## ABSTRACT

NAGARAJAN, ARUN BABU. System Virtualization for Proactive Fault-Tolerant Computing. (Under the direction of Associate Professor Dr. Frank Mueller).

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 thesis 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.

System Virtualization for Proactive Fault-Tolerant Computing

by
Arun Babu Nagarajan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, NC

2008

APPROVED BY:

| | |
|---|---|
| Dr. Xiaosong Ma | Dr. Xiaohui (Helen) Gu |

Dr. Frank Mueller
Chair of Advisory Committee

# DEDICATION

To my parents: Thank you for everything.

# BIOGRAPHY

Arun Babu Nagarajan was born on April 27, 1982. He received his Bachelor of Engineering in Computer Science from Madurai Kamaraj University, Madurai, India in May 2003. With the defense of this thesis he will receive his Master of Science in Computer Science from North Carolina State University in August 2008.

# ACKNOWLEDGMENTS

I would like to acknowledge the following people for their support towards the completion of this thesis: My adviser Dr. Frank Mueller and my thesis committee - Dr. Xiaosong Ma and Dr. Helen Gu. I would also like to thank my sister, all my friends, folks at the systems lab and roommates for their support and encouragement.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

# Introduction

## 1.1 Background

High-end parallel computing is increasingly relying upon large clusters with thousands of processors. At such large count of processors, faults are becoming commonplace. Reactive fault tolerance is traditionally employed to recover from such faults. In this thesis, we are promoting a proactive fault tolerant scheme to complement the reactive schemes. Let us first introduce common terminology for our work.

The objective of *high performance computing (HPC)* is to solve numerical problems from the scientific domain by using a large collection of computers/processors connected with each other using interconnects that work cooperatively with specific communication methodologies to solve a computational problem.

A *cluster* is a collection of compute nodes (PCs or workstations) that are generally identical in configuration and interconnected with a dedicated network. They serve as a platform for multiprogramming workloads. Furthermore, clusters generally run an identical version of the operating system and feature a shared file system [14]. *Single Program Multiple Data (SPMD)* is a parallel programming model wherein the same program is executed on all the nodes of a parallel system(cluster) whereas the nodes operate on different data [20]. SPMD is suitable for both shared memory and message passing environments. This thesis focuses on message passing environments typically manifested by clusters.

*Message Passing Interface (MPI)* is a standard specification for the interface of a message-passing library for writing parallel programs [19]. The extensive set of interfaces provided by MPI helps in writing efficient SPMD programs with minimal effort. MPI sup-

ports point-to-point communication, collective communication and synchronization among others. HPC codes are typically written as SPMD programs with MPI and are designed to run on cluster-based environments.

The shift towards parallel computing in general is driven mainly by hardware limitations [20]. From the view of computer design, clock frequencies are increasing slowly having reached their physical limits. In addition, the difference in speed of operation of processor and the memory is continuously increasing. This property popularly known as "memory wall", widens the bottleneck and also contributes to the shift. This has not only affected the high performance computing (HPC) environments, but the effect is well observed even in general-purpose computing as manufacturers have moved from uni-processors to the now ubiquitous multi-core processors over the past few years. More importantly, the trend in HPC has been to rely more upon a large collection of processors.

## 1.2 Necessity of Fault Tolerance

As seen above, faults are common in high performance computing enviroments with thousands of processors. 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 [29]. When one node fails, a 1024-processor mid-plane had to be temporarily shut down to replace the card.

Results from related work [26], depicted in Table 1.1, 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 [46]. Another study conducted by LANL estimates the MTBF, extrapolating from current system performance [40], to be 1.25 hours on a petaflop machine.

Commercial installations, such as Google (see Table 1.1) experience an interpolated fault rate of just over one hour for equivalent number of nodes, yet their fault-tolerant

Table 1.1: Reliability of HPC Clusters

| 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 |

middleware hides such failures altogether so that user services remain completely intact [21]. In this spirit, our work focuses on fault-tolerant middleware for HPC systems.

## 1.3 Traditional Approach

Current techniques to tolerate faults focus on reactive schemes where fault recovery commonly relies on a checkpoint/restart (C/R) mechanism.

The mode of operation is to allow the fault to happen and recover from the fault. A typical C/R system checkpoints the application's progress over its entire runtime by recording the state during every time delta on all the nodes and saving it to a global file system. In case of a node failure, the state of the application can be restored by restarting the whole system with the latest available checkpoint. The frequency of checkpoints affects the performance on the system: If the checkpoint is very frequent, the checkpoint overhead is increased but, on a failure, the work to repeat lost computation is reduced. On a contrary, if the checkpoint is less frequent, the overhead is reduced but, on a failure, the work to repeat lost computation is increased. Since C/R does not contribute to useful computation, there is a considerable overhead on the system. In fact, the LANL study [40] 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. Clearly, a mechanism which aids in reducing the checkpointing overhead is necessary.

## 1.4 Motivation

### 1.4.1 Fault Detection Feasibility

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 [47, 42]. 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 [4], and, more generically, by critical-event prediction [43]. 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 [10, 11, 12, 37]. 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.

### 1.4.2 Minimal Overhead of OS Virtualization

This thesis 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 [53] using Class C inputs over Xen [7]. 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 Chapter 3 for configuration details). The results in Figure 1.1 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

[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.

have demonstrated [34, 33]. 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.

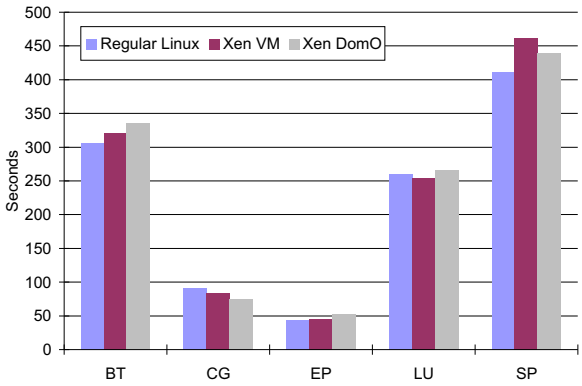


Figure 1.1: Xen Overhead for NAS PB, Class C, 16 Nodes

## 1.5 Thesis Statement

The objective of this thesis is to develop a proactive fault tolerance scheme for HPC environments. We explore solutions addresssing the following questions in this thesis.

- Can transparent and automatic fault tolerance be achieved in a proactive manner for a cluster-based HPC environment running arbitrary MPI applications?
- Is virtualization a feasible (efficient and supportive) option for providing FT?
- Can proactive FT complement reactive fault tolerance?

## 1.6 Overview of our approach

We have designed and implemented a proactive FT system for HPC over Xen [7], of which we provide a brief overview here. 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 [13]. 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 can revert 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. An earlier version of this work was published at International Conference on Supercomputing, June 2007 [36].

## 1.7 Thesis Layout

This thesis is structured as follows. Chapter 2 presents the design and implementation of our health monitoring and migration system with its different components. Chapter 3 describes the experimental setup. Chapter 4 discusses experimental results for a set of benchmarks. Chapter 5 contrasts this work to prior research. Chapter 6 summarizes the contributions.

# Chapter 2

# System Design and Implementation

A proactive fault tolerance system, as the name implies, should provide two mechanisms - one for proactive decision making and another for addressing load balancing, which, in combination, provide fault tolerance. The various components which are involved in the design and the way in which they act together to achieve the goal of proactive fault tolerance are explained in the following sections.

## 2.1 Setup of the components

An overview of the various components in the system and their interaction is depicted in Figure 2.1. The components depicted in the figure are discussed below:

1. Each node in the cluster hosts an instance of the Xen Virtual Machine Monitor (VMM) or Xen Hypervisor. The Hypervisor works at the highest privilege level than the operating systems which run on it.

2. On top of the VMM runs a privileged/host virtual machine, which is a para-virtualized Linux version in our case. It is called 'privileged' because only this virtual machine is provided with some capabilities *via* a set of tools to manage the other virtual machines running on the hypervisor - e.g, start / suspend / shutdown / migrate the virtual machines.

3. The privileged virtual machine hosts the following among others.

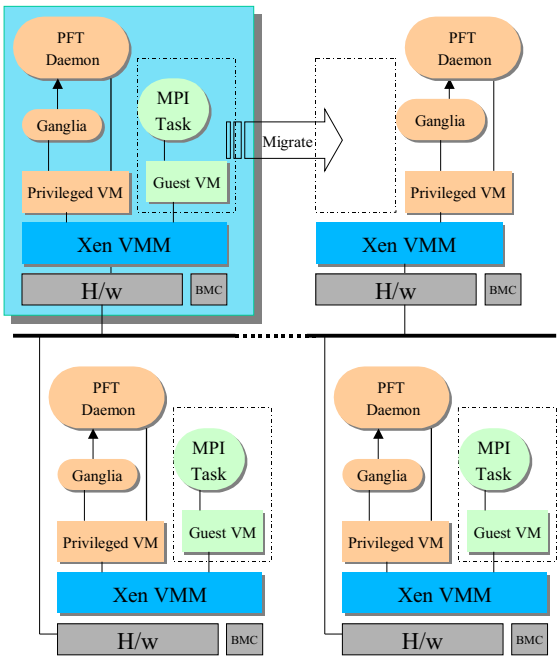


Figure 2.1: Overall setup of the components

- A daemon for Ganglia, which aids in selecting the target node for migration - explained in Section: 2.4.
- Proactive FT daemon (PFTd) used to monitor health and initiate migration - explained in Section 2.5.
- A standard daemon called Xend or Xen daemon which aids in administration of the other virtual machines.

4. A guest virtual machine (also the same version of Linux) runs on top of the Xen VMM. One or more of them can be running on the Hypervisor. But for our purpose we have used only one of them on a node.

5. The guest virtual machines form a multi-purpose daemon (MPD) ring of all cluster nodes [9] 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.

6. There is another hardware component available with each of the nodes in the cluster called the BMC (Baseboard Management Controller), which provides the functionality of monitoring the hardware using sensors. The BMC is explained in more detail in

Section 2.3.

7. Apart from the above mentioned components, we also have some spare node(s) with all the above components except for the guest VM running on it. Availability of spare nodes was discussed in the Section 1.4.1.

Upon deteriorating health of a compute node, determined through the monitoring capabilities of BMC and a spare/weakly loaded node identified by the PFTd through Ganglia, the entire guest VM can be migrated to the identified node. We will describe each of the above components of our system in the following sections in detail.

## 2.2 Fault Tolerance over Xen

To provide an efficient fault tolerance system certain qualities are desired:

1. **Transparency:** The MPI task does not need to be aware of the fault tolerance activity that happens in response to an anticipated fault.

2. **Relocation:** A mechanism is needed that would gracefully aid the relocation of an MPI task, thereby enabling it to run on a different physical node with minimum possible overhead.

Xen proves to be an apt candidate for us since it provides the above two qualities which are discussed below:

### 2.2.1 Transparency

Xen is a Virtual Machine Monitor (VMM) that provides a way for multiple operating systems to run over it, with proper isolation between guest operating systems [7]. Xen is a *para-virtualized* environment requiring that the hosted virtual machine be adapted/modified to run on the Xen virtual machine monitor (VMM). However, the application binary interface(ABI) needs no modifications. Because of para-virtualization, Xen has an improved performance benefits compared to a fully-virtualized environment [7]. Xen provides a complete transparency not even at the level of applications, but to the whole Operating System with virtualization. One of the virtual machines which run on the VMM is called 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.

### 2.2.2 Relocation

Xen provides the functionality of *migration*, which enables the guest VM to be transferred from one physical node to another. 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.

Xen supports two types of migration - *Live* migration and *Stop and Copy* migration. The mechanism behind these two techniques are discussed here.

**Mechanism of Xen Live Migration:**

A Live migration [13] refers to the virtual machine being in operation while the migration is performed except for a very short period of time, during which the machine is actually stopped. Live migration occurs as a sequence of the following 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) [13], 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 [13]. 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 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 phase when dirty pages are transferred during migration. Experiments show that this overhead is negligible compared to that of the total wall-clock time for HPC codes.

**Mechanism of Xen Stop & Copy Migration:**

Stop & Copy migration is simple compared to live migration. It also is the last phase of the live migration. The sequence of events that happen are:

1. Once the migration command is initiated, the execution of the VM is stopped.
2. The image of the VM is transferred to the destination with all the state information needed to resume execution on the other side.
3. The execution restarts at the destination from the received data.

The live and stop & copy migrations are contrasted while discussing the experimental results in Section 4.6.

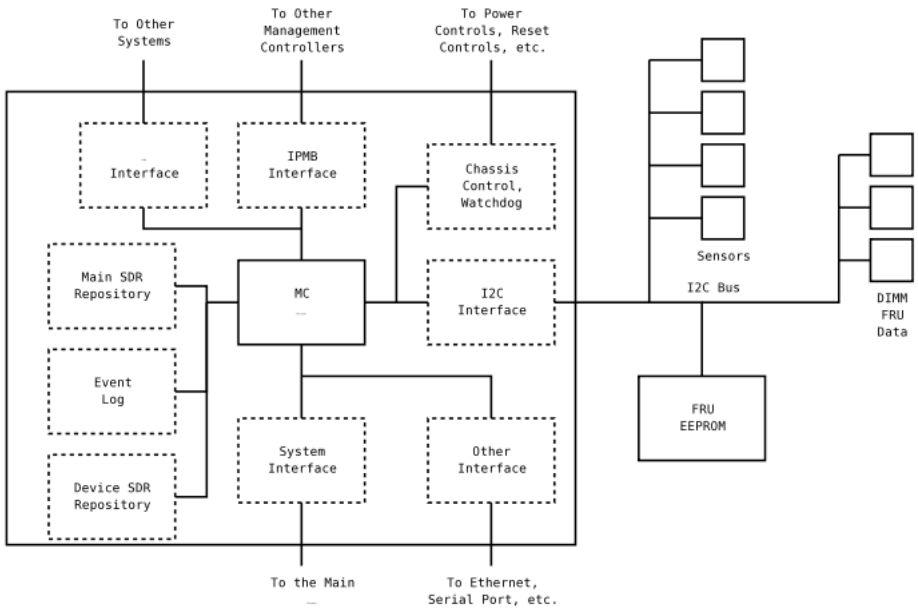


Figure 2.2: Parts of Baseboard Management Controller

## 2.3 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).

### 2.3.1 Introduction to 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.[1]

The Baseboard Management Controller (BMC) as introduced in Figure 2.1 is elaborated in Figure 2.2. (This figure is made available from the documentation of OpenIPMI [3]). As we can infer from the figure, the BMC acts at the center of the IPMI system, providing interfaces to all the various components of interest namely the sensors. More importantly, the figure presented is a generic one in that the interface to the sensors of the MC

[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.

might be different for different hardware manufacturers. But the MC manages to hide these details and provides proper abstractions so that the sensors can be read through the IPMI. Another relevant information from the figure is the Sensor Device Record (SDR) repositories that store information about the sensors. It is the SDRs which the BMC will contact to fetch the required information when we query for details. Notable sensors available in our system are CPU temperature sensor (one each for each processor), voltage sensors, multiple CPU fan sensors, multiple system fan sensors and battery sensors.

### 2.3.2 OpenIPMI

OpenIPMI [3] 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, at a specific level of event registration, OpenIPMI did not provide stable/ working mechanism at the time of development. Hence, we had to resort to the more costly periodic sampling alternative. More details on how OpenIPMI is configured and used are presented in the Section 2.5.

## 2.4 Load Balancing with Ganglia

When a node failure is predicted due to deteriorating health, as indicated by the sensor readings, a target node needs to be 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.

## 2.5 PFT Daemon Design

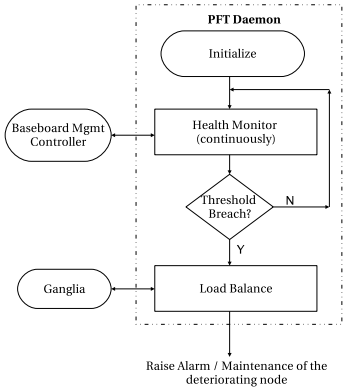


Figure 2.3: Proactive Fault Tolerance Daemon

We have designed and implemented a proactive fault tolerance daemon (PFTd). In our system depicted in Figure 2.1, 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 2.3). 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. The above described operations are explained in the following sections in detail.

### 2.5.1 Initialization

Initialization of the daemon happens in the following steps.

1. The PFTd reads a configuration file containing a list of parameters/sensors to be monitored. 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.

2. Next, the PFTd initializes the OpenIPMI library using the calls available through the OpenIPMI interface. It also gathers the details needed to sets up a connection for the specified network destination (determined by the type of interface, *e.g.*, as LAN, remote host name and authentication parameters, such as user-id and password). A connection to the BMC becomes available after successful authentication.

3. Using the above collected details, a domain is created (using the domain API) so that various entities (fans, processors, etc.) are attached to it. The sensors monitor these entities. It is during this call that a connection is tried to set up with the BMC.

4. OpenIPMI, as we discussed earlier, provides an event-driven system interface, which is somewhat involved, as seen next. As in any event-driven system, 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 in the previous step, 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.

5. 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.

6. 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.

7. 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.

As mentioned earlier namely, there were problems with the functionality of the OpenIPMI version with the registration of the final handler with the IPMI, the callback was not triggered with the OpenIPMI interface. So we resorted to the periodic sampling by forcing a read of the sensors after a sampling interval.

### 2.5.2 Heath Monitoring and Load Balancing

After the above discussed lengthy one-time initialization, the PFTd goes into a health monitoring mode by communicating with the BMC. Here it starts monitoring the health *via* periodic sampling of values from the given set of sensors before comparing it with the threshold values.

In case of any threshold is exceeded, control is transferred to the load balancing module of the PFTd. Here, a target node needs to be selected to migrate the guest VM to. The PFTd then connects to the locally running Ganglia daemon to determine the least loaded node. Since the state of the entire cluster is broadcasted *via* UDP and available at each node of the cluster, the selection process of the spare node is almost instantaneous.

Since the PFTd runs on a privileged domain and also has been authorized to use the xend (Xen daemon), it can perform administrative task on the Guest domain. Having identified a potential fault and also a target 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 can raise an alarm to inform the administrator about the change and can log the sensor values that caused the disruption pending further investigation.

# Chapter 3

# Experimental Setup

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 [13] 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 [53]. 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.[1] 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.

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

# Chapter 4

# Experimental Results

Our experiments focus on various aspects: (a) overheads associated with node failures — single or multiple failures[1], (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.

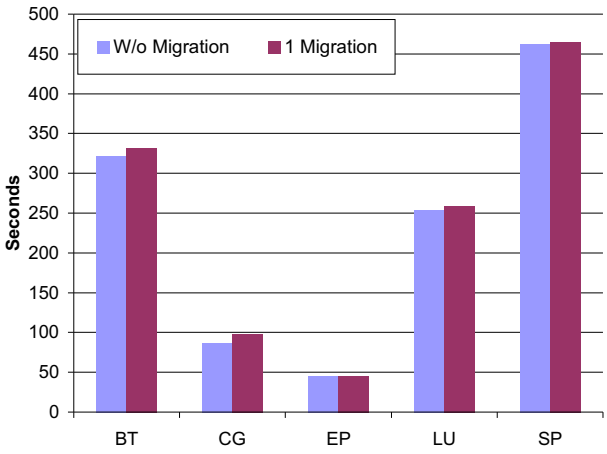


Figure 4.1: Execution Time for NPB Class C on 16 Nodes (standard deviation for wallclock time was 0-5 seconds — excluding migration — and less than 1 second for migration overhead)

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

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

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 4.1, 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 terms of wallclock time. The migration overhead of single-node and double-node failures over 4 base nodes is depicted in Figure 4.2. 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.

## 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 4.3). 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.

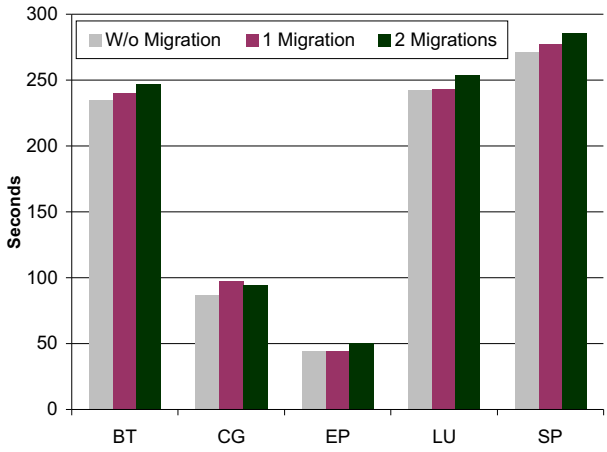


Figure 4.2: Execution Time for NPB Class B on 4 Nodes

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 4.3 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 [38]. 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.

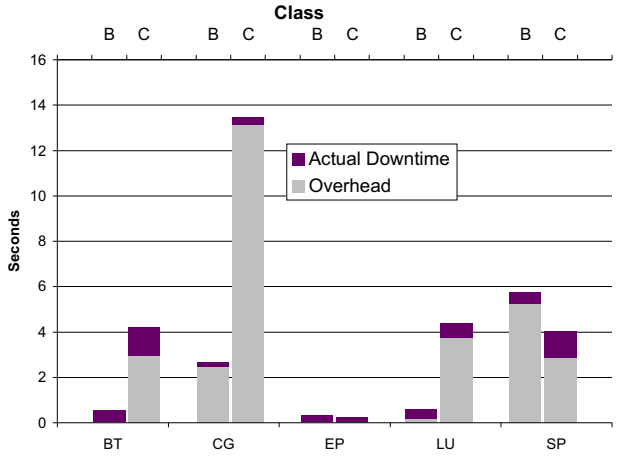


Figure 4.3: Problem Scaling: Migration Overhead for NPB on 16 Nodes

## 4.4 Effect of Task Scaling

We next examined the behavior of migration by increasing the number of nodes involved in computation. Figure 4.4 depicts the overhead for the benchmarks with Class C inputs on varying number of nodes (4, 8/9 and 16).

