfiniBand could obtain if the verbs interface were used. Linear regression finds $d_{\text{migrate}} = 47.8$ ms and $B_{\text{migrate}} = 223.2$ MB/s.

Table I summarizes the relative importance of the delay terms introduced above, during a checkpointing operation. This table represents the case where each VM has a memory size equal to 2 GByte. As the data suggests, the delay is dominated by the Xen save operation, and the entire overhead

introduced by our methodology impacts less than 1% on the overall delay.

Phase	Delay	
Signal propagation	2.21 ms	0.02%
Draining*	30.79 ms	0.27%
QP destruction	2.02 ms	0.02%
Save VM image	11501.00 ms	99.02%
Signal propagation	2.21 ms	0.02%
Initialize HCA	1.14 ms	0.01%
Exchange of LIDs	0.19 ms	0.00%
QP creation	59.81 ms	0.51%
Region refresh	15.18 ms	0.13%
Total	11614.55 ms	100.00%

TABLE I

Typical delays involved in checkpointing a 32-process LU-decomposition application. (*): draining is the only application-dependent delay term.

V. CONCLUSION

We presented a novel software infrastructure that supports automatic and user-transparent migration of PGAS applications for the purpose of fault tolerance. This approach relies on our implementation of a Linux device driver that enhances the existing kernel drivers to allow Xen migration over the latest generation of commercially available InfiniBand network adapters. In addition, we also proposed and implemented an algorithm to enforce the Global Recovery Line (GRL), a common point in time where a coordinated checkpoint can be taken without affecting the correctness of the application. The focus on the Partitioned Global Address Space programming models simplified our design and allowed a seamless integration of GRL and the migration schemes with ARMCI, a communication run-time layer used in implementation of multiple PGAS programming models.

Despite its prototype status, our software infrastructure demonstrates that it is indeed possible to virtualize all the resources of a processing node, including the communication network, with a negligible overhead. The cost of the checkpoint/migration scheme is relatively low, and mostly affected by the speed of the I/O devices (an important factor in itself not covered in this paper due to time and space limitations).

Our experimental results show that components of our automatic global recovery line approach take just a few milliseconds or less, and most are insensitive to the characteristics of the user applications. Overall a node migration can be achieved in tens of milliseconds, a negligible delay if checkpoints are taken every few minutes.

We plan to continue our work with a more extensive analysis of user applications and how I/O devices impact the performance of the overall checkpoint/migration strategies.

REFERENCES

- [1] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, pages 589–604, July/September 2005.

- [2] N. R. Adiga and et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the Supercomputing, also IBM research report RC22570 (W0209-033)*, November 16–22, 2002.
- [3] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, May 25–29, 2002.
- [4] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Usenix Winter 1995 Technical Conference*, January 16–20, 1995.
- [5] J. S. Plank, M. Beck, and G. Kingsley. Compiler-Assisted Memory Exclusion for Fast Checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.
- [6] J. S. Plank, Y. Kim, and J. J. Dongarra. Diskless Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.
- [7] J. S. Plank and K. Li. ickp — A Consistent Checkpointer for Multicomputers. *IEEE Parallel and Distributed Technologies*, 2(2):62–67, Summer 1994.
- [8] A. Geist and C. Engelmann. Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors. Oak Ridge National Laboratory, 2002.
- [9] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, May 25, 1997.
- [10] C. J. Li and W. K. Fuch. CATCH - Compiler Assisted Techniques for Checkpointing. In *Proceedings of the International Symposium on Fault Tolerant Computing*, pages 74–81, June 1990.
- [11] Fabrizio Petrini, Kei Davis, and José Carlos Sancho. System-Level Fault-Tolerance in Large-Scale Parallel Machines with Buffered Coscheduling. In *9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems (FTPDS04)*, Santa Fe, NM, April 2004.
- [12] Qianfeng Jiang and D. Manivannan. An Optimistic Checkpointing and Selective Message Logging Approach for Consistent Global Checkpointing Collection in Distributed Systems. In *IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, March 2007.
- [13] José Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernández, and Eitan Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. In *Proceedings of the International Parallel and Distributed Processing Symposium 2004 (IPDPS04)*, Santa Fe, NM, April 2004.
- [14] Joseph Ruscio, Michael Heffner, and Srinidhi Varadarajan. DejaVu: Transparent User-Level Checkpointing, Migration and Recovery for Distributed Systems. In *IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, March 2007.
- [15] Syantan Chakravorty and Laxmikant Kalé. A Fault Tolerance Protocol for Fast Fault Recovery. In *IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, March 2007.
- [16] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance under UNIX. *ACM Transactions on Computer Systems (TOCS)*, 7(1):1–24, February 1989.
- [17] Roberto Gioiosa, José Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. Transparent Incremental Checkpointing at Kernel Level: A Foundation for Fault Tolerance for Parallel Computers. In *IEEE/ACM Conference on Supercomputing SC'05*, Seattle, WA, November 2005.
- [18] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen Scott. A Job Pause Service under LAM/MPI + BLCR for Transparent Fault Tolerance. In *IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, March 2007.
- [19] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
- [20] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhableswar Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *USENIX Annual Technical Conference*, Boston, MA, June 2006.
- [21] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhableswar Panda. A Case for High Performance Computing with Virtual Machines. In *The 20th ACM International Conference on Supercomputing (ICS'06)*, Cairns Queensland, Australia, June 2006.
- [22] Wei Huang, Jiuxing Liu, M. Koop, Bulent Abali, and Dhableswar K. Panda. Nomad: Migrating OS-bypass networks in virtual machines. In *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, June 2007.
- [23] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. BlueGene/L Failure Analysis and Prediction Models. In *International Conference on Dependable Systems and Networks*, Philadelphia, PA, June 2006.
- [24] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *In Proceedings of the OASIS ASPLOS Workshop, 2004*.
- [25] A. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the 21th ACM International Conference on Supercomputing (ICS) 2007*, Seattle, WA, USA, June 16–20, 2007. To appear.
- [26] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.
- [27] Lamia Youseff, Richard Wolski, Brent C. Gorda, and Chandra Krintz. Paravirtualization for HPC systems. In Geyong Min, Beniamino Di Martino, Laurence Tianruo Yang, Minyi Guo, and Gudula Rünger, editors, *ISPA Workshops*, volume 4331 of *Lecture Notes in Computer Science*, pages 474–486. Springer, 2006.
- [28] Ludmila Cherkasova and Rob Gardner. Measuring CPU overhead for i/o processing in the Xen virtual machine monitor. In *Proceedings of the USENIX 2005 Annual Technical Conference, General Track*, pages 387–390, 2005.
- [29] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23, New York, NY, USA, 2005. ACM Press.
- [30] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Darius Buntinas, Weikuan Yu, Balasubraman Chandrasekaran, Ranjit M. Noronha, Pete Wyckoff, and Dhableswar K. Panda. MPI over infiniband: Early experiences. Technical Report OSU-CISRC-10/02-TR25, Ohio Supercomputer Center, The Ohio State University, August 2003.
- [31] Jiuxing Liu, Jiesheng Wu, and Dhableswar K. Panda. High performance RDMA-based MPI implementation over infiniband. *Int. J. Parallel Program.*, 32(3):167–198, 2004.
- [32] J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *RTSP of IPPS/SDP'99*, 1999.
- [33] Jarek Nieplocha, Vinod Tipparaju, Manojkumar Krishnan, and Dhableswar Panda. High Performance Remote Memory Access Communications: The ARMCI Approach. *International Journal of High Performance Computing and Applications*, 20(2), 2006.
- [34] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.
- [35] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A Multiplatform Co-Array Fortran Compiler. In *Proceedings of the 13th Intl. Conference of Parallel Architectures and Compilation Techniques*, Antibes Juan-les-Pins, France, September 29 - October 3 2004.
- [36] K. Parzyszek, J. Nieplocha, and R. A. Kendall. Generalized Portable SHMEM Library for High Performance Computing. In M. Guizani and X. Shen, editors, *IATED Parallel and Distributed Computing and Systems*, pages 401–406, Las Vegas, Nevada, 2000. IATED.
- [37] Ebcioglu, Saraswat, and Sarkar. X10: An experimental language for high productivity programming of scalable systems. In *P-HEC Workshop*, 2005.
- [38] Jarek Nieplocha, Vinod Tipparaju, Manojkumar Krishnan, and Dhableswar Panda. High Performance Remote Memory Access Communications: The ARMCI Approach. *International Journal of High Performance Computing and Applications*, 20(2), 2006.
- [39] V. Tipparaju, G. Santharaman, J. Nieplocha, and D. K. Panda. Host-assisted zero-copy remote memory access communication on InfiniBand. In *International Parallel and Distributed Computing Symposium (IPDPS)*, Santa Fe, NM, USA, 2004. IEEE.
- [40] D. Buntinas, A. Saify, D.K. Panda, and J. Nieplocha. Optimizing Synchronization Operations for Remote Memory Communication Systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, April 2003.