Proactive Fault Tolerance for HPC with Xen Virtualization *

Arun Babu Nagarajan[1], Frank Mueller[1], Christian Engelmann[2], Stephen L. Scott[2]

[1] Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534

[2] Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6016

e-mail: mueller@cs.ncsu.edu

ABSTRACT

Large-scale parallel computing is relying increasingly on clusters with thousands of processors. At such large counts of compute nodes, faults are becoming common place. Current techniques to tolerate faults focus on reactive schemes to recover from faults and generally rely on a checkpoint/restart mechanism. Yet, in today's systems, node failures can often be anticipated by detecting a deteriorating health status.

Instead of a reactive scheme for fault tolerance (FT), we are promoting a proactive one where processes automatically migrate from "unhealthy" nodes to healthy ones. Our approach relies on operating system virtualization techniques exemplified by but not limited to Xen. This paper contributes an automatic and transparent mechanism for proactive FT for arbitrary MPI applications. It leverages virtualization techniques combined with health monitoring and load-based migration. We exploit Xen's live migration mechanism for a guest operating system (OS) to migrate an MPI task from a health-deteriorating node to a healthy one without stopping the MPI task during most of the migration. Our proactive FT daemon orchestrates the tasks of health monitoring, load determination and initiation of guest OS migration. Experimental results demonstrate that live migration hides migration costs and limits the overhead to only a few seconds making it an attractive approach to realize FT in HPC systems. Overall, our enhancements make proactive FT a valuable asset for long-running MPI application that is complementary to reactive FT using full checkpoint/restart schemes since checkpoint frequencies can be reduced as fewer unanticipated failures are encountered. In the context of *OS virtualization*, we believe that this is the first comprehensive study of proactive fault tolerance where live migration is actually triggered by health monitoring.

*The research at NCSU was supported in part by NSF grants CCR-0237570(CAREER), CNS-0410203, CCF-0429653 and DOE DE-FG02-05ER25664. The research at Oak Ridge National Laboratory (ORNL) is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. ORNL is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

ICS'07 June 18–20, 2007, Seattle, WA, USA.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; D.4.8 [**Operating Systems**]: Performance—*measurements*

General Terms

Performance, Reliability

Keywords

High-Performance Computing, Proactive Fault Tolerance, Virtualization

1. INTRODUCTION

High-end parallel computing is relying increasingly on large clusters with thousands of processors. At such large counts of compute nodes, faults are becoming common place. For example, today's fastest system, BlueGene/L (BG/L) at Livermore National Laboratory with 65,536 nodes, was experiencing faults at the level of a dual-processor compute card at a rate of 48 hours during initial deployment [22]. When one node fails, a 1024-processor midplane had to be temporarily shut down to replace the card.

Results from related work [20], depicted in Table 1, show that the existing reliability of larger HPC clusters is currently constrained by a mean time between failures (MTBF) / interrupts (MTBI) in the range of 6.5-40 hours, depending on the maturity / age of the installation. The most common causes of failure were processor, memory and storage errors / failures. This is reinforced by a study of HPC installations at Los Alamos National Laboratory (LANL) indicating that, on average, 50% of all failures were due to hardware and almost another 20% due to software with more than 15% of the remaining failure cases unaccounted for in terms of their cause [36]. Another study conducted by LANL estimates the MTBF, extrapolating from current system performance [30], to be 1.25 hours on a petaflop machine.

System	# CPUs	MTBF/I
ASCI Q	8,192	6.5 hrs
ASCI White	8,192	5/40 hrs ('01/'03)
PSC Lemieux	3,016	9.7 hrs
Google	15,000	20 reboots/day

Table 1: Reliability of HPC Clusters

Commercial installations, such as Google (see Table 1) experience an interpolated fault rate of just over one hour for equivalent

number of nodes, yet their fault-tolerant middleware hides such failures altogether so that user services remain completely intact [17]. In this spirit, our work focuses on fault-tolerant middleware for HPC systems. More specifically, this paper promotes operating system virtualization as a means to support fault tolerance (FT). Since OS virtualization is not an established method in HPC due to the potential overhead of virtualization, we conducted a study measuring the performance of the NAS Parallel Benchmark (NPB) suite [42] using Class C inputs over Xen [6]. We compared three Linux environments: Xen Dom0 Linux (privileged domain 0 OS), Xen DomU Linux (a regular guest OS), and a regular, non-Xen Linux version on the same platform (see Section 3 for configuration details). The results in Figure 1 indicate a relative speed of 0.81-1.21 with an average overhead of 1.5% and 4.4% incurred by Xen DomU and Dom0, respectively. This overhead is mostly due to the additional software stack of virtualizing the network device, as OS-bypass experiments with InfiniBand and extensions for superpages have demonstrated [26, 25]. With OS bypass, the overhead is lowered to $\approx \pm 3\%$ for NAS PB Class A. In our experiments with Class C inputs, CG and LU result in a reproducible speedup (using 10 samples for all tests) for one or both Xen versions, which appears to be caused by memory allocation policies and related activities of the Xen Hypervisor that account for 11% of CG's runtime, for example. The details are still being investigated. Hence, OS virtualization accounts for only marginal overhead and can easily be amortized for large-scale systems with a short MTBF.

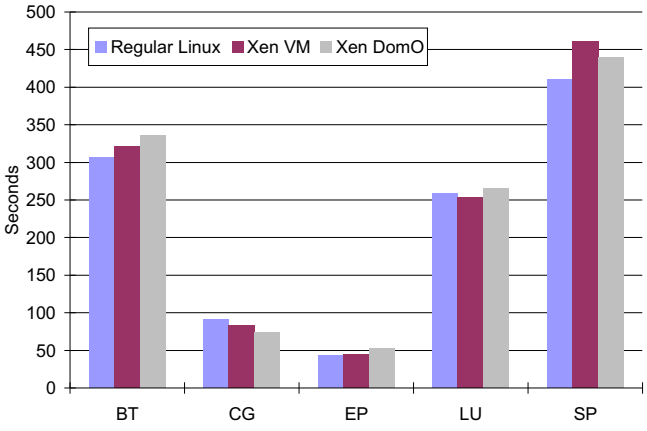

Figure 1: Xen Overhead for NAS PB, Class C, 16 Nodes

Current techniques to tolerate faults focus on reactive schemes where fault recovery commonly relies on a checkpoint/restart (C/R) mechanism. However, the LANL study [30] also estimates the checkpointing overhead based on current techniques to prolong a 100 hour job (without failure) by an additional 151 hours in petaflop systems.

