Monitoring and Predicting Hardware Failures in HPC Clusters with FTB-IPMI

Raghunath Rajachandrasekar[1], Xavier Besseron[2] and Dhabaleswar K. Panda[1]

[1] *Network-Based Computing Laboratory*
The Ohio State University
{rajachan, panda}@cse.ohio-state.edu

[2] *Computer Science and Communication Research Unit*
University of Luxembourg
xavier.besseron@uni.lu

***Abstract*—Fault-detection and prediction in HPC clusters and Cloud-computing systems are increasingly challenging issues. Several system middleware such as job schedulers and MPI implementations provide support for both reactive and proactive mechanisms to tolerate faults. These techniques rely on external components such as system logs and infrastructure monitors to provide information about hardware/software failure either through detection, or as a prediction. However, these middleware work in isolation, without disseminating the knowledge of faults encountered. In this context, we propose a light-weight multi-threaded service, namely FTB-IPMI, which provides distributed fault-monitoring using the Intelligent Platform Management Interface (IPMI) and coordinated propagation of fault information using the Fault-Tolerance Backplane (FTB). In essence, it serves as a middleman between system hardware and the software stack by translating raw hardware events to structured software events and delivering it to any interested component using a publish-subscribe framework. Fault-predictors and other decision-making engines that rely on distributed failure information can benefit from FTB-IPMI to facilitate proactive fault-tolerance mechanisms such as preemptive job migration. We have developed a fault-prediction engine within MVAPICH2, an RDMA-based MPI implementation, to demonstrate this capability. Failure predictions made by this engine are used to trigger migration of processes from failing nodes to healthy spare nodes, thereby providing resilience to the MPI application. Experimental evaluation clearly indicates that a single instance of FTB-IPMI can scale to several hundreds of nodes with a remarkably low resource-utilization footprint. A deployment of FTB-IPMI that services a cluster with 128 compute-nodes, sweeps the entire cluster and collects IPMI sensor information on CPU temperature, system voltages and fan speeds in about *0.75 seconds*. The average CPU utilization of this service running on a single node is *0.35%*.**

***Keywords*-Fault detection, coordinated fault propogation, IPMI, FTB, and Clusters.**

I. Introduction

Modern High Performance Computing (HPC) Clusters continue to grow to ever increasing proportions. However, performance gains that could be obtained in traditional single/dual core processors by employing schemes such as frequency scaling and instruction pipelining have greatly diminished due to problems in power consumption, heat dissipation and fundamental limitations in exploiting Instruction Level Parallelism. Processor speeds no longer double every 18 - 24 months. As a result, HPC systems no longer rely on the speed of a single processing element to achieve the desired performance. They instead tend to exploit the parallelism available in a massive number of moderately fast distributed processing elements which are connected together using a high performance network interconnect such as InfiniBand [1].

However, as these clusters scale out, their Mean Time Between Failures (MTBF) rapidly deteriorates. With this exponential increase in the number of components in the cluster, the MTBF has reduced from days to a couple of hours [2]. Many real world applications that study Molecular Dynamics [3]–[5], Finite Element Analysis [6], [7], etc. take anywhere from a few hours to a couple of days to complete their computation. Given that the MTBF of such modern clusters is smaller than the average running time of these applications, multiple failures can be expected during the lifetime of the application. As a result, it has become imperative for these clusters to be equipped with Fault Tolerant capabilities.

Although most of the individual hardware and software components within a cluster implement mechanisms to provide some level of fault tolerance, these components work in isolation. They work independently, without sharing information about the faults they encounter. This lack of a system-wide fault information coordination has emerged to be one of the biggest problems in leadership-class HPC systems. Furthermore, fault-prediction is a challenging issue that several researchers are trying to address. Fault-prediction models and toolkits will have to work in unison with fault coordination and propagation frameworks allowing system middleware to make informed decisions. This will also make way for proactive measures and actions that provide resiliency to end-user applications.

In this context, our work presented in this paper addresses the following questions:

1) Can a scalable light-weight tool that provides services like distributed fault monitoring, failure event propagation and failure prediction be designed?
2) How can existing HPC middleware leverage fault-information from such a service to provide preemptive fault-tolerance?

