A Scalable Unified Fault Tolerance for HPC Environments

Kulathep Charoenpornwattana[*], Chokchai Leangsuksun[*1], Geffory Vallee[**], Anand Tikotekar[**], Stephen Scott[**2]

[*] Louisiana Tech University, Ruston,
71272 Louisiana, U.S.A
{kch020, box}@latech.edu
[**] Oak Ridge National Laboratory, Oak Ridge
37831 Tennessee, U.S.A
{valleegr, tikotekaraa, scottsl}@ornl.gov

Abstract. *Reliability is one of the major issues in High Performance Computing (HPC) systems today. It is expected to become even a greater challenge in the next generation peta-scale systems. The traditional fault tolerance mechanisms (e.g., checkpoint/restart mechanisms) may not be efficient in every scenario in such large scale systems due to the scalability and performance issues. In this paper, we aim to provide a distributed scalable Unified Fault Tolerance (UFT), which consists of Proactive Fault Avoidance (PFA) and traditional Reactive Fault Tolerance (RFT) for HPC systems based on fault prediction and virtualization technologies. The results from simulation suggest the performance improvement over the existing solutions.*

1. Introduction

In a development of a fault tolerance (FT) framework for HPC systems, scalability and performance are the most two important aspects that we need to take to our consideration. The goal of this research is to design and develop a proof-of-concept fault tolerance framework to deploy in a large scale cluster. Therefore, the overhead from the fault tolerance framework must be minimized.

In general, FT policies can be classified in two categories, namely, Reactive Fault Tolerance (RFT) and Proactive Fault Avoidance (PFA). Reactive Fault Tolerance (RFT) (*e.g., Checkpoint/Restart* - C/R) is the traditional fault tolerance approach and considered as the most reliable which has been widely deployed in the most production HPC clusters today's. The RFT approach allows long-running applications to be restarted from the last checkpoint, if any nodes in the cluster fail during their execution. However, since there is no one-policy-fits-all, the RFT alone may be not suitable for every the HPC system especially in the large scale systems. The RFT ap-

[1] Research supported by the Department of Energy Grant no: DE-FG02-05ER25659
[2] Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

proach currently presents two main drawbacks: (i) it produces unnecessary overhead if no failure occurs, and (ii) it suffers from the scalability issues (typically caused by I/O [4]).

The PFA is another approach that is capable to anticipate different component faults on the system, which allows the framework to take actions proactively against system failures to avoid the recovery overhead and minimize the impacts, which may disrupt executing applications. The performance of the PFA approach however directly depends on the accuracy of fault prediction and the number of predictive faults on the system. In addition, it may also suffer from the scalability issues caused by fault-alarm.

In this paper, we propose a PFA solution with event-driven and mailbox techniques to avoid unnecessary overheads from periodic data polling and from data diffusion, targeting a deployment in a next generation HPC systems. More details of the framework will be discussed later. Our PFA is able to interplay with existing RFT mechanisms to form up a *Unified Fault Tolerance* framework.

The remaining of the paper is organized as following. Section 2 discusses related work. Section 3 entails our Proactive Fault Avoidance framework. Section 4 describes the proof-of-concept integrated framework. Section 5 is analysis and discussion. Section 6 concludes the work.

2. Related Work

Applying virtualization techniques in HPC clustering to improve system reliability is relatively new. Although there are a number of solutions [2] [3] that are available based on system-level virtualization to ensure the applications survivability. Other previous studies on OS level PFA [2] are based on Xen [11] with hardware monitoring mechanisms such as Intelligent Platform Management Interface (IPMI) [13] and Self-Monitoring Analysis Reporting Technology (SMART) [12]. In [2] the authors combined the thermal sensors with kernel log parser to predict I/O and file system errors. Nagarajan, et al [3] developed a framework based on Ganglia to determine load average of remote systems prior the virtual machine migration. The VM are transferred to one of the spare nodes that have lowest load average for the best computation performance.

On the other hand, the RFT mechanisms has been long studied and widely implemented in today's production HPC systems. The most common RFT mechanisms are Checkpoint/Restart (C/R) and Rollback/Recovery mechanisms. Applications must be checkpointed periodically, and restarted from the last checkpoint when a failure occurs. The runtime or applications may need to be modified for checkpoint mechanisms to work properly. Examples of process-level RFT mechanisms are BLCR [9], Kerrighed [6] and Libckpt [8]. Several studies on transparent and automated C/R have been done based on LAM/MPI and BLCR. The framework in [1] provides compute nodes monitoring mechanism, and a transparent automated recovery enhancement to HA-OSCAR [7]. With the framework, MPI application can continue to run regardless of head node or compute nodes failures. The study on a job pause service for transparent FT [5] is another example of process-level RFT. Addition to transparent

and automated mechanisms, a job pause service also provides a mechanism to preserve all the healthy processes and to restart only the process on the failed node. The network connections of the operational nodes are reused; hence, the restart overhead is reduced. Some of the checkpoint/restart mechanisms are also included in the jobs/resources scheduler such as Loadleveler [16], and Condor [15]. Moreover, there are several checkpoint/restart implementations available in commercial version such as Evergrid's Cluster Availability Management Suite [18].

3. The Scalable Proactive Fault Avoidance (PFA)

Most of today's mid to high end hardware are often integrated with the system health/components monitoring systems allowing users to gather system status and attributes. The main purposes of such mechanisms are to reduce the complexity and the maintenance cost as well as to improve the system reliability [14]. In addition, the data gathering from such mechanisms could also be used for faults analysis and prediction. For example, Intelligent Platform Management Interface (IPMI) allows user to monitor system fans, temperatures, voltages, etc. In some failure scenarios, such as failures due to the temperature, IPMI can help detecting the increased temperature before the heat terminates the system and permanently damages the hardware components.

3.1 Overview

The basic idea of PFA is to anticipate components faults on every node in the system with integrated common health monitoring systems (*e.g.*, IPMI, SMART, lm-sensor). If the fault is predicted on one of the nodes, we take advantage of VM fault management features (*e.g.*, live migration) to mitigate or avoid the system failure, which can crash the running applications.
However, the PFA mechanism could potentially impose performance overhead and scalability issues especially in a large scale system because data needs to be globally polled periodically. In addition, massive data collected on every node for fault prediction are distributed across the system, which could devour an extensive amount of the network bandwidth. Nevertheless, the benefit from PFA mechanisms may outweigh performance penalty that may occur and new virtualization technologies such as Intel VT [20] or AMD Pacifica [21] may help alleviating such concerns. Therefore, we investigate several techniques in order to address performance and overhead concerns implied by the PFA approach.

3.2 Methodology

To resolve the aforementioned issues, the periodically global-data polling and the unnecessary communications between nodes must be eliminated or minimized. We introduce an abstraction in the PFA framework, which consists of three major mod-

ules: one for *fault prediction*, one that implements an *event system*, and one for the implementation of the *fault tolerance policy*.

The fault prediction module is distributed on every node across the system. It keeps monitoring several sensors with system health monitoring mechanisms. In this paper, we use the IPMI system health monitoring mechanism to gather system data and sensor thresholds. The common sensor data that can be retrieved from IPMI includes system/processor temperature, system voltages, system/component fan speed. The sensor thresholds are the manufacturer specified environmental conditions where the system should be operated. If the system doesn't operate under the specified conditions, it may indicate anomaly. We use this fact to develop our fault prediction system. This IPMI based monitoring mechanism is hardware based and thus does not interfere with main processing units, if any critical sensors drop below or raise beyond the manufacturer's pre-set thresholds, the alarm is generated and diffused via the event system. Therefore, when the system is healthy (components are in normal working condition), the fault prediction system is in idle mode.

In addition, we are also developing an intelligent fault prediction technique based on Neural Network with IPMI data. The goal is to improve the prediction accuracy of the framework. We used the historical IPMI data to train the Neural Net, and used it to predict future sensor data.

In order to minimize the communications in the system, we have designed a module called *event system* to handle all communications in the system in a more efficient way. The event system is considered as a core of the framework and acts as communication backbone among other modules. Furthermore, we included a notion of mailbox technique, to eliminate repeated global data polling. The concept of the mailbox technique is similar to the magazine delivery in the postal system. There are three parties involved in the system: i) subscribers, ii) publishers, and iii) a post office. The process starts by subscribers expressing their interest in subscribing magazines to a post office and magazine publishers. Magazine publishers also need to contact the post office to arrange the bulk delivery to every subscriber. The post office needs to maintain the record of publishers and subscribers and makes sure that when there are magazines arrived, they are delivered to subscribers in timely manner.

We applied the same "*mailbox*" idea to our PFA framework. The event system acts as a post office. The modules on the system could be one of these three following cases: i) only subscriber, ii) only publisher, or iii) both publisher and subscriber. The subscriber modules register themselves and express their interest in receiving some or all magazines (events). The publisher modules register themselves and inform the event system which type of events they will be publishing. Once the publisher module generates an event, the event is sent to the event system. Then it forwards the event to interested subscribers based on their record.

Applying this technique to PFA framework clearly has three major benefits: i) neither subscribers nor event system need to periodically query for the events of their interest; ii) the event system is not implemented on compute nodes, therefore, the event handling does not impact the application execution; iii) new subscribers/publishers can be easily added to the framework without modifying any other participant.

The fault tolerance module takes actions to evade system failures caused by predicted faults. Specifically, the computing entities such as applications or VMs are either migrated away from the faulty components through virtual machine's live migration or will be readjusted the resources usage to avoid the faulty components.

3.2.1 Virtualization on Multi-Core Architecture

Due to the recent trends of processors, the size of new released processors will pack more circuits (the size of the silicon is predicted to reduce to 16nm by year 2013 and 8 nm by year 2017 [22] and have more computational cores on the same die rather than higher clock speed. The statement is confirmed by presentations of two leading processor manufacturers, Intel and AMD, during the Oklahoma supercomputing symposium 2007. Whereas, operating systems and applications still take each computational core as one CPU. For example, one system has dual quad-core processors with hyper-threading disabled. The number of processors appears to be 8 on the operating system. In addition, the advancement in VM technology allows VM to be assigned to specific cores or processors. In other words, virtual CPUs of virtual machine could be mapped to the physical cores/processors.

Because of these changes in processor trends, we have modeled our fault tolerance policy to mitigate faulty cores/processors. For that, we assume that we have fault detection/prediction mechanisms to anticipate faults at the computational core level, and fault containment mechanism to prevent the faults to affect other components causing the system total failure. Even though, these features may not be available at the present time, we however envision that, there will be such mechanisms in the future. It is quite necessary to have such fault tolerance in the next generation multi-core processors system. There are three fault tolerance strategies that we have modeled in our fault tolerance architecture. We have included 3 fault tolerance models in our PFA.

3.2.2 Virtual Machine Inter-node Live Migration

Virtual machine live migration (Figure 1) allows a relocation of active virtual machines between physical nodes over the network. The overhead of the migration is considerably less than checkpoint/restart mechanisms [3]. We hope to take advantage of live migration feature to migrate the running VMs away from nodes that are about to fail to one of the healthy spare nodes. The migration has minimum impacts to the running applications as long as we can predict an imminent failure in advance enough to complete the migration. The live migration is one of important features associated to system-level virtualization and has been included in many modern virtualization solutions (*e.g.*, Xen, VMware).

The only restriction that comes along with the inter-node live migration is that, the source and destination systems are expected to have identical configurations. In other words, processors must be the same type, and the memory and disk space on the destination host must be larger than the source.

3.2.3 Fault Masking

The number of computational cores in future generation processors will increase extensively in the next few years. The chance of processor failure could also increase as the number of computational cores increases. Therefore, we expect at some point

that we will face the situation where increasing the number of computational cores without built-in fault tolerance mechanisms could face some serious issues. For example, if any computational cores fail during the job execution, it could cause the entire processor or system to fail and could eventually crash the running jobs.

Fault masking technique (Figure 2) could help to reduce impacts of such issues; the technique can be done at the Virtual Machine Monitor (VMM) or at the hardware level. When an error is predicted on one of the computational cores, the VMM masks a failure by masking the core as faulty and un-mapping the Virtual CPU (VCPU) that is currently mapped to the faulty core. The fault masking technique helps reducing the uncertainty of running jobs on unreliable computational cores. The drawback of this approach is that, the performance of the VM is slightly decreased since the number of available cores is less from fault masking.

3.2.4 Live VCPU Remapping

The *live VCPU remapping* approach (Figure 3) allows user to change the mapping policy between the computational cores and VCPUs while the VM is still active. In some scenarios, for example, one of the computational cores is detected with an unusual behavior; we can use the live VCPU remapping to remap the VCPU from defect cores to non-defect cores, or from defect processor to non-defect processors. In such a situation, live VCPU remapping could significantly help reducing overhead of inter-node live migration and recycling some of the valuable resources. The live VCPU remapping requires the spare resources (cores/processors), so that if some faults occur, we can use the spare resources on the same machine or die.

The idea behind live VCPU remapping is quite similar to the fault masking approach. It, however, not only avoids the fault components, but also allocates and utilizes some of the available spared resources to sustain the performance.

To illustrate our different fault tolerance policies, we present three failure scenarios. In the first scenario, all resources (cores) are allocated to one VM. There are no spare resources on the system; therefore, the live VCPU remapping is not applicable. Since there are no spare resources, if there are some faults on the system, the VM needs to be relocated to a more stable node. With live migration, the virtual system is allowed to relocate without interrupting the running applications. In Figure 1, a VM runs on a system with multi-core processor and utilizes all the resources (left). We assume that a system has a spare node. Once, a failure is predicted on the environment that the VM is currently on; in this case, the only option to prevent the failure is to use the inter-node live migration. The running VM is lively migrated from the fault system to the same configuration spare node (right).

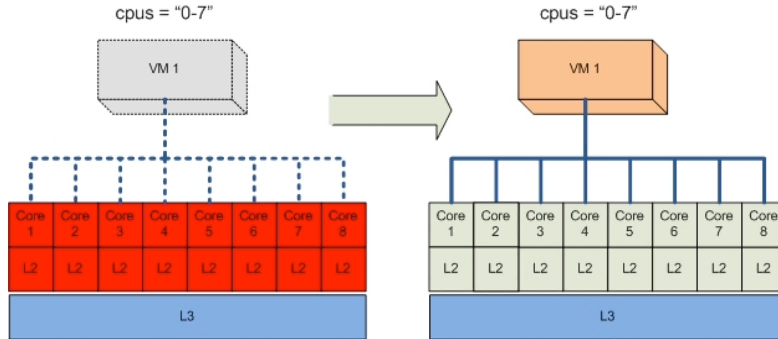

Fig. 1. Inter-node Live Migration

In Figure 2, the system has a single multi-core processor (left). There is one VMs running, and fully utilizes all the cores. Then core 2 on processor is predicted with some faults. The VMM of VM1 then acknowledges the issue and marks core 2 as faulty, and then it re-maps the VCPU of VM 1 to exclude the fault core (right).

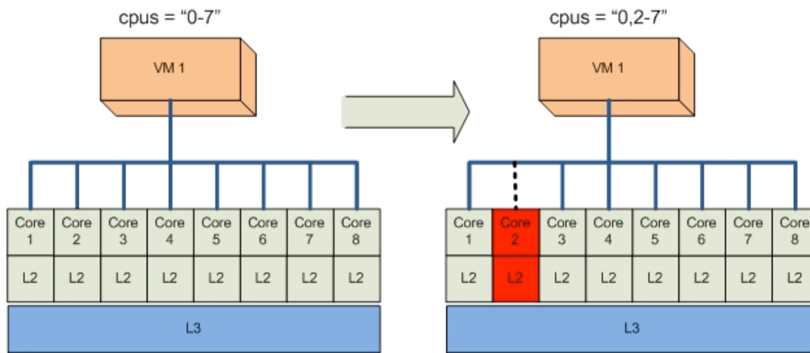

Fig. 2. Fault Masking

The third failure scenario is shown in Figure 3. On the left, a VM runs on top of a system with a single multi-core processor. The VCPUs of the virtual machine are mapped to 6 computational cores and have 2 cores as spare cores. On the right, faults predicted on core 2 and 4. The VCPUs are remapped from defect core (core 2 and 4) to the healthy backup cores (core 7 and 8) by VMM.

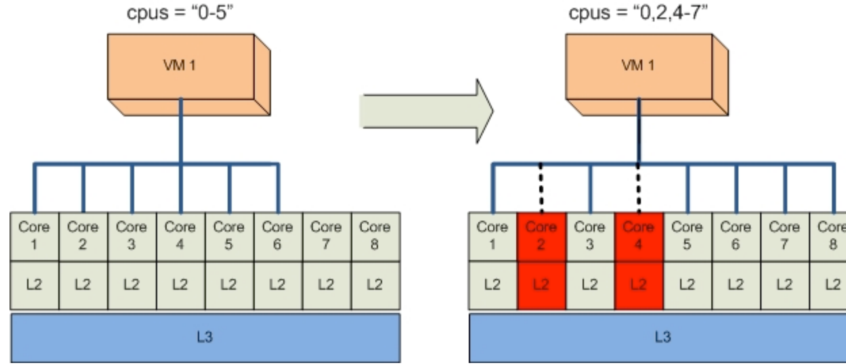

Fig. 3. VCPU Remapping

3.3 Architecture and Implementation

Our PFA Framework was designed and implemented with modular architecture and composed of three main modules: (i) *the fault prediction system*, (ii) *the event system*, and (iii) *fault tolerance daemon*. Fault Prediction system aims to proactively anticipate components faults which could lead to the potential system failure on each node. OpenIPMI tool is used to gather sensor data and thresholds. If any sensor values drop below or raise beyond the thresholds, an event alarm is generated and sent to the event system for further analysis. The received alarm may or may not trigger any actions based on the predefined policy. The default policy may be defined to relocate the VM running on a faulty system to one of the healthy nodes. Fault Tolerance Daemon (FTD) takes an action request from the event system then proceeds (if policy decided to take action) according to the policy. We enlist each module and its implementation in details below.

3.3.1 Fault Prediction System Implementation

We employed Nagios [10], a services and networks monitoring tools, as the core FP module in our prototype. Nagios is one of the most popular open-source system and network monitoring tools. It is designed to monitor virtually any systems/services and can be integrated with alerting systems, which is very helpful for detecting unusual events. Moreover it has a built-in log feature, which enables us to track any anomaly events for future failure analysis. Nagios framework consists of three modules: monitoring module (plug-in), Nagios Core Logic, and event handler/notification modules. Nagios architecture adopts a modular approach, which allows users to create and easily integrate custom plug-in/event handlers into its framework in order to monitor new entities and handle new events. Additional, all options (*e.g.*, monitor interval) can be customized via Nagios's configuration files. Therefore, we selected Nagios as an event monitor to fulfill our needs of FP. However, the core framework could be substituted with other monitoring frameworks with a similar modular and efficient approach (*e.g.*, MonAMI).

3.3.2 Event System

The event system is the most important modules in our FT solution. It is considered as the core of the framework because all the events occur in the system are directed to and handled in this module. As we proposed, the event-driven architecture with the mailbox technique is the most suitable approach for our event system.

The event-driven implementation with the mailbox technique is not so trivial. Since the system has to involve with all other components and handle every single event in the framework. The event system must provide two common interfaces for interaction among other components. The interface should include a "*post*" operation for allowing publishers to send the events to the event system and a "*register*" operation for enabling subscribers to express their interest to receive the events. In addition, event system needs to have buffer cache to store all the events, in order to resend events if subscribers are not available.

We currently design our framework based on a global view of the distributed system for simplification. We also include the state handler for updating global system status. Since, in the FT framework, there might be more than multiple events occurring in the same period, we need to make sure that all the received events are handled and not lost. For this reason, the event system must maintain a list of alarmed nodes. This list will be updated when the node is recovered or if there is a permanent failure, which needs a manual recovery.

3.3.3 Fault Tolerance Daemon

Fault Tolerance Daemon (FTD) stands by on every compute node in the cluster and interacts with the head node, when policy decides to take actions against the failures. Currently three actions are allowed (VM pause, VM un-pause, and inter-node live migration), However, these actions may differ depend upon the underlying mechanisms. Because FTD was designed to be generic and pluggable, thus we included concept of connector. A connector is the abstraction of underlying mechanisms, which allows new low-level mechanisms to be added without modifying the core framework. Another important aspect of connector concept is that the underlying mechanisms are only visible to the direct-interfaced modules. The detail implementation of our PFA is explained in our previous work [2].

4. The Unified Fault Tolerance Framework

Generally, failures in HPC clusters can be seen from two standpoints: predictive and non-predictive failures. A predictive failure could be anticipated prior to the actual failure. It usually depreciates the system over time and eventually causes the system to completely terminate if not be predicted (*e.g.*, failures caused by temperature). Today, most of servers have integrated hardware sensors and management mechanisms for monitoring and predicting such fault in various components in the system. This data can be accessed via operating system APIs through in-band or direct out-band channels and could be used in real-time analysis for fault prediction. Details of fault predictive analysis can be found in [23]. On the other hand, a non-predictive failure is caused by errors that are not anticipated before their occurrence. It is an un-

foreseen event, which cannot proactively be predicted by any fault prediction mechanisms, and usually causes more damage to the system because the system may not be prepared for such events (*e.g.*, power outages).

Our objectives are to design and develop a framework to handle failures in both scenarios as we clearly categorized them earlier. Thus we proposed the Unified Fault Tolerance (UFT) framework with two fundamental policies: PFA and RFT. The PFA aims to handle predictive failures by anticipating and proactively taking actions before the failure occurs to prevent further damages and reduce the failures recovery overhead. The RFT aims to handle non-predictive failures or predictive failures that PFA fails to engage.

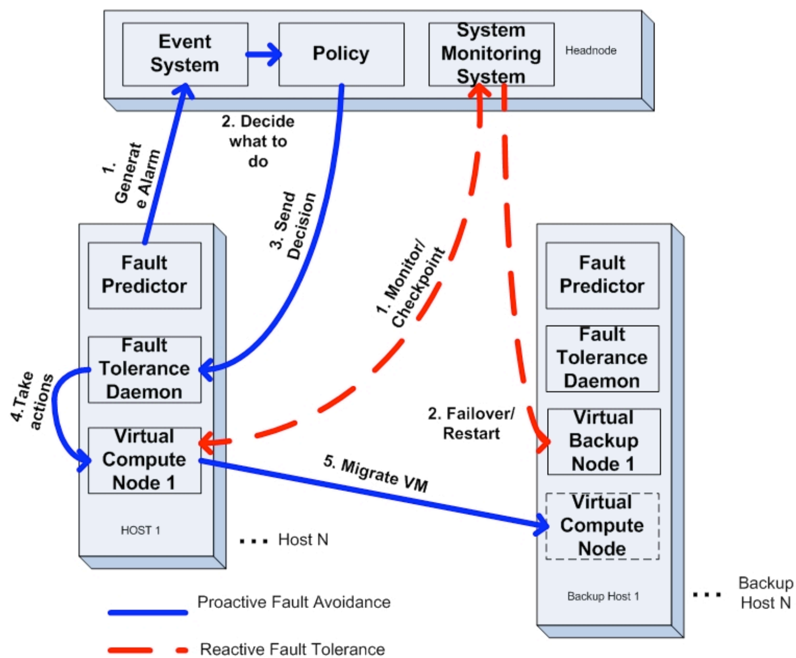

Fig. 4. Architecture of the Unified Fault Tolerance Framework

Figure 4 illustrates the architecture of the UFT framework. The framework consists of control two main flows; the solid lines represent a flow of PFA and the dot lines represent the flow of RFT. Even thought, virtualization technology enables us to deploy multiple VMs per host. According to our preliminary experiment, stacking VM however will significantly reduce the computational performance due to the fact that the resources will be shared among those VMs. [23]

5. Evaluation and Analysis

Our framework describes the UFT policy, which combines the two complementary FT mechanisms; namely a PFA and a RFT. In this section we evaluate these mechanisms on their general properties using a simulation approach, which is described in detail in [17]. In [17], we employed our simulator to evaluate various FT policies such as fully proactive, fully reactive, hybrid or proactive combined with reactive using failure logs from Lawrence Livermore National Laboratory (LLNL). In this paper, we extend data collection period from the same source.

We start by describing the failure distribution as it affects the simulated applications.

The applications run by the simulator are considered standard MPI applications with no FT capabilites. Moreover, it is possible to setup simulator parameters in term of overhead associated to fault tolerance mechanisms (for instance, it is possible to set the VM migration and checkpoint/restart overhead). We set parameters based on experimentations on our physical execution environment.
Figure 5 shows how four applications, also called "job", are affected when the simulated failure prediction accuracy varies (the cluster is shared between these four applications).

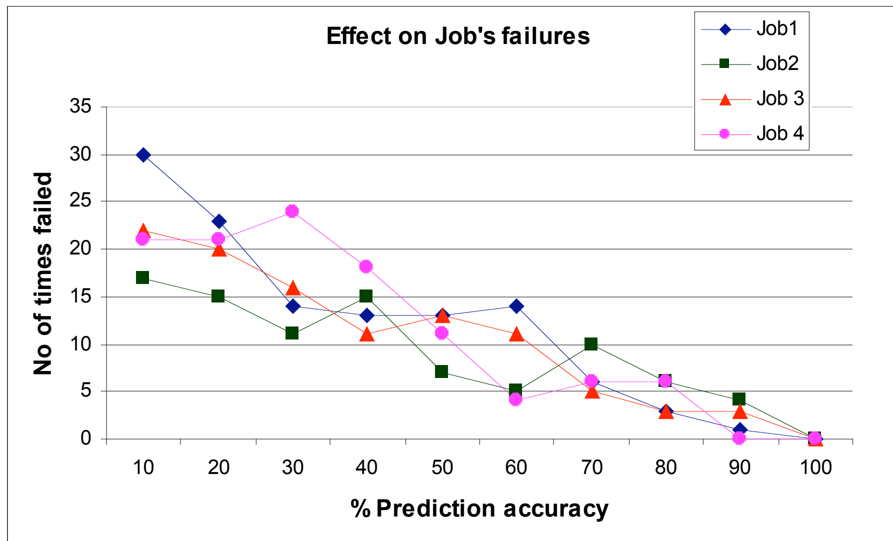

Fig. 5. Effect of Fault Prediction Accuracy on Application Execution

The two months failure data begins from 1/10/2001 and ends on 3/10/2001 with 117 failures. In [17], our data contained only 41 failures. The average execution time for each application/job is 1415 hours (approximately: 2 months) in the first case. An average application ran for 744 hours (approximately 1 month). We have marked failures as predictable or unpredictable (based on the distinction we made in the previous sections) using a 70/30 scheme. The 70/30 scheme means that 70% of the predictable

failures are marked in the second half of an application's execution time-frame, while 30% are marked in the first half. The reason for this was that the density of failures in the first half (from 1/10 to 2/10) is approximately 30% and 70% in the second half. Please note that although we use a 70/30 scheme, the actual marking of failures remains random [17].

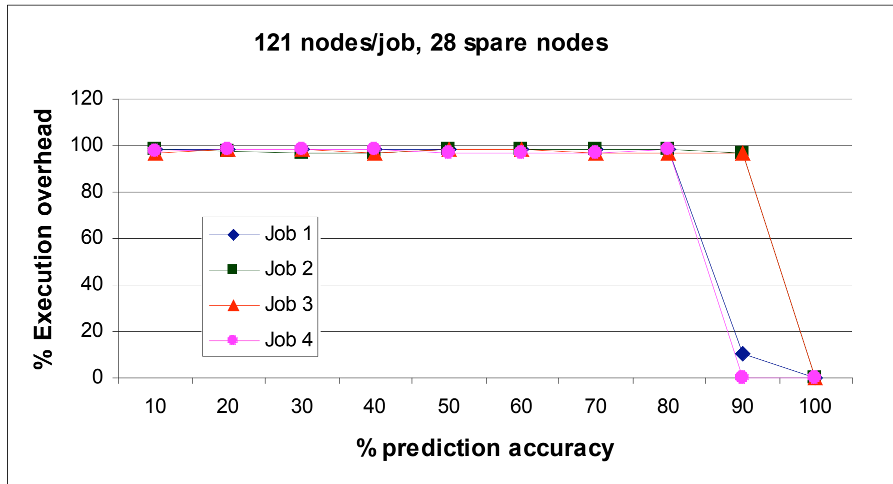

Fig. 6. Proactive Fault Tolerance Policy

In Figure 6, we evaluate a PFA policy. The policy runs four jobs each having 121 nodes with 28 spare nodes. As mentioned, with our framework, an entire virtual machine is migrated to an available spare node in the event of a failure alarm. The cost of each migration is 10 minutes. The PFA results from the simulator currently do not consider stacking of VMs on a single host OS. The results suggest that the PFA strategy is almost useless when the prediction accuracy is not perfect. The reason for such an extreme conclusion is a simple observation that a pure PFA depends on the location of the last failure. It can be easily seen that each unpredicted failure leads to a restart of the failed application from scratch, undoing any work done by an application until the failure time. In this particular case, there are enough failures toward the end of the application execution so that all applications must be restarted from scratch. Figure 6 highlights this problem to an extent that even 90% prediction accuracy cannot save application overhead. We agree that a pure PFA is at the mercy of a "single failure that can not be predicted" (especially with the simulated platform). However, before dismissing pure PFA as too precarious to be useful, a thorough study must be performed on its properties so that it may be used in conjunction with RFT for greater efficacy.

We now evaluate a RFT policy. As mentioned, with our the framework, an RFT policy is based on reactive measures in the event of failures; in our case, application checkpoint/restart. We consider a checkpoint overhead of 50mins/checkpoint (value based on experimentations using our execution environment). We do not consider re-

start latency/overhead for this run. This policy does not contain any failure prediction information and therefore is failure reactive only.

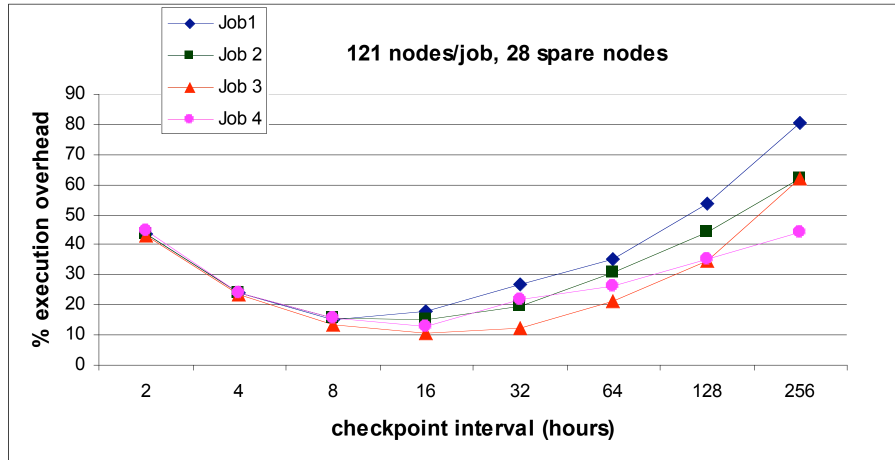

Fig. 7. Reactive Fault Tolerance Policy

Figure 7 exhibits a bathtub curve behavior. In [17], we also had a similar result for a reactive policy but differed in its "sinking point". We expect this phenomenon since the results are from the same execution platform even though the data is different. At this point, if we compare PFA with RFT, we can see that RFT is clearly better because it has less rollback time.

We can now appreciate that UFT aims to address drawbacks of both PFA and RFT. The weak point of a PFA is the loss of all work time in the case of unpredictable failure (the application is restarted from scratch). The main weak point of a RFT is the unnecessary checkpoint overhead, especially the overhead caused by checkpoints for failures that could have been predicted.

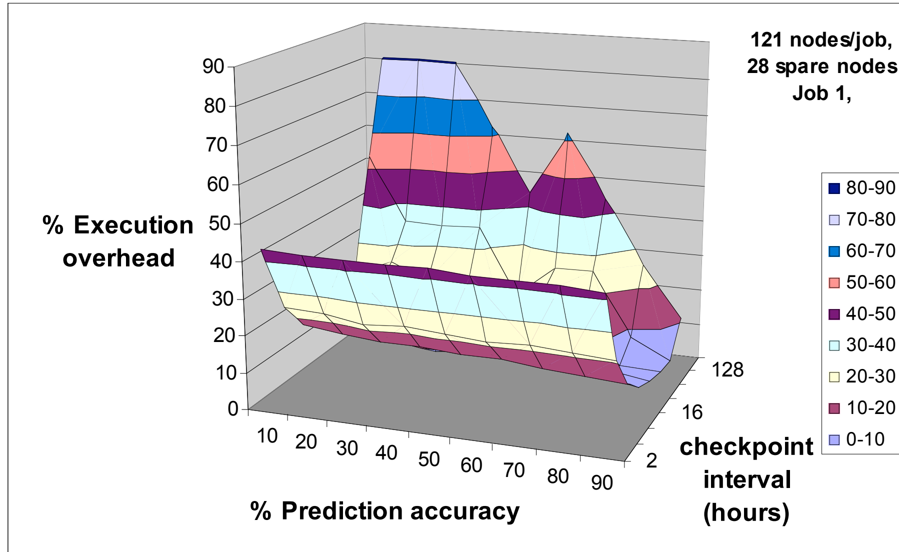

Fig. 8. Impact of a Unified Fault Tolerance Policy

Figure 8 shows the impact of a UFT policy, *i.e*., combining RFT and PFA, on one application (called "job 1"). It is easy to see that UFT is clearly better than PFA alone for almost all the prediction accuracies except for 90% and 100%. UFT is also better than RFT for most values of checkpoint intervals. Only at the extreme ends of PFA and RFT, does the UFT falter. The reasoning for such a scenario is obvious: for very low checkpoint intervals the benefit of a PFA starts to decrease due to less rollback time and the overhead due to migration adds up. Similar reasoning is applicable for very high perdition accuracies.

However, to summarize, it can be said that a UFT outperforms both PFA and RFT in most cases.

6. Conclusion and Future Work

In this paper, we present a unified fault tolerance framework, which consists of proactive fault avoidance (PFA), and reactive fault tolerance (RFT) framework aiming to provide resiliency in the large scale HPC systems. Our PFA framework is designed with event-driven architecture with mailbox technique to minimize the computation and communication of the framework. Furthermore, we foresee the direction of the next generation processors from the new released processors and technology trends. Therefore, we developed a novel fault tolerance techniques like fault masking and VCPU remapping to include in our framework. Our PFA is unified with the RFT based transparent checkpoint/restart framework as reactive mechanism to form a complete UFT framework.

We validated potential benefits of the UFT framework with the fault tolerance simulator, which is based on failure logs from LLNL's ASC White. The experimental results suggest that the performance of our UFT framework is better than the PFA or RFT solely in most cases. The exclusive PFA framework, however, performs better than UFT at > 90% accuracy but it is very difficult to develop a mechanism to anticipate fault at that high precision.

The failure prediction mechanism of PFT in the current implementation relies on hardware health monitoring mechanisms (*e.g.*, IPMI, SMART) and kernel log parser (*e.g.*, dmesg), which are still in a very infancy phase. More sophisticated prediction mechanisms need to be developed and incorporated into the framework in order to increase the prediction accuracy. In addition, we will pursue the analysis of historical data to determine the probability of failures of the received alarm and develop better policy to handle more complex situations with various types of events and to minimize fault positive alarms.

References

[1] A. Tikotekar, C. Leangsuksun, and S. L. Scott. "On the survivability of standard MPI applications", In Proceedings of 7th LCI internation conference 2005, Norman, OK, USA, May 1-4, 2006.

[2] G. Vallée, K. Charoenpornwattana, C. Engelmann, A. Tikotekar, C. Leangsuksun, T. Naughton, S. L. Scott, "A Framework For Proactive Fault tolerance", In proceedings of International Conference on Availability, Reliability and Security (ARES 2008)

[3] A. Nagarajan, F. Mueller, C. Engelmann, S. L. Scott, "Proactive Fault Tolerance for HPC with Xen Virtualization"in ICS '07: Proceedings of the 21st annual international conference on Supercomputing. New York, NY, USA97–104.

[4] Ron Oldfield, "Investigating Lightweight Storage and Overlay Networks for Fault Tolerance", HAPCW 2006 Santa Fe, New Mexico

[5] C. Wang, F. Mueller, C. Engelmann, S. L. Scott, "A Job Pause Service under LAM/MPI + BLCR for Transparent Fault Tolerance", in Proceedings of International Parallel and Distributed Processing Symposium, Apr 2007

[6] C. Morin, et. al., "Kerrighed: a single system image cluster operating system for high performance computing" in Proc. Of Europar 2003.

[7] HA-OSCAR: http://xcr.cenit.latech.edu/ha-oscar

[8] S. Plank, "Libckpt: Transparent Checkpoint under UNIX", In Usenix Winter 1995 Technical Conference 1995

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

[10] Nagios : http://www.nagios.org

[11] Xen: http://www.xensource.com

[12] Bruce Allen, "Monitoring Hard Disks with SMART", Linux Journal, January 2004

[13] Intelligent Platform Management Interface: http://www.intel.com/design/servers/ipmi/

[14] Intel, "How IPMI Provides Mechanisms for Maintaining and Monitoring the Health of a Linux Sysem.
http://www.intel.com/cd/ids/developer/asmona/eng/os/linux/resources/whitepapers/53867.htm

[15] Condor High Throughout Computing: http://www.cs.wisc.edu/condor/

[16] LoadLeveler's Checpoint Restart Mechanism: http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.loadl.doc/loadl331/am2ug30309.html

[17] A. Tikotekar, G. Vallée, T. Naughton, S. L. Scott, and C. Leangsuksun, "Evaluation of fault-tolerant policies using simulation", in Proceedings of IEEE Cluster 2007, Austin, Texas USA.

[18] Evergrid : http://www.evergrid.com

[19] Google SoC 2007, "Cluster Virtualization with Xen" : http://code/google.com/soc

[20] Intel VT : http://www.intel.com/

[21] AMD Pacifica: http://www.amd.com/

[22]Stephen Wheat, "Exa, Zeta, Yotta: Not Just Goofy Words", http://symoposium2007.oscar.ou.edu/oksupercompsym/p2007_talk_wheat_exazetayotta_20071003.pdf

[23] Kulathep Charoenpornwattana, "HPC cluster reliability improvement with Virtualization Technologies", Masters thesis, Louisiana Tech University 2008