In today's systems, node failures can often be anticipated by detecting a deteriorating health status using monitoring of fans, temperatures and disk error logs. Recent work focuses on capturing the availability of large-scale clusters using combinatorial and Markov models, which are then compared to availability statistics for large-scale DOE clusters [37, 32]. Health data collected on these machines is used in a reactive manner to determine a checkpoint interval that trades off checkpoint cost against restart cost, even though many faults could have been anticipated. Hence, instead of a reactive scheme for FT, we are promoting a proactive one that migrates processes away from "unhealthy" nodes to healthy ones. Such an approach has the advantage that checkpoint frequencies can be reduced as sudden, unexpected faults should become the exception. The availability of spare nodes is becoming common place in recent cluster acquisitions. We expect such spare nodes to become a commodity provided by job schedulers upon request. Our experiments assume availability of 1-2 spare nodes.[1]

The feasibility of health monitoring at various levels has recently been demonstrated for temperature-aware monitoring, *e.g.*, by using ACPI [3], and, more generically, by critical-event prediction [33]. Particularly in systems with thousands of processors, such as BG/L, fault handling becomes imperative, yet approaches range from application-level and runtime-level to the level of operating system (OS) schedulers [9, 10, 11, 27]. These and other approaches are discussed in more detail in Section 5. They differ from our approach in that we exploit OS-level virtualization combined with health monitoring and live migration.

In related, orthogonal work [40], experiments were conducted with process-level BLCR [14] to assess the overhead of saving and restoring the image of an MPI application on a faulty node, which we compare with the save/restore overhead over Xen [6]. For BLCR, this comprises the process of an MPI task while for Xen, the entire guest OS is saved. Process-level FT with BLCR showed an overhead of 8-10 seconds for BLCR and 15-23 seconds for Xen for NPB programs under Class C inputs on a common experimental platform. Variations are mostly due to the memory requirements of specific benchmarks. These memory requirements also dominate those of the underlying OS, which explains why Xen remains competitive in these experiments. From this, we conclude that both process-level and OS-level C/R mechanisms are viable alternatives. This paper focuses on the OS virtualization side.

We have designed and implemented an automatic and transparent mechanism for proactive FT of arbitrary MPI applications over Xen [6]. A novel proactive FT daemon orchestrates the tasks of health monitoring, load determination and initiation of guest OS migration. To this extent, we exploit the intelligent performance monitoring interface (IPMI) for health inquiries to determine if thresholds are violated, in which case migration should commence. Migration targets are determined based on load averages reported by Ganglia. Xen supports *live* migration of a guest OS between nodes of a cluster, *i.e.*, MPI applications continue to execute during much of the migration process [12]. In a number of experiments, our approach has shown that live migration can hide migration costs such that the overall overhead is constrained to only a few seconds. Hence, live migration provides an attractive solution to realize FT in HPC systems. Our work shows that proactive FT complements reactive schemes for long-running MPI jobs. Specifically, should a node fail without prior health indication or while proactive migration is in progress, our scheme reverts to reactive FT by restarting from the last checkpoint. Yet, as proactive FT has the potential to prolong the mean-time-to-failure, reactive schemes can lower their checkpoint frequency in response, which implies that proactive FT can lower the cost of reactive FT. In the context of *OS virtualization*, this appears to be the first comprehensive study of proactive fault tolerance where live migration is actually triggered by health monitoring.

The paper is structured as follows. Section 2 presents the design and implementation of our health monitoring and migration system with its different components. Section 3 describes the experimental setup. Section 4 discusses experimental results for a set of bench-

[1]Our techniques also generalize to task sharing on a node should not enough spare nodes be available, yet the cost is reduced performance for tasks on such a node. This may result in imbalance between all tasks system-wide and, hence, decrease overall performance. In this model, tasks sharing a node would still run within multiple guest OSs hosted by a common hypervisor on a node.

marks. Section 5 contrasts this work to prior research. Section 6 summarizes the contributions.

2. SYSTEM DESIGN AND IMPLEMENTATION

A proactive fault tolerance system, as the name implies, should provide two mechanisms, namely one for proactive decision making and another to address load balancing, which, in combination, provide fault tolerance. An overview of the system components and their interaction is depicted in Figure 2. Each node hosts an instance of the Xen Virtual Machine Monitor (VMM). On top of the VMM runs a privileged/host virtual machine, which is a para-virtualized Linux version in our case. In addition, a guest virtual machine (also Linux) runs on top of the Xen VMM as well. The privileged virtual machine hosts, among others, a daemon for Ganglia, which aids in selecting the target node for migration, and our proactive FT daemon (PFTd) used to monitor health and initiate migration. The guest virtual machines form a multi-purpose daemon (MPD) ring of all cluster nodes [8] on which the MPI application can run (using MPICH-2). Other MPI runtime systems would be handled equally transparently by Xen for the migration mechanism. Upon deteriorating health, determined through the monitoring capabilities of the baseboard management controller (BMC), the entire guest VM is migrated to another node that already hosts a privileged VM but no guest VM. We will describe each of these components of our system in the following.

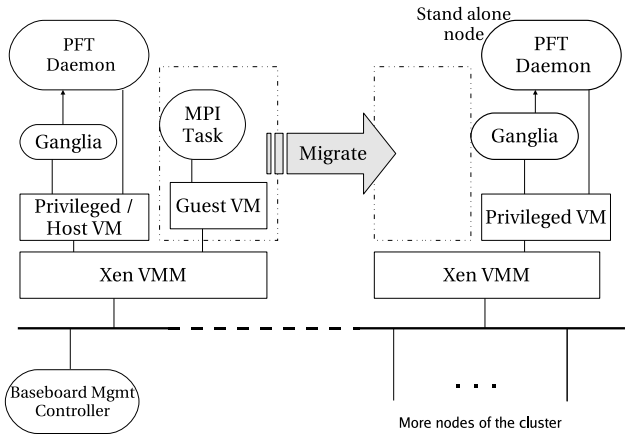

Figure 2: Overall setup of the components

2.1 Fault Tolerance over Xen

To provide an effective fault tolerance system, a mechanism is required that gracefully aids the relocation of an MPI task, thereby enabling it to run on a different physical node with minimum possible overhead. More importantly, the MPI task should not be stopped while migration is in progress. Xen provides exactly this capability. Xen is a para-virtualized environment that requires the hosted virtual machine to be adapted to run on the Xen virtual machine monitor (VMM). Applications, however, need not be modified. On top of the VMM runs a privileged/host virtual machine with additional capabilities exceeding those of other virtual machines. We can start other underprivileged guest virtual machines on that host VM using the command line interface. Most significantly, Xen provides *live migration*, which enables the guest VM to be transferred from one physical node to another [12]. Xen's mechanism exploits the pre-migration methodology where all state is transferred prior to target activation. Migration preserves the state of all the processes on the guest, which effectively allows the VM to continue execution without interruption. Migration can be initiated by specifying the name of guest VM and the IP of the destination physical node hosted by the VM. Live migration occurs as a sequence of phases:

1. When the migration command is initiated, the host VM inquires if the target has sufficient resources and reserves them as needed in a so-called pre-migration and reservation step.

2. Next, the host VM sends all pages of the guest VM to the destination node in a first iteration of the so-called pre-copy step. Prior to sending a page, the corresponding modified (dirty) bit is cleared in the shadow page table entry (PTE) of the guest OS. During the transfer, the guest VM is still running. Hence, it will modify data in pages that were already send. Using page protection, a write to already sent pages will initially result in a trap. The trap handler then changes the page protection such that subsequent writes will no longer trap. Furthermore, the dirty bit of the page is automatically set in the PTE so that it can later be identified.

3. The host VM now starts sending these dirty pages iteratively in chunks during subsequent iterations on the pre-copy step until a heuristic indicates that pre-copy is no longer beneficial. For example, the ratio of modified pages to previously sent pages (in the last iteration) can be used as a termination condition. At some point, the rate of modified pages to transfer will stabilize (or nearly do so), which causes a transition to the next step. The portion of the working set that is subject to write accesses is also termed in writable working set (WSS) [12], which gives an indication of the efficiency of this step. An additional optimization also avoids copying modified pages if they are frequently changed.

4. Next, the guest VM is actually stopped and the last batch of modified pages is sent to the destination where the guest VM restarts after updating all pages, which comprises the so-called stop & copy, commitment and activation steps.

The actual downtime due to the last phase has been reported to be as low as 60 ms [12]. Keeping an active application running on the guest VM will potentially result in a high rate of page modifications. We observed a maximum actual downtime of around three seconds for some experiments, which shows that HPC codes may have higher rates of page modifications. The overall overhead contributed to the total wallclock time of the application on the migrating guest VM can be attributed to this actual downtime plus the overhead associated with the active phase when dirty pages are transferred during migration. Experiments show that this overhead is negligible compared to that of the total wallclock time for HPC codes.

2.2 Health monitoring with OpenIPMI

Any system that claims to be proactive must effectively predict an event before it occurs. As the events to be predicted are fail-stop node failures in our case, a health monitoring mechanism is needed. To this extent, we employ the Intelligent Platform Management Interface (IPMI). IPMI is an increasingly common management/monitoring interface that provides a standardized message-based mechanism to monitor and manage hardware, a task performed in the past by software with proprietary interfaces.[2] The

[2]Alternatives to IPMI exist, such as lm_sensor, but they tend to be

Baseboard Management Controller (BMC), depicted in Figure 2, is equipped with sensors to monitor different properties. For example, sensors provide data on temperature, fan speed, and voltage. IPMI provides a portable interface for reading these sensors to obtain data for health monitoring.

OpenIPMI [2] provides an open-source higher-level abstraction from the raw IPMI message-response system. We use the OpenIPMI API to communicate with the Baseboard Management Controller of the backplane and to retrieve sensor readings. Based on the readings obtained, we can evaluate the health of the system. We have implemented a system with periodic sampling of the BMC to obtain readings of different properties. OpenIPMI also provides an event-triggered mechanism allowing one to specify ,*e.g.*, a sensor reading exceeding a threshold value and register a notification request. When the specified event actually occurs, notification is triggered by activating an asynchronous handler. This event-triggered mechanism might offload some overhead from the application side since the BMC takes care of event notification. Unfortunately, OpenIPMI did not provide stable event notification at the time of writing. Hence, we had to resort to the more costly periodic sampling alternative.

2.3 Load Balancing with Ganglia

When a node failure is predicted due to deteriorating health, as indicated by the sensor readings, a target node is selected to migrate the virtual machine to. We utilize Ganglia [1], a widely used, scalable distributed monitoring system for HPC systems, to select the target node in the following manner. All nodes in the cluster run a daemon that monitors local resource (*e.g.*, CPU usage) and sends multicast packets with the monitored data. All nodes listen to such messages and update their local view in response. Thus, all nodes have an approximate view of the entire cluster.

By default, Ganglia measures the CPU usage, memory usage and network usage among others. Ganglia provides extensibility in that application-specific metrics can also be added to the data dissemination system. For example, our systems requires the capability to distinguish whether a physical node runs a virtual machine or not. Such information can be added to the existing Ganglia infrastructure. Ganglia provides a command line interface, gmetric, to this respect. An attribute specified through the gmetric tool indicates whether the guest VM is running or not on a physical node. Once added, we obtain a global view (of all nodes) available at each individual node. Our implementation selects the target node for migration as the one which does not yet host a guest virtual machine and has the lowest load based on CPU usage. We can further extend this functionality to check if the selected target node has sufficient available memory to handle the incoming virtual machine. Even though the Xen migration mechanism claims to check the availability of sufficient memory on the target machine before migration, we encountered instances where migration was initiated and the guest VM crashed on the target due to insufficient memory. Furthermore, operating an OS at the memory limit is known to adversely affect performance.

2.4 PFT Daemon Design

We have designed and implemented a proactive fault tolerance daemon (PFTd). In our system depicted in Figure 2, each node runs an instance of the PFTd on the privileged VM, which serves as the primary driver of the system. The PFTd gathers details, interprets them and makes decisions based on the data gathered. The PFTd provides three components: Health monitoring, decision making and load balancing (see Figure 3). After initialization, the PFTd monitors the health state and checks for threshold violations. Once a violation is detected, Ganglia is contacted to determine the target node for migration before actual migration is initiated.

system-specific (x86 Linux) and may be less powerful. Also, disk monitoring can be realized portably with SMART.

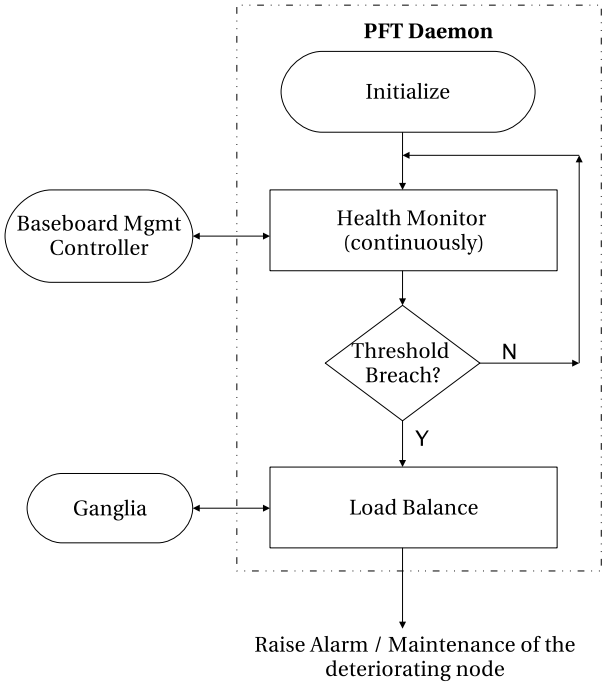

Figure 3: Proactive Fault Tolerance Daemon

Upon PFTd initialization, a configuration file containing a list of parameters to be monitored is consulted. In addition to a parameter name, the lower and upper thresholds for that particular parameter can also be specified. For example, for dual processor machines, the safe temperature range for two CPUs and the valid speed range for system fans is specified. Next, the PFTd initializes the OpenIPMI library and sets up a connection for the specified network destination (determined by the type of interface, *e.g.*, as LAN, remote hostname and authentication parameters, such as userid and password). A connection to the BMC becomes available after successful authentication. A domain is created (using the domain API) so that various entities (fans, processors, etc.) are attached to it. The sensors monitor these entities.

OpenIPMI, as we discussed earlier, provides an event-driven system interface, which is somewhat involved, as seen next. We need to register a handler for an event with the system. Whenever the event occurs, that particular handler will be invoked. While creating a domain, a handler is registered, which will be invoked whenever a connection changes state. The connection change handler will be called once a connection is successfully established. Within the connection change handler, a handler is registered for an entity state change. This second handler will be invoked when new entities are added. (Upon program start, it discovers entities one by one and adds them to the system.) Inside the entity change handler, a third handler is registered that is triggered upon state changes of sensor readings. It is within the sensor change handler that PFTd discovers various sensors available from the BMC and records their internal sensor identification numbers for future reference. Next, the list of requested sensors is validated against the list of those available to report discrepancies. At this point, PFTd registers a final handler for reading actual values from sensors by specifying the identification numbers of the sensors indicated in the configuration

file. Once these values are available, this handler will be called and the PFTd obtains the readings on a periodic basis.

After this lengthy one-time initialization, the PFTd goes into a health monitoring mode by communicating with the BMC. It then starts monitoring the health *via* periodic sampling of values from the given set of sensors before comparing it with the threshold values. In case any of the thresholds is exceeded, control is transferred to the load balancing module of the PFTd. Next, a target node is selected to migrate the guest VM to. The PFTd then contacts Ganglia to determine the least loaded node. The PFTd next issues a migration command that initiates live migration of the guest node from the "unhealthy" node to the identified target node. After the migration is complete, the PFTd raises an alarm to inform the administrator about the change and also logs the sensor values that caused the disruption pending further investigation.

3. EXPERIMENTAL FRAMEWORK

Experiments were conducted on a 16 node cluster. The nodes are equipped with two AMD Opteron-265 processors (each dual core) and 2 GB of memory interconnected by a 1 Gbps Ethernet switch. The Xen 3.0.2-3 Hypervisor/Virtual Machine Monitor is installed on all the nodes. The nodes run a para-virtualized Linux 2.6.16 kernel as a privileged virtual machine on top of the Xen hypervisor. The guest virtual machines are configured to run the same version of the Linux kernel as that of the privileged one. They are constrained within 1 GB of main memory. The disk image for the guest VMs is maintained on a centralized server. These guest VMs can be booted disklessly on the Xen hypervisor using PXE-like netboot via NFS. Hence, each node in the cluster runs a privileged VM and a guest VM. The guest VMs form an MPICH-2 MPD ring on which MPI jobs run. The PFTd runs on the privileged VM and monitors the health of the node using OpenIPMI. The privileged VM also runs Ganglia's gmond daemon. The PFTd will inquire with gmond to determine a target node in case the health of a node deteriorates. The target node is selected based on resource usage considerations (currently only process load). As the selection criteria are extensible, we plan to consult additional metrics in the future (most significantly, the amount of available memory given the demand for memory by Xen guests). In the event of health deterioration being detected, the PFTd will migrate the guest VM onto the identified target node.

We have conducted experiments with several MPI benchmarks executed on the MPD ring over guest VMs. Health deterioration on a node is simulated by running a supplementary daemon on the privileged daemon that migrates the guest VM between the original node and a target node. The supplementary daemon synchronizes migration control with the MPI task executing on the guest VM by utilizing the shared file system (NFS in our case) to indicate progress / completion. To assess the performance of our system, we measure the wallclock time for a benchmark with and without migration. In addition, the overhead during live migration can be attributed to two parts: (1) overhead incurred due to transmitting dirty pages and (2) the actual time for which the guest VM is stopped. To measure the latter, the Xen user tools controlling the so-called "managed" migration [12] are instrumented to record the timings. Thus, the actual downtime for the VM is obtained.

Results were obtained for the NAS parallel benchmarks (NPB) version 3.2.1 [42]. The NPB suite was run on top of the experimental framework described in the previous section. Out of the NPB suite, we obtained results for the BT, CG, EP, LU and SP benchmarks. Class B and Class C data inputs were selected for runs on 4, 8 or 9 and 16 nodes.[3] Other benchmarks in the suite were not suitable, *e.g.*, IS executes for too short a period to properly gauge the effect of imminent node failures while MG required more than 1 GB of memory (the guest memory watermark) for a class C run.

4. EXPERIMENTAL RESULTS

Our experiments focus on various aspects: (a) overheads associated with node failures — single or multiple failures[4], (b) the scalability of the solution (task and problem scaling on migration) and (c) the total time required for migrating a virtual machine. Besides the above performance-related metrics, the correctness of the results was also verified. We noted that in every instance after migration, the benchmarks completed without an error.

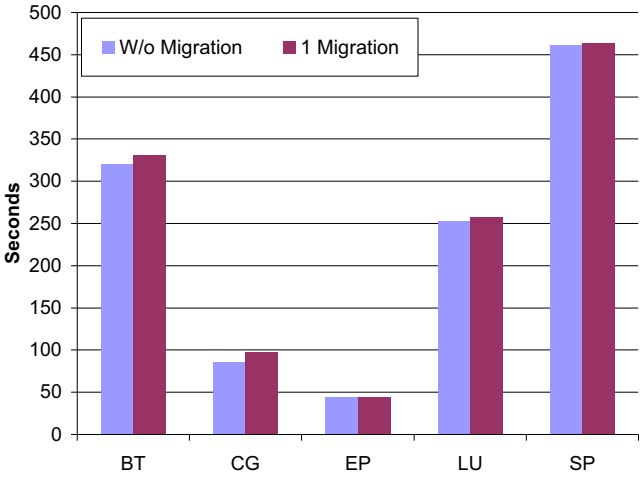

Figure 4: Execution Time for NPB Class C on 16 Nodes (standard deviation for wallclock time was 0-5 seconds — excluding migration — and less than 1 second for migration overhead)

As a base metric for comparison, all the benchmarks were run without migration to assess a base wallclock time (averaged over 10 runs per benchmark). The results obtained from various experiments are discussed in the following.

4.1 Overhead for Single-Node Failure

The first set of experiments aims at estimating the overhead incurred due to one migration (equivalent to one imminent node failure). Using our supplementary PFT daemon, running on the privileged VM, migration is initiated and the wallclock time is recorded for the guest VM including the corresponding MPD ring process on the guest. As depicted in the Figure 4, the wallclock time for execution with migration exceeds that of the base run by 1-4% depending on the application. This overhead can be attributed to the migration overhead itself. The longest execution times of 16-17 minutes were observed for NPB codes BT and SP under Class C inputs for 4 nodes (not depicted here). Projecting these results to even longer-running applications, the overhead of migration can become almost insignificant considering current mean-time-to-failure (MTTF) rates.

4.2 Overhead for Double-Node Failure

In a second set of experiments, we assessed the overhead of two migrations (equivalent to two simultaneous node failures) in

[3]Some NAS benchmarks have 2D, others have 3D layouts for 2^3 or 3^2 nodes, respectively.

[4]We use the term failure in the following interchangeably with imminent failure due to health monitoring.

terms of wallclock time. The migration overhead of single-node and double-node failures over 4 base nodes is depicted in Figure 5. We observe a relatively small overhead of 4-8% over the base wallclock time. While the probability of a second failure of a node decreases exponentially (statistically speaking) when a node had already failed, our results show that even multi-node failures can be handled without much overhead, provided there are enough spare nodes that serve as migration targets.

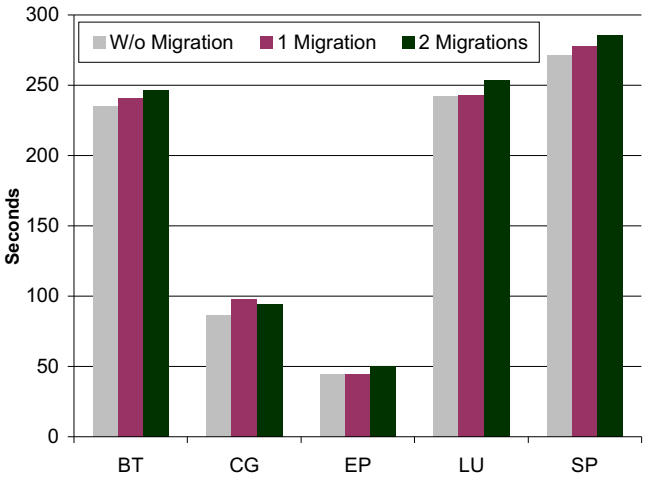

Figure 5: Execution Time for NPB Class B on 4 Nodes

4.3 Effect of Problem Scaling

We ran the NPB suite with class B and C inputs on 16 nodes to study the effect of migration on scaling the problem size (see Figure 6). Since we want to assess the overhead, we depict only the absolute overhead encountered due to migration on top of the base wallclock execution time for the benchmarks. Also, we distinguish the overhead in terms of actual downtime of the virtual machine and other overheads (due transferring modified pages, cache warm-up at the destination, etc.), as discussed in the design section.

The downtime was determined in a ping-pong migration scenario since the timestamps of a migration source nodes and of a target node cannot be compared due to insufficient clock synchronization. Hence, we obtain the start time, s1A, of the stop & copy phase within the first live migration on node A, the finish, f1B, of the first and the start, s2B, of the second stop & copy phase on node B, and the finish time, f2A, of the second migration on node A again. The total downtime per migration is calculated the duration for each of the two downtimes divided by two:

$$downtime = \frac{(f2A - s1A) - (s2B - f1B)}{2}.$$

Since the two timestamps on A and the two timestamps on B are consistent with one another in terms of clock synchronization, we obtain a valid overhead metric at fine time granularity.

Figure 6 shows that, as the task size increases from Class B to Class C, we observe either nearly the same overhead or an increase in overhead (except for SP). This behavior is expected. Problem scaling results in larger data per node. However, the migration mechanism indiscriminately transfers all pages of a guest VM. Hence, problem sizes per se do not necessarily affect migration overhead. Instead, the overhead is affected by the modification rate of pages during live migration. The overhead further depends on whether or not page transfers can be overlapped with application execution and on the moment the migration is initiated. If migration coincides with a global synchronization point (a collective, such as a barrier), the overhead may be smaller compared than that of a migration initiated during a computation-dominated region [28]. SP under class C input appears to experience a migration point around collective communication while memory-intensive writes may dominate for others, such as CG and — to a lesser extent — BT and LU.

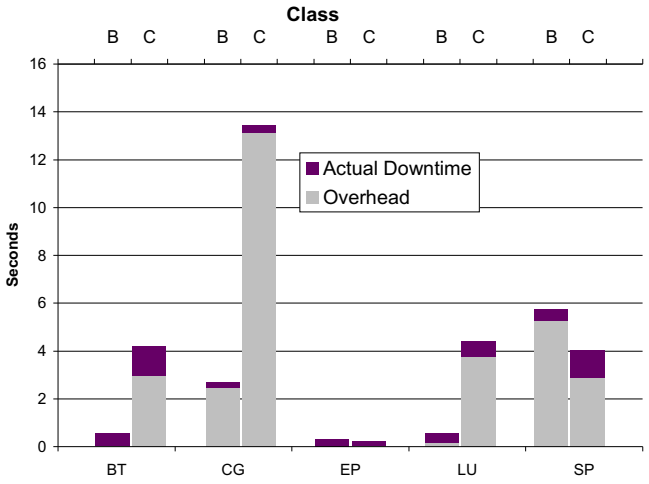

Figure 6: Problem Scaling: Migration Overhead for NPB on 16 Nodes

4.4 Effect of Task Scaling

We next examined the behavior of migration by increasing the number of nodes involved in computation. Figure 7 depicts the overhead for the benchmarks with Class C inputs on varying number of nodes (4, 8/9 and 16).

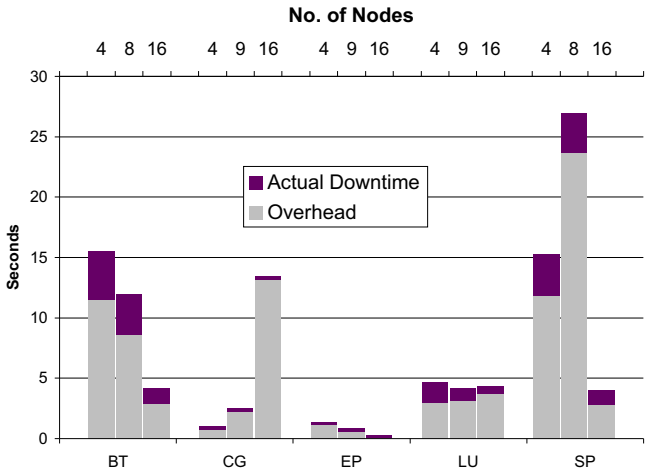

Figure 7: Task Scaling: Migration Overhead for NPB Class C

As with problem scaling, we distinguish actual downtime from other overheads. For most of the benchmarks (BT, EP, LU and SP), we observe a trend of decreasing overheads for increasing number of nodes. Only for CG, we observe an increasing overhead. This can be attributed to additional communication overhead combined with smaller data sets per nodes. This communication overhead adversely affects the time required for migration. These results indicate the potential of our approach for when the number of nodes is increased.

Next, we examine the overall execution time for varying number of nodes. Figure 8 depicts the speedup on 4, 8/9 and 16 nodes normalized to the wallclock time on 4 nodes. The figure also shows the relative speedup observed with and without migration. The lightly colored bars represent the execution time of the benchmarks in the presence of one node failure (and one live migration). The aggregate value of the light and dark stacked bars present the execution time without node failures. Hence, the dark portions of the bars represent the loss in speedup due to migration. The results indicate an increasing potential for scalability of the benchmarks (within the range of available nodes on our cluster) that is not affected by the overhead of live migration.

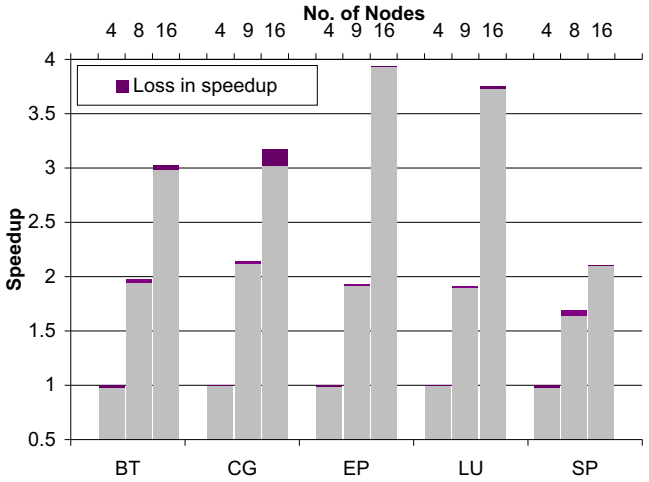

Figure 8: Speedup for NPB Class C

4.5 Cache Warm-up Time

The reported overhead (in previous measurements) includes cache-warm at the migration target. To quantify the cache warm-up effect due to starting the guest VM and then filling the caches with the application's working set, we consider architectural effects. The Opteron processors have 64KB split I+D 2-way associative L1 caches and two 16-way associative 1MB L2 caches, one per core. We designed a microbenchmark to determine the warm-up overhead for the size of the entire L2 cache. Our experiments indicate an approximate cost of 1.6 ms for a complete refill of the L2 cache. Compared to the actual downtime depicted in Figure 6, this warm-up effect is relatively minor compared to the overall restart cost.

4.6 Total Migration Time

We already discussed the overhead incurred due to the migration activity. We next provide insight into the amount of time it takes on the host VM to complete the migration process. On average, 13 seconds are required for relocating a guest virtual machine with 1 GB of RAM that does not execute any applications. Hence, all the migration commands have to be initiated prior to actual failure by at least this minimum bound.

In addition to live migration, Xen provides another way of migration called stop & copy migration. This essentially is the last phase of the live migration, wherein the execution of the VM is stopped and the image is transferred before execution restarts at the destination side. The attractive feature about this mode of migration is that, no matter how data intensive or computation intensive the application, migration takes the same amount of time. In fact, this time is constrained by the amount of memory allocated to a guest VM, which is currently transferred in its entirety so that the cost is mostly constrained by network bandwidth. The memory pages of a process, while it remains inactive, simply cannot be modified during stop & copy. In contrast, live migration requires repeated transfers of dirty pages so that its overhead is a function of the write frequency to memory pages. Our experiments confirm that the stop & copy overhead is nearly identical to the base overhead for relocating the entire memory image of the guest OS. However, the application would be stopped for the above-mentioned period of time. Hence, the completion of the application would be delayed by that period of time.

We have obtained detailed measurements to determine the time required to complete the migration command for the above benchmarks with (a) live and (b) stop & copy migration. These durations were obtained in ping-pong migration experiments similar to the ones for determining the downtime, yet the starting times are when the respective migration is initiated (and not at a later point during migration, as in the earlier downtime measurements).

Figure 9 shows the time taken from initiating migration to actual completion on 16 nodes for the NPB with Class B and C inputs. Live migration duration ranged between 14-24 seconds in comparison to stop & copy with a constant overhead of 13-14 seconds. This overhead includes the 13 seconds required to transfer a 1 GB inactive guest VM.

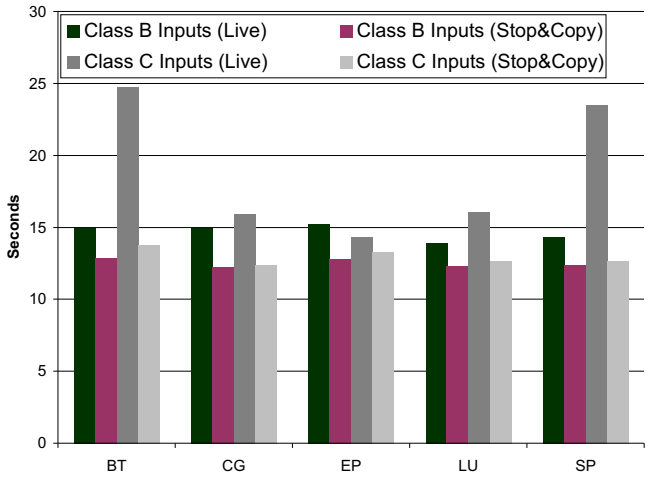

Figure 9: Migration Duration for NPB on 16 Nodes (with a standard deviation of 0.5-3 seconds)

In case of live migration, we observe that the duration for migration increases for BT and SP from Class B to Class C. In contrast, for CG, EP and LU, little variation is observed. In order to investigate this further, we measured the memory usage and also the count of pages transferred during live migration to assess the rate at which pages are modified for 16-node jobs of these benchmarks. The results are depicted in Table 2. We observe an increased memory usage from Class B to Class C for all benchmarks except for EP. Yet, the increase in the number of modified pages, indicated in the last column, shows significant increases for only BT and SP. Thus, the page modification rate has a decisive impact on the migration overhead explaining the more significant overall increases for BT and SP between class B and C under live migration in Figure 9. The results in the Figure also show that, in contrast to live migration, stop & copy migration results in constant time overhead for all the benchmarks.

Figure 10 shows the migration duration for different numbers of nodes for NPB with Class C inputs comparing live and stop & copy migration modes. In case of live migration, for the input-

NPB	Memory Usage in MB		% Increase in Memory Usage	Number of Pages Transferred		% Increase in Pages Transferred
	Class B	Class C		Class B	Class C	
BT	40.81	121.71	198.23	295,030	513,294	73.98
CG	43.88	95.24	117.04	266,530	277,848	4.25
EP	10.61	10.61	0.01	271,492	276,313	1.78
LU	24.15	61.05	152.76	292,070	315,532	8.03
SP	42.54	118.67	178.93	315,225	463,674	47.09

Table 2: Memory Usage, Page Migration Rate on 16 Nodes

sensitive codes BT and SP, we observe a decreasing duration as the number of nodes increases. Other codes experience nearly constant migration overhead irrespective of the number of nodes. In case of stop & copy migration, we note that the duration is constant. These results again assert a potential of our proactive FT approach for scalability within the range of available nodes in the cluster.

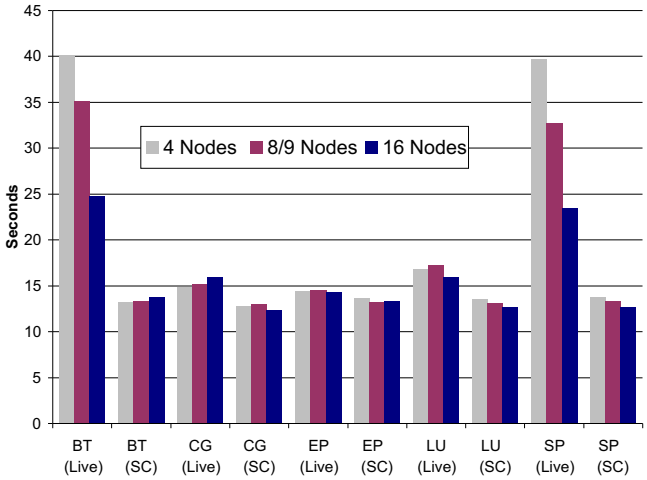

Figure 10: Migration Duration for NPB Class C inputs

While live migration has a higher overhead than the stop & copy approach, the application continues to execute in the former but not in the latter. Hence, we next compare the overall execution time of the benchmarks to assess the trade-off between the two approaches. Figure 11 depicts the overall execution times of the benchmarks with Class B and C inputs on 16 nodes, both for live migration and stop & copy migration with a single node failure.

We observe that live migration results in a lower overall wall-clock execution time compared to stop & copy migration for all the cases (except for nearly identical times for CG under input C). Considering earlier results indicating that the total duration for migration in live approach keeps decreasing as the number of nodes increases (see Figure 10), live migration overall outperforms the stop & copy approach.

Besides the above comparison, the actual migration duration largely depends on the application and the network bandwidth. Migration duration is one of the most relevant metrics for proactive FT. The health monitoring system needs to indicate deteriorating health (*e.g.*, a violated threshold of temperatures or fan speeds) prior to the actual failure of a node. Migration duration provides the metric to bound the minimum alert distance required prior to failure to ensure successful migration completion. Future work is needed in the area of observing the amount of lead time between a detected health deterioration and the actual failure in practice, as past work in this area is sparse [33].

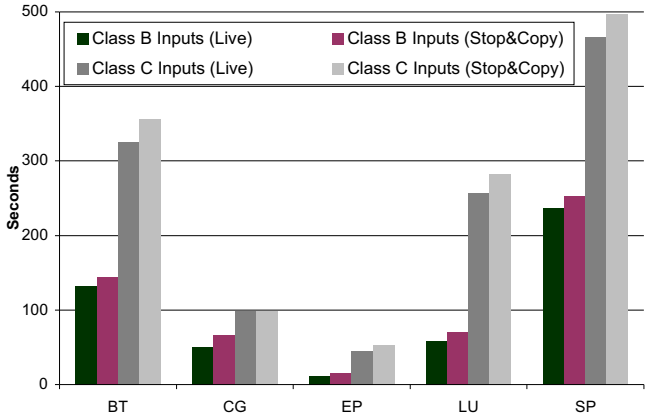

Figure 11: Execution Time for NPB on 16 Nodes

5. RELATED WORK

A number of systems have been developed that combine FT with the message passing implementations of MPI, ranging from automatic methods (checkpoint-based or log-based) [38, 34, 7] to non-automated approaches [4, 16]. Checkpoint-based methods commonly rely on a combination of OS support to checkpoint a process image (*e.g.*, *via* Berkeley Labs Checkpoint Restart (BLCR) Linux module [14]) combined with a coordinated checkpoint negotiation using collective communication among MPI tasks. Another variation to the checkpointing approach is a co-operative checkpointing scheme [28] wherein the checkpoint operation is not performed at a periodic interval. The application instead indicates suitable points for a checkpoint, *e.g.*, at the end of a timestep when data has been consolidated. The runtime/OS then decides to grant or deny the request based on system-wide parameters, *e.g.*, network utilization. Log-based methods generally rely on logging messages and possibly their temporal ordering, where the latter is required for asynchronous approaches. Non-automatic approaches generally involve explicit invocation of checkpoint routines.

Different layers have been utilized to implement these approaches ranging from separate frameworks over the API level to the communication layer or a combination of the two. While higher-level layers are perceived to impose less overhead, lower-level layers encompass a larger amount of state, *e.g.*, open file handles. Virtualization techniques, however, have not been widely used in HPC to tolerate faults, even though they capture even more state (including the entire IP layer). This paper takes this approach and shows that overheads are quite manageable, even in the presence of faults, making virtualization-based FT in HPC a realistic option. LA-MPI [4] operates at a different abstract level, namely that of the network/link layer and, as such, is not designed to transparently provide checkpoint/restart capabilities. It differs in that it provides a complete MPI implementation and transparently hides network errors rather than node failures. FT-MPI [16] is a reactive fault-tolerant solution that keeps the MPI layer and the application alive once a process failure has occurred. This is done by reconfiguring the MPI layer (MPI Communicator) and by letting the application decide how to handle failures. It is the application's responsibility to recover from failures by compensating for lost data/computation within its algorithmic framework, which shifts the burden to the programmer. Compared to potential resynchronization of MPI layer of an entire machine, the restart of lost process and the roll back of all other processes, the performance penalty of our approach is quite minimal.

Virtualization as a technique to tolerate faults in HPC has been

studied before showing that MPI applications run over a Xen virtualization layer [6] result in virtually no overheads [21]. To make virtualization competitive for message-passing environments, OS bypassing is required for the networking layer [26, 25]. This paper leverages Xen as an abstraction to the network layer to provide FT for MPI jobs. It does not exploit OS bypass for networking as this is not an integrated component of Xen. Yet, it does not preclude such extensions without changes to our work in the future. Our FT support leverages the Xen live migration mechanism that, in addition to disk-based checkpointing (and restarting) of an entire guest OS, allows a guest OS to be relocated on another machine [12]. During the lion's share of the migration's duration, the guest OS remains operational while first an initial system snapshot of all pages and then a smaller number of pages (modified since the last snapshot) are transferred. Finally, the guest OS is frozen and last changes are communicated before the new target node is activating the migrated guest OS. This guest OS still uses the same IP number (due to automatic updates of routes at the Xen host level) and is not even aware of its relocation (other than a short lapse of inactivity). We exploit live migration for proactive FT to move MPI tasks from unstable (or unhealthy) nodes to stable (healthy) ones. While the FT extensions to MPI cited above focus on reactive FT, our approach emphasizes proactive FT as a complementary method (at lower cost). Instead of costly recovery after actual failures, proactive FT anticipates faults and migrates MPI tasks onto healthy nodes.