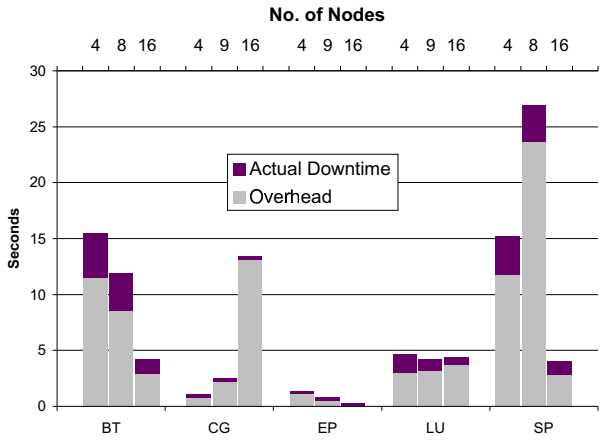


Figure 4.4: Task Scaling: Migration Overhead for NPB Class C

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 4.5 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.

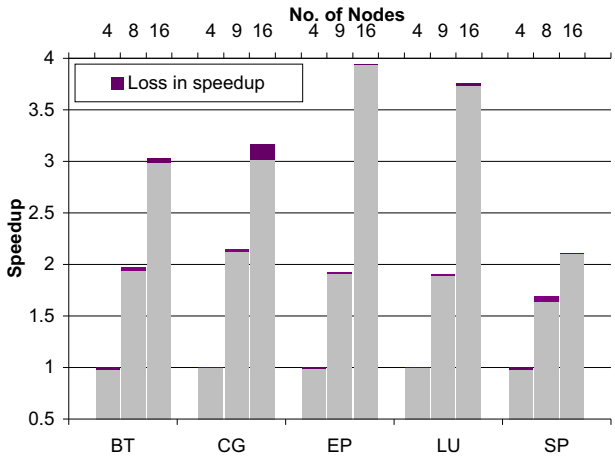


Figure 4.5: Speedup for NPB Class C

## 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 4.3, 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.

The attractive feature about the stop & copy 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 4.6 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.

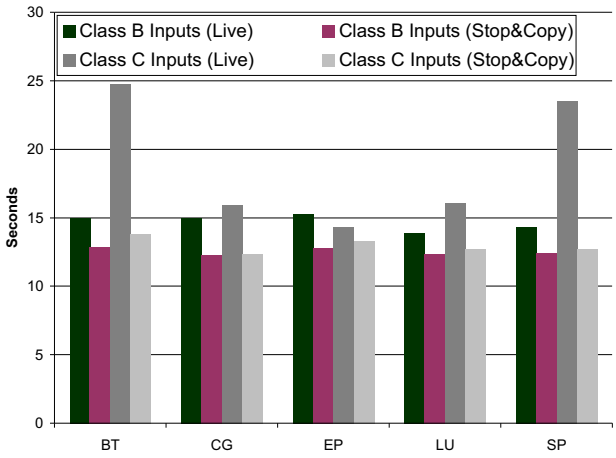


Figure 4.6: Migration Duration for NPB on 16 Nodes (with a standard deviation of 0.7-3 seconds)

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 4.1. 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 4.6. 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 4.7 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-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

Table 4.1: Memory Usage, Page Migration Rate on 16 Nodes

| 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 | 10.71 | 10.71 | 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 |

for scalability within the range of available nodes in the cluster.

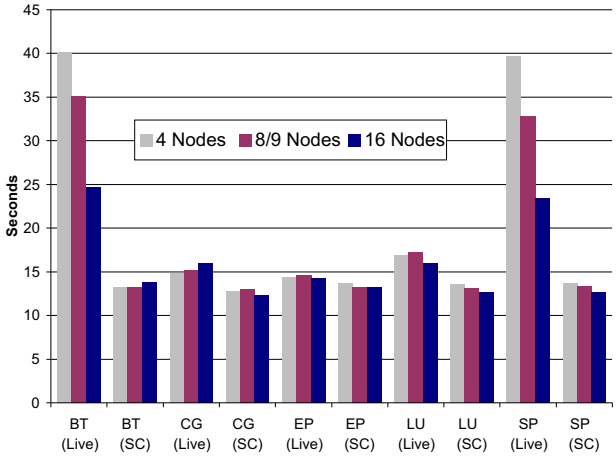


Figure 4.7: Migration Duration for NPB Class C inputs

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 4.8 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 wallclock 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 4.7), live migration overall outperforms the stop & copy approach.

Besides the above comparison, the actual migration duration largely depends on

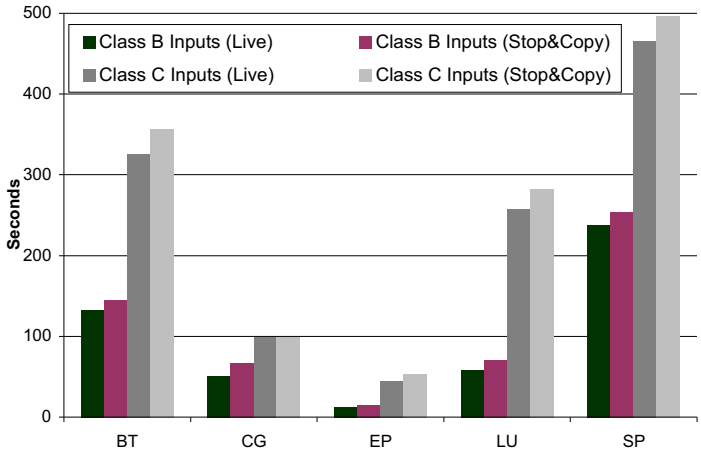


Figure 4.8: Execution Time for NPB on 16 Nodes

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 [43].

# Chapter 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) [48, 44, 8] to non-automated approaches [5, 18]. 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 [16]) combined with a coordinated checkpoint negotiation using collective communication among MPI tasks. Another variation to the checkpointing approach is a co-operative checkpointing scheme [38] 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 thesis 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 [5] 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 [18] 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 [7] result in virtually no overheads [27]. To make virtualization competitive for message-passing environments, OS bypassing is required for the networking layer [34, 33]. This thesis 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 [13]. 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 [37]. 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 [10, 11, 12]. Causal message logging is due to Elnozahy *et al.* [17]. 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 [3] 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 levels has different merits due to associated costs. Process-level migration [41, 49, 30, 6, 15, 16] 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.

In related, orthogonal work [51], experiments were conducted with process-level BLCR [16] 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 [7]. 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 thesis focuses on the OS virtualization side.

Even though the prediction of faults seems to be a difficult task, best effort can be put forward with the available monitoring tools like IPMI, SMART and by inspecting the system log for IO or file system errors. As a considerable improvement over the ordinary threshold-based approach to detect hard-drive failures using SMART, [35] compares various machine learning methods for predicting hard-drive failures using the attributes monitored internally by individual drives. This work shows that SMART thresholds implemented in hard drives provide a 3-10% failure detection rate with 0.1% false alarms whereas the machine learning algorithms can push the failure detection rate as high as 50.6% without

any false alarms. Also, the majority of the failures were predicted within 100 hours before the failure with machine learning methods. In [32], the authors had collected event logs on BlueGene/L over a period of 100 days and investigated the characteristics of fatal failure events, including their relation to non-fatal events. This work showed that it is possible to predict around 80% of memory and network failures and 47% of the application I/O failures.

In contrast to reactive and proactive failure management schemes, [22] discusses an on-line failure forecast system to achieve predictive failure management for fault-tolerant data stream processing. While we have concentrated on identifying hardware failures with hardware sensors, this scheme focuses on observing system logs (*e.g.*, available memory, CPU idle time) and application behavior to identify degradation in performance of applications as an indicator for failure. This approach uses hooks in the applications (similar to query processing) to identify degrading performance (*e.g.*, reduction in number of output tuples per second) and uses the alerts generated for performing preventive actions. The above work on on-line failure prediction was further shown to be applicable to distributed stream processing systems [23]. While our primary focus has been the HPC environment, this approach focuses mainly on the distributed processing systems.

In addition to the health monitoring framework that we have implemented, there is another framework called Nagios [2], a system and network monitoring application that provides alert services for specific registered events. Furthermore, studies have been conducted in the use of artificial neural networks for the purpose of improved fault prediction. Neural networks can be trained for specific input and output patterns — those datasets that actually produced fault and those that did not. Based on the input datasets, the neural network learns autonomously and provides a better indication of occurrence of faults in the system.

In [50], the authors provide a generic framework based on a modular architecture allowing the implementation of new proactive fault tolerance policies/mechanisms. An agent oriented framework [28] was developed for grid computing environments with separate agents to monitor individual classes or subclasses of faults and proactively act to avoid or tolerate a fault.

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 [24], a solution closely related to Xen's

live migration, which is implemented over the L4 microkernel [25]. Even non-live migration strategies under virtualization [45, 31, 52, 39] could be integrated but would be less effective due to their stop & copy semantics. Demand-based migration [54], however, is unsuitable in a proactive environment as it does not tightly bound the migration duration.

# Chapter 6

# Conclusion and Future Work

In this work, we have provided answers to the original thesis statement in Section 1.5 as follows.

- We have shown how automatic and transparent fault tolerance can be supported in a proactive manner for arbitrary MPI applications on a cluster based HPC environment.
- Virtualization is a feasible option for providing proactive FT. By combining virtualization techniques with health monitoring and load-based migration, we assessed 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.

Node failures on contemporary computers can often be anticipated by monitoring health and detecting a deteriorating status. To exploit anticipatory failures, we have promoted 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.

# Bibliography

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

[2] Nagios. http://nagios.sourceforge.net/.

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

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

[5] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall, and Mitchel W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *International Parallel and Distributed Processing Symposium*, 2004.

[6] Amnon Barak and Richard 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.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Symposium on Operating Systems Principles*, pages 164–177, 2003.

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

[9] Ralph Butler, William Gropp, and Ewing L. Lusk. A scalable process-management environment for parallel programs. In *Euro PVM/MPI*, pages 168–175, 2000.

[10] 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.

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

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

[13] C. Clark, K. Fraser, S. Hand, J.G. 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.

[14] D. Culler and J. Singh. *Parallel Computer Architecture*. Morgan Kaufmann, 1998.

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

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

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

[18] Graham E. Fagg and Jack 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.

[19] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[20] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[22] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. Online failure forecast for fault-tolerant data stream processing. In *International Conference on Data Engineering*, 2008.

[23] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. Toward predictive failure management for distributed stream processing systems. In *International Conference on Distributed Computing Systems*, 2008.

[24] Jacob Gorm Hansen and Eric 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.

[25] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of $\mu$-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, October 1997. ACM Press.

[26] Chung-H. Hsu and Wu-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.

[27] 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.

[28] M. T. Huda, H. W. Schmidt, and I. D. Peake. An agent oriented proactive fault-tolerant framework for grid computing. In *International Conference on e-Science and Grid Computing*, pages 304–311, 2005.

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

[30] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

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

[32] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. Bluegene/l failure analysis and prediction models. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 425–434, Washington, DC, USA, 2006. IEEE Computer Society.

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

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

[35] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado. Comparison of machine learning methods for predicting failures in hard drives. *Journal of Machine Learning Research*, 2005.

[36] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *ICS '07: Proceedings of the 21st annual International Conference on Supercomputing*, pages 23–32. ACM, 2007.

[37] 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.

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

[39] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. In *OSDI*, 2002.

[40] Ian 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.

[41] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Symposium on Operating Systems Principles*, pages 110–119, October 1983.

[42] Sunil Rani, Chokchai Leangsuksun, Anand Tikotekar, Vishal Rampure, and Stephen Scott. Toward efficient failre detection and recovery in hpc. In *High Availability and Performance Computing Workshop*, page (accepted), 2006.

[43] 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.

[44] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, October 2003.

[45] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *OSDI*, 2002.

[46] Bianca Schroeder and Garth 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.

[47] Hertong Song, Chokchai Leangsuksun, and Raja Nassar. Availability modeling and analysis on high performance cluster computing systems. In *First International Conference on Availability, Reliability and Security*, pages 305–313, 2006.

[48] 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.

[49] Marvin Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. In *SOSP*, pages 2–12, 1985.

[50] Geoffroy Vallee, Kulathep Charoenpornwattana, Christian Engelmann, Anand Tikotekar, Chokchai (Box) Leangsuksun, Thomas Naughton, and Stephen L. Scott. A framework for proactive fault tolerance. In *International Conference on Availability, Reliability and Security*, pages 659–664, March 4-7, 2007.

[51] 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), April 2007.

[52] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Symposium on Networked Systems Design and Implementation*, pages 169–182, 2004.

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

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