Figure 1 illustrates the fundamental contributions of this paper and shows how existing system software stack can make use of the proposed service, namely FTB-IPMI. FTB-IPMI uses two key technologies - the Fault-Tolerance Backplane (FTB) developed as a part of the Coordinated Infrastructure for Fault Tolerant Systems (CIFTS) initiative [8] to propagate fault information, and the IPMI interface standard to monitor system events. These two technologies are explained in depth in Section II. A rule-based prediction engine has been developed as part of the MVAPICH2 [9] MPI library to demonstrate how it can benefit from the failure information given by FTB-IPMI. Such an integration to provide fault-resilience capabilities is possible with any FTB-enabled software that can subscribe

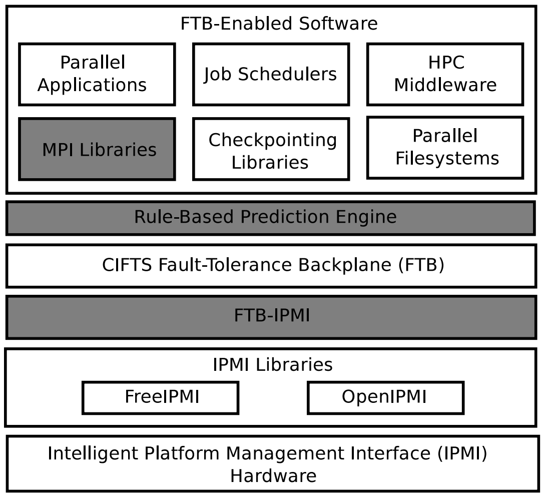

Fig. 1: FTB-IPMI Architecture

to events from FTB-IPMI. Section V discusses how existing FTB-enabled middleware can benefit from FTB-IPMI.

II. Background

A. Fault-Tolerance Backplane(FTB)

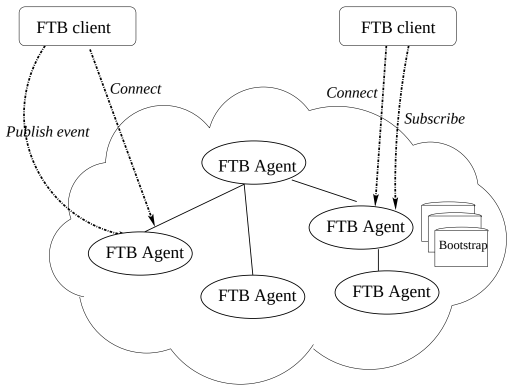

Fig. 2: FTB Architecture (Courtesy: [10])

The CIFTS Fault Tolerance Backplane [10] is an asynchronous messaging backplane that provides communication between the various system software components. The Fault Tolerance Backplane (FTB) provides a common infrastructure for the Operating System, Middleware, Libraries and Applications to exchange information related to hardware and software failures in real time. Different components can subscribe to be notified about one or more events of interest from other components, as well as notify other components about the faults it detects.

The FTB physical infrastructure is shown in Figure 2. The FTB framework comprises a set of distributed daemons, called FTB Agents which contain the bulk of the FTB logic and manage most of the book-keeping and event communication throughout the system. FTB agents connect to each other to form a tree-based topology. If an agent loses connectivity during its lifetime, it can reconnect itself to a new parent in the topology tree, making the tree fault tolerant and self-healing. From the software perspective, the FTB Software Stack consists of three layers, namely, the Client Layer, the Manager Layer and the Network Layer.

The Client Layer consists of a set of APIs for the clients to interact with each other. These FTB Client APIs consist of a small set of simple but powerful routines. The `FTB_Connect` API is used by the clients to connect to the FTB framework. Each client has to register itself for a given *Namespace*. Once registered, the Namespace to which the client belongs cannot be changed during its lifetime. `FTB_Publish` is used by the clients to advertise a specific event to other clients. All events thrown by the client will belong to the Namespace to which the client had registered. The events can have varying severity such as INFO, WARNING, ERROR or FATAL. Clients which have subscribed to that event will be notified through either synchronously if they poll for events or asynchronously through a registered callback. In addition to the event name, the `FTB_Publish` also allows clients to include a small amount of data as a *payload*. This proves to be a useful feature, as will be seen in the future section. `FTB_Subscribe` is used to indicate the Namespace of Events for which the client needs to be notified. The `FTB_Unsubscribe` and `FTB_Disconnect` APIs are used by clients to disassociate from the FTB infrastructure.

The Manager Layer handles the bookkeeping and decision making logic. It handles the client subscriptions, subscription mechanisms and event notification criteria. This layer is responsible for event matching and routing events across to other FTB Agents. This layer exposes a set of APIs for the Client Layer to interact with it. The interface is internal to FTB and is not exposed to external clients.

The Network Layer is the lowest layer of the software stack. This layer deals with the sending and receiving of data. The Network Layer is transparent to the upper layers and is designed to support multiple communication protocols such as TCP/IP and shared-memory communication.

B. Intelligent Platform Management Interface (IPMI)

The Intelligent Platform Management Interface (IPMI) [11] defines a set of common interfaces to a computer system which can be used to monitor system health. IPMI consists of a main controller called the Baseboard Management Controller (BMC) and other management controllers distributed among different system modules that are referred to as Satellite Controllers (SC). The BMC connects to SCs within the same chassis through the Intelligent Platform Management Bus/Bridge (IPMB). Amongst other pieces of information, IPMI maintains a Sensor Data Records (SDR) repository which provides the readings from individual sensors present on the system, including, sensors for voltage, temperature and fan speed.