Past work has shown the feasibility of proactive FT [27]. More recent work promotes FT in Adaptive MPI using a combination of (a) object virtualization techniques to migrate tasks and (b) causal message logging within the MPI runtime system of Charm++ applications [9, 10, 11]. Causal message logging is due to Elnozahy *et al.* [15]. Our work focuses on assessing the overhead of Xen-based proactive FT for MPI jobs. It contributes an integrated approach to combine health-based monitoring with OpenIPMI [2] to predict node failures and proactively migrate MPI jobs to healthy nodes. In contrast to the Charm++ approach, it is coarser grained as FT is provided at the level of the entire OS, thereby encapsulating one or more MPI tasks and also capturing OS resources used by applications, which are beyond the MPI runtime layer.

FT support at different different levels has different merits due to associated costs. Process-level migration [31, 39, 23, 5, 13, 14] may be slightly less expensive than virtualization support. Yet, the former may only be applicable to HPC codes if certain resources do not need to be captured that virtualization covers — at the cost of increased memory utilization due to host and guest OS consumption for virtualization. A system could well support different FT options to let the application choose which one best fits its code and cost constraints.

While integrated with Xen's live migration, our solution is, in it's methodology, equally applicable to other virtualization techniques, such as live migration strategies implemented in VMWare's VMotion or NomadBIOS [18], a solution closely related to Xen's live migration, which is implemented over the L4 microkernel [19]. Even non-live migration strategies under virtualization [35, 24, 41, 29] could be integrated but would be less effective due to their stop & copy semantics. Demand-based migration [43], however, is unsuitable in a proactive environment as it does not tightly bound the migration duration.

6. CONCLUSION

Node failures on contemporary computers can often be anticipated by monitoring health and detecting a deteriorating status. To exploit anticipatory failures, we are promoting proactive fault tolerance (FT). Instead of a reactive scheme proactive FT system, processes automatically migrate from "unhealthy" nodes to healthy ones. This is in contrast to a reactive scheme where recovery occurs in response to already occurred failures.

We have contributed an automatic and transparent mechanism for proactive FT for arbitrary MPI applications. Combining virtualization techniques with health monitoring and load-based migration, we assess the viability of proactive FT for contemporary HPC clusters. Xen's live migration allows a guest OS to be relocated to another node, including running tasks of an MPI job. We exploit this feature when a health-deteriorating node is identified, which allows computation to proceed on a healthy node, thereby avoiding a complete restart necessitated by node failures. The live migration mechanism allows execution of the MPI task to progress while being relocated, which reduces the migration overhead for HPC codes with large memory footprints that have to be transferred over the network. Our proactive FT daemon orchestrates the tasks of health monitoring, load determination and initiation of guest OS migration. Experimental results confirm that live migration hides the costs of relocating the guest OS with its MPI task. The actual overhead varies between 1-16 seconds for most NBP codes. We also observe migration overhead to be scalable (independent of the number of nodes) within the limits of our test bed. Our work shows that proactive FT complements reactive schemes for long-running MPI jobs. As proactive FT has the potential to prolong the mean-time-to-failure, reactive schemes can lower their checkpoint frequency in response.

7. REFERENCES

[1] Ganglia. http://ganglia.sourceforge.net/.

[2] OpenIPMI. http://openipmi.sourceforge.net/.

[3] Advanced configuration & power interface. http://www.acpi.info/, 2004.

[4] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall, and M. W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *International Parallel and Distributed Processing Symposium*, 2004.

[5] A. Barak and R. Wheeler. MOSIX: An integrated multiprocessor UNIX. In USENIX Association, editor, *Proceedings of the Winter 1989 USENIX Conference: January 30–February 3, 1989, San Diego, California, USA*, pages 101–112, Berkeley, CA, USA, Winter 1989. USENIX.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Symposium on Operating Systems Principles*, pages 164–177, 2003.

[7] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, Nov. 2002.

[8] R. Butler, W. Gropp, and E. L. Lusk. A scalable process-management environment for parallel programs. In *Euro PVM/MPI*, pages 168–175, 2000.

[9] S. Chakravorty, C. Mendes, and L. Kale. Proactive fault tolerance in large systems. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[10] S. Chakravorty, C. Mendes, and L. Kale. Proactive fault tolerance in mpi applications via task migration. In *International Conference on High Performance Computing*, 2006.

[11] S. Chakravorty, C. Mendes, and L. Kale. A fault tolerance protocol with fast fault recovery. In *International Parallel and Distributed Processing Symposium*, 2007.

[12] C. Clark, K. Fraser, S. Hand, J. Hansem, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation*, May 2005.

[13] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Softw., Pract. Exper.*, 21(8):757–785, 1991.

[14] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Tr, Lawrence Berkeley National Laboratory, 2000.

[15] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, 1992.

[16] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting, Lecture Notes in Computer Science*, volume 1908, pages 346–353, 2000.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[18] J. G. Hansen and E. Jul. Self-migration of operating systems. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, page 23, New York, NY, USA, 2004. ACM Press.

[19] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-Kernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 66–77, New York, Oct. 1997. ACM Press.

[20] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[21] W. Huang, J. Liu, B. Abali, and D. Panda. A case for high performance computing with virtual machines. In *International Conference on Supercomputing*, June 2006.

[22] IBM T.J. Watson. Personal communications. Ruud Haring, July 2005.

[23] E. Jul, H. M. Levy, N. C. Hutchinson, and A. P. Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

[24] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 40–, 2002.

[25] J. Liu, W. Huang, B. Abali, and D. Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Conference*, June 2006.

[26] A. Menon, A. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. In *USENIX Conference*, June 2006.

[27] A. Oliner, R. Sahoo, J. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for bluegene/l systems. In *International Parallel and Distributed Processing Symposium*, 2004.

[28] A. J. Oliner, L. Rudolph, and R. K. Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In *International Conference on Supercomputing*, pages 14–23, 2006.

[29] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *OSDI*, 2002.

[30] I. Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[31] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Symposium on Operating Systems Principles*, pages 110–119, Oct. 1983.

[32] S. Rani, C. Leangsuksun, A. Tikotekar, V. Rampure, and S. Scott. Toward efficient failre detection and recovery in hpc. In *High Availability and Performance Computing Workshop*, page (accepted), 2006.

[33] R. Sahoo, A. Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–435, 2003.

[34] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Oct. 2003.

[35] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *OSDI*, 2002.

[36] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 249–258, 2006.

[37] H. Song, C. Leangsuksun, and R. Nassar. Availability modeling and analysis on high performance cluster computing systems. In *First International Conference on Availability, Reliability and Security*, pages 305–313, 2006.

[38] G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[39] M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the v-system. In *SOSP*, pages 2–12, 1985.

[40] C. Wang, F. Mueller, C. Engelmann, and S. Scott. A job pause service under lam/mpi+blcr for transparent fault tolerance. In *International Parallel and Distributed Processing Symposium*, page (accepted), Apr. 2007.

[41] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *Symposium on Networked Systems Design and Implementation*, pages 169–182, 2004.

[42] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Supercomputing*, 1999.

[43] E. R. Zayas. Attacking the process migration bottleneck. In *SOSP*, pages 13–24, 1987.