Proactive Fault Tolerance for HPC with Xen Virtualization [*]

Arun Babu Nagarajan, Frank Mueller

Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
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 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. Furthermore, migration overhead is shown to be independent of the number of nodes in our experiments indicating the potential for scalability of our approach. Overall, our enhancements make proactive FT a valuable asset for long-running MPI application, particularly as a complementary scheme to reactive FT using full checkpoint/restart schemes. 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.*

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 [18]. When one node fails, a 1024-processor midplane had to be temporarily shut down to replace the card. A study by Los Alamos National Laboratory estimates the mean time between failure (MTBF), extrapolating from current system performance [25], to be 1.25 hours on a petaflop machine.

Current techniques to tolerate faults focus on reactive schemes where fault recovery commonly relies on a checkpoint/restart (C/R) mechanism. However, the Los Alamos study [25] 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.

Yet, 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 [31, 27]. 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 fault tolerance (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

[*]This work was supported in part by NSF grants CCR-0237570(CAREER), CNS-0410203, CCF-0429653 and DOE DE-FG02-05ER25664.

the exception. The feasibility of health monitoring at various levels has recently been demonstrated for temperature-aware monitoring, *e.g.*, by using ACPI [2], and more generically, by critical-event prediction [28]. Particularly in systems with thousands of processor, 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 [8, 9, 10, 23]. These and other approaches are discussed in more detail in the related work. They differ from our approach in that we exploit OS-level virtualization combined with health monitoring and live migration.

We have designed and implemented an automatic and transparent mechanism for proactive FT of arbitrary MPI applications over Xen [5]. 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 [11]. 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. We further show migration overhead to be independent of the number of nodes in a system. Hence, live migration provides a scalable solution to realize FT. 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*, we believe that this is the first comprehensive study of proactive fault tolerance where live migration is actually triggered by health monitoring.

The paper is structures 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 benchmarks. 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 at least two functions — proactive decision making and load balancing (which in turn provides fault tolerance). An overview of the system components and their interaction is depicted in Figure 1. Next, we describe how each of these components of our system.

2.1 Fault Tolerance over Xen

To provide an effective fault tolerance system, we need a mechanism 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 [11]. 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 starts sending the pages of the guest VM to the destination node in a first iteration of the so-called pre-copy step. 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" page is logged so that it can later be identified.

3. The host VM now starts sending these logged pages iteratively in chunks during subsequent iterations on the pre-copy step. Repeated page differences are sent till a heuristic indicates that this diff process 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) [11], 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 and copy, commitment and activation steps.

The actual downtime due to the last phase has been reported to be as low as 60 ms [11]. 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 two seconds for some experiments, which shows that HPC codes may have higher rates of page modifications. The overall overhead contributed to the total wall-clock time of the application on the migrating guest VM can be attributed to this actual downtime plus the overhead associated with the active page-difference operation during migration. Experiments show that this overhead is negligible compared to that of the total wall-clock time for HPC codes.

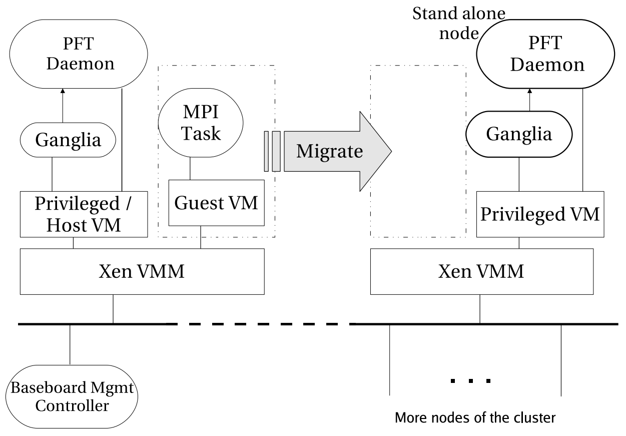

Figure 1. Overall setup of the components

2.2 Heath 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 node failures in our case, a health monitoring mechanisms 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 to monitor and manage hardware, a task performed in the past by software with proprietary interfaces.[1] The Baseboard Management Controller (BMC) 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 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-based mechanism allowing one to specify an event (*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-based mechanism might offload some overhead from the application side since the BMC takes care of notifying back when an event occurs. 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, we need to select a target node to migrate the virtual machine to. We utilize Ganglia, 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. We need to know 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 run a guest virtual machine and has the lowest load

[1]Alternatives to IPMI exist, such as lm_sensor, but they tend to be system-specific (x86 Linux) and may be less powerful. Also, disk monitoring can be realized portably with SMART.

based on CPU usage. We can further extend this functionality to check if the selected target node has enough unused 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

Before explaining the design of the Proactive Fault Tolerance PFT daemon (PFTd), let us explain the way each node in the cluster is set up (see Figure 1). First, Xen Virtual Machine Monitor (VMM) is installed. On top of the VM runs a privileged/host virtual machine. In addition, a guest virtual machine runs on top of the Xen VMM. The privileged virtual machine hosts, among others, a daemon for Ganglia, which aids in selecting the target node for migration. The guest virtual machines form a multi-purpose daemon (MPD) ring [7] 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.

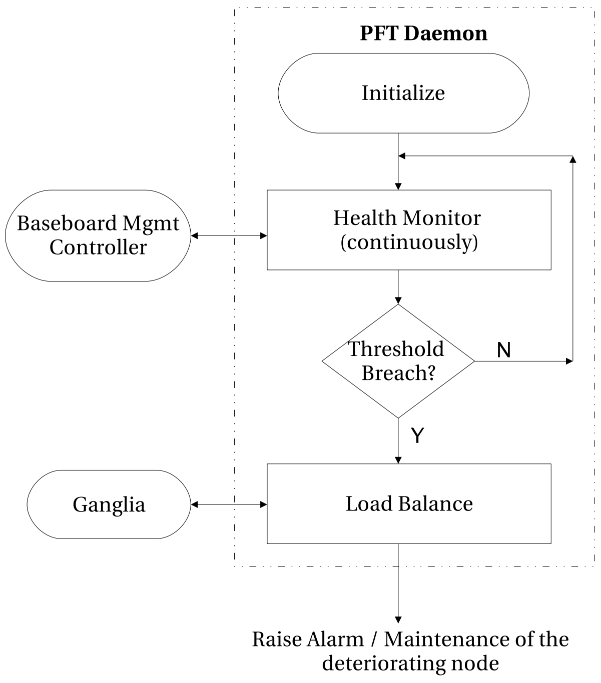
PFT Daemon
Initialize
Baseboard Mgmt Controller
Health Monitor (continuously)
Threshold Breach?
N
Y
Ganglia
Load Balance
Raise Alarm / Maintenance of the deteriorating node

Figure 2. Proactive Fault Tolerance Daemon

Next, the design of the proactive fault tolerance daemon (PFTd) is detailed. In the above setup, 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 primarily three components: Health monitoring, decision making and load balancing (see Figure 2). 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.

Initially, when PFTd starts up, it reads a configuration file containing a list of parameters to be monitored. In addition to a parameter name, the lower and upper thresholds for that particular parameter can also be specified. For example, since we have dual processor machines, we specify the safe temperature range for two CPUs and the valid speed range for system fans. Next, 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 needs to be 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 set up. Within the connection change handler, a handler is registered for an entity state change. This second handler will be invoked when a 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 for catching 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 configuration file. Once these values are available, this handler will be called and 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 are 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 obtain the least loaded node. After target node is identified, the PFTd issues a migration command, which initiates live migration of the guest node from the "unhealthy" node to the identified target node. After the migration is complete, PFTd can raise an alarm to inform the administrator about the change and also log the sensor values which caused the disruption for 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 diskless-booted 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 Proactive Fault Tolerance Daemon (PFTd) runs on the privileged VM and monitors the health of the node using OpenIPMI. The privileged VMs 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 by running several benchmarks on the MPD ring over guest VMs. Health deterioration on a node is simulated by running a supplementary daemon on the privileged daemon, which migrates the guest VM between the original node and a target node. The supplementary daemon synchronizes migration control with the MPI task running 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 wall-clock 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 diff operations on the pages and (2) the actual time for which the guest VM is stopped. To measure the latter, the Xen user tools controlling so-called "managed" migration [11] are instrumented to record the timings and, hence, the actual downtime for the VM is measured.

Results were obtained for the NAS parallel benchmarks (NPB) version 3.2.1. 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 (depending on input requirements) and 16 nodes. Other benchmarks in the suite were not suitable, *e.g.*, IS executes for too short a period to properly gauge the effect of immanent node failures while MG required more than 1 GB of memory (the guest memory watermark) for a class C run.

4 Experimental Results

As a base metric for comparison, all the benchmarks were run without migration to assess a base wall-clock time (averaged over 10 runs per benchmark). Also, the results obtained with migration are verified for correctness. The benchmarks completed without error in every instance after migration. The experiments are organized here in three areas. One focuses on the overheads associated with node failures. (We use the term failure in the following interchangeably with immanent failure due to health monitoring.) The second one assesses the scalability of the solution. The third measures the total migration time.

We also conducted a series of experiments to measure the overhead associated with single/double node failures and to observe the behavior of task and problem scaling on migration. The results obtained are explained in detail below.

4.1 Overhead for Single-Node Failure

The first set of experiments aims at estimating the overhead incurred in one migration (equivalent to one immanent node failure).

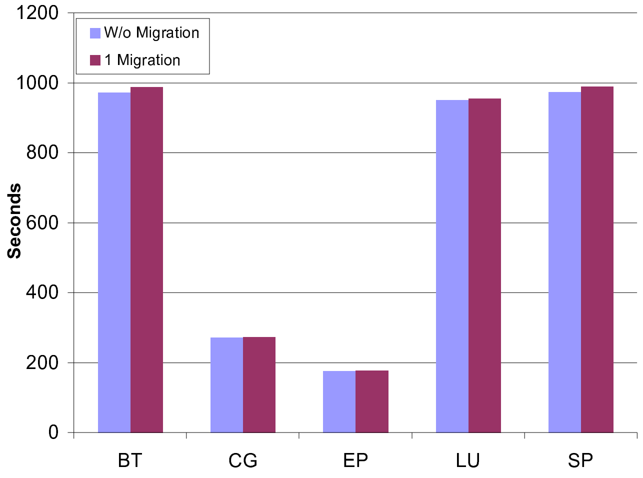

Figure 3. NPB Class C with 4 nodes

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 3, the wall-clock time for execution with migration exceeds than that of the base run by 1-4% depending on the application. This overhead can be attributes to the migration overhead. The NPB codes BT and SP ran the longest for Class C at 16-17 minutes for 4 nodes. 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 terms of wall-clock time. Again, we observe a relatively small overhead of 4-8% over the base wall-clock time, as depicted in the Figure 4. Even though 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.

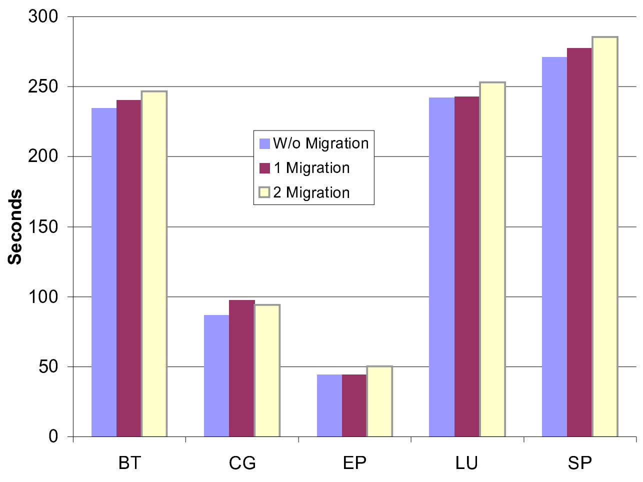

Figure 4. NPB Class B over 4 nodes

4.3 Effect of Problem Scaling

We ran the NPB suite with class B and C inputs on the same number of nodes (4 nodes) to study the effect of migration on scaling the task size per node. Since we are concerned only about the overhead in addition to the base wall-clock execution time for the benchmarks, we plot only the absolute overhead encountered due to migration. Also, we distinguish the overhead in terms of actual downtime of the virtual machine and other overheads (due to the page difference operation, cache warm-up at the destination, etc.), as discussed in the design section. Figure 5 shows that in all cases (except for CG), as the task size increases from Class B to Class C, we observe an increase in overhead.

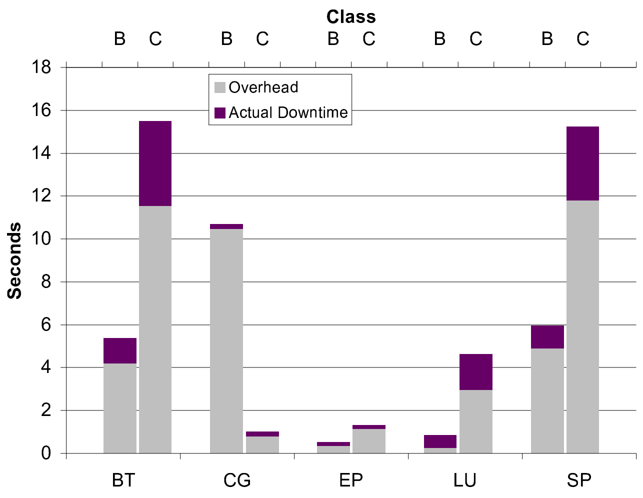

Figure 5. Migration Overhead for 4 Nodes

This behavior is somewhat expected. Problem scaling leads to an increased amount of modified pages while migration is in progress. This, in turn, increases the overhead of repeated transfers of modified pages. We also observe an anomaly in CG that may be due to a number of reasons. The exact overhead associated with an application entirely depends on the moment the migration is initiated. If migration coincides with a global synchronization point (a collective, such as a barrier), we expect the overhead to be smaller compared than that of a migration initiated during a computation-dominated region.

4.4 Effect of Task Scaling

We next examined the behavior of the migration by increasing the number of nodes involved in computation. Since our focus is on the overhead, we have depicted only the overhead observed in all the cases. We ran benchmarks with Class C inputs on varying number of nodes (4, 8/9 and 16). The results are shown in the Figure 6. As in task scaling, we distinguish actual downtime from other overheads. When increasing the number of nodes from 4 to 8, the overhead of BT, EP and LU actually decreases. Conversely, the remaining codes show increasing overhead. From 8 to 16 nodes, the overhead also increases for all benchmarks except for SP. 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. The 16-node overhead for BT and LU at 60 and 50 seconds, respectively, is only explained in part by additional communication overhead. In fact, measurements show that only 32 and 15 seconds for BT and LU are accounted for by migration duration exhibited by the Xen migration directive. Some of the remaining cost may be due to activation on the target node, cache warm-up and TLB misses, a direction that is currently under investigation.

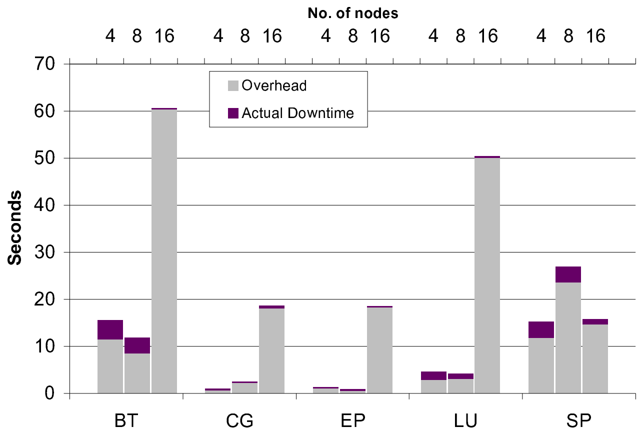

Figure 6. Migration Overhead, Class C Inputs

4.5 Scalability (Total Execution Time)

Since these experiments were conducted on 4, 8/9 and 16 nodes, the results provide initial insight to the scalability of the design. Figure 7 depicts the speedup on 8/9 and 16 nodes with respect to the wall-clock time on 4 nodes. The figure also shows the relative speedup observed with and without migration. The lightly colored region of the bars represent the normalized execution time of the benchmarks with one node failure. The aggregate value of the lightly and the dark-colored portions of the bars represent the execution time normalized to the equivalent runtime without node failures. Hence, the dark-colored regions of the bars represent the loss in speedup due to migration. As we see from the figure, the speedup with migration is close to that achieved without migration.

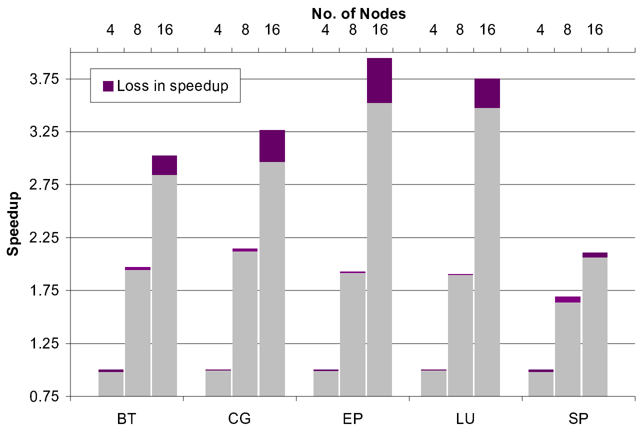

Figure 7. NPB Class C with 4, 8/9, 16 nodes

4.6 Cache Warm-up Time

The reported overhead includes cache-warm at the migration target. We tried to quantify the cache warm-up effect due to starting the guest VM and then filling the caches with the application's working set. 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 5, this warm-up effect is relatively minor compared to the overall restart cost.

4.7 Total Migration Time

We already discussed the overhead incurred due to the migration activity in detail. We now give an insight into the amount of time it takes on the host VM to complete the migration process. On average, 13-14 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.

We also obtained detailed timing information during the experiments to determine the time required to complete the migration command for the above benchmarks. Migration duration ranged between 14-40 seconds. This overhead includes a minimum of 13 seconds to transfer a 1 GB inactive guest VM. Figure 8 shows the time taken from initiating migration to actual completion on 4 nodes for the NPB with Class B and C inputs. Due to the increased number of modified pages from class B to class C, the time taken for migration increases for BT and SP. For CG, EP and LU, in contrast, little variation is observed.

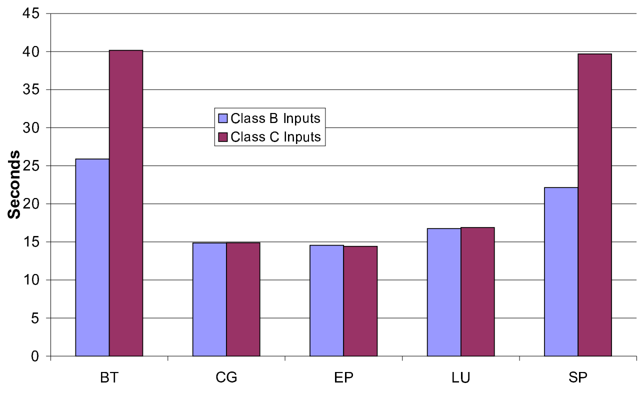

Figure 8. Migration Duration for 4 Nodes

Figure 9 shows the migration duration for different numbers of nodes for NPB with Class C inputs. For the input-sensitive code BT and SP, we observe a decreasing duration as the number of nodes increases. Other codes experience nearly constant migration overhead independent of the number of nodes. This again asserts the scalability of the solution.

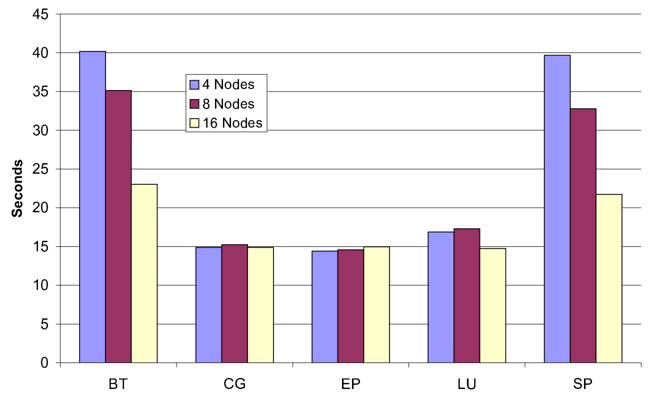

Figure 9. Migration Duration for 4/8/16 Nodes

The actual migration duration largely depends on the application and the network bandwidth. Migration duration is a relevant metric for proactive FT. The health monitoring system needs to indicate deteriorating health through a violated threshold 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 time given between a detected health deterioration and the actual failure in practice. We are not aware of any work to this extent.

5 Related Work

A number of systems have been developed that combine FT with the message passing implementing MPI, ranging from automatic methods (checkpoint-based or log-based) [32, 29, 6] to non-automated approaches [3, 14]. 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 [13]) combined with a coordinated checkpoint negotiation using collective communication among MPI tasks. 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.

Virtualization as a technique to tolerate faults in HPC has been studied before showing that MPI applications run over a Xen virtualization layer [5] result in virtually no overheads [17]. To make virtualization competitive for message-passing environments, OS bypassing is required for the networking layer [22, 21]. 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 it 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 [11]. During the lion's share of the migration's duration, the guest OS remains operational while first an initial system snapshot and then a smaller amounts of state (modified since the last snapshot) are transferred. Finally the guest OS is frozen and final 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.

Proactive FT is a scheme to move computation away from resources in anticipation of imminent faults. Past work has shown the feasibility of proactive FT [23]. 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 [8, 9, 10]. Causal message logging is due to Elnozahy *et al.* [**?**]. 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 [1] 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 [26, 33, 19, 4, 12, 13] 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 it's 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 [15], a solution closely related to Xen's live migration, which is implemented over the L4 microkernel [16]. Even non-live migration strategies under virtualization [30, 20, 34, 24] could be integrated but would be less effective due to their stop-and-copy semantics. Demand-based migration [35], 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 one and sixteen seconds for the NBP codes. We also observe migration overhead to be scalable (independent of the number of nodes) in 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.

References

[1] Openipmi. http://openipmi.sourceforge.net/.

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

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

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

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

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

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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*, Sante Fe, New Mexico, USA, October 2003.

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

[31] 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.

[32] 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.

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

[34] 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.

[35] E. R. Zayas. Attacking the process migration bottle-neck. In *SOSP*, pages 13–24, 1987.