IPMI can be used to monitor system health using in-band (running locally) or out-of-band (connected remotely) methods. In-band monitoring can be done using one of the several drivers, including a direct Keyboard Controller Style (KCS) interface driver, a Linux SMBus System Interface (SSIF) driver through the SSIF device (i.e. /dev/i2c-0), the OpenIPMI Linux kernel driver (i.e. /dev/ipmi0), and the Sun/Solaris BMC driver (i.e. /dev/bmc). Out-of-band monitoring is done by communication with a remote node's BMC which has been configured for such communication.

There are several tools and libraries available to query the IPMI hardware for system events and sensors readings. These include OpenIPMI [12], FreeIPMI [13], ipmiutil and ipmitool. The FreeIPMI library was chosen for the implementation of FTB-IPMI as it supports both in-band and out-of-band monitoring based on IPMI v1.5/2.0 specification. It has features like sensor monitoring, system event monitoring, power control and serial-over-LAN (SOL). The FreeIPMI package includes several tools and libraries that facilitate IPMI-based system management. It also has several OEM-specific add-ons via ipmi-oem for a variety of vendors like Dell, Fujitsu, IBM, Inventec, Quanta, Sun and Supermicro.

There are four main libraries that are a part of FreeIPMI project - Libfreeipmi, Libipmiconsole, Libipmimonitoring and Libipmidetect. Libfreeipmi is a C library that includes KCS, SSIF, and OpenIPMI Linux, and Solaris BMC drivers, IPMI 1.5 and IPMI 2.0 LAN communication interfaces, IPMI packet building utilities, IPMI command utilities and utilities for reading/interpreting/managing IPMI. Libipmiconsole is a library for Serial-over-Lan (SOL) console access. SOL console access is abstracted into a file descriptor interface, so users may read and write console data through a file descriptor. Libipmidetect is library for IPMI node detection. Libipmimonitoring is library for sensor monitoring and interpretation. Sensor monitoring and interpretation of those sensors is abstracted into an API with an iterator interface. FTB-IPMI uses the Libipmimonitoring library to monitor sensor readings and states.

III. Design and Implementation

FTB-IPMI is designed to run as a single stand-alone daemon which handles multiple operations like reading IPMI sensors, classifying events based on severity, and propagating the fault information via FTB. A single instance of the FTB-IPMI daemon running on one node can manage an entire cluster.

Once initialized, the following actions are performed at periodic user-set intervals (as illustrated in Figure 3):

(A) Querying IPMI: Collects the values and states of IPMI sensors on all nodes
(B) Sensor State Analysis: Analyzes collected data and identifies relevant events that need to be propagated
(C) Event Publication: Publishes FTB events that correspond to observed sensor state changes
(D) Component Notification: The FTB framework notifies all FTB-enabled system components that subscribe to events from FTB-IPMI

Figure 3 describes the work-flow of FTB-IPMI running on a representative computing cluster installation with a front-end node and a set of compute nodes. In such a typical configuration, FTB-IPMI runs on the head node as a central service to monitor all the compute nodes. The following subsections describe steps (A) through (C) in detail.

A. Querying IPMI

FTB-IPMI gathers information about the IPMI sensors of all the nodes using FreeIPMI library described in Section II-B. Privileged-access to any of the compute nodes or the node which hosts the FTB-IPMI daemon is not required, as the sensor data is only being fetched and not written. However, FreeIPMI's out-of-band monitoring interface requires a username/password based authentication to communicate with remote BMCs.

FTB-IPMI takes a host-list as mandatory argument from the user during startup. This host-list should contain the hostnames/IPs of the nodes that need to be monitored for faults by FTB-IPMI. Typical supercomputing systems have a dedicated Ethernet network for IPMI traffic. In such cases, the hostname or the IP address corresponding to this network would have to be specified. Based on this host-list, an internal task-list is generated. Each task corresponds to an IPMI query that must be performed.

In order to efficiently collect data from a large list of hosts, the FTB-IPMI daemon is multi-threaded. The number of threads is defined internally during initialization, or via a user-provided configuration file. Threads in this context refer to a pool of worker threads that pick tasks from the task-list as they become idle. Such a mechanism also provides load-balancing amongst the threads. Once assigned a list of tasks, each worker thread uses the out-of-band monitoring interface provided by IPMI to query sensor data from the nodes in its list. On fetching sensor data from a given node, it is removed from the task list. Once the task list is empty, data collection is considered complete. The internal task list gets regenerated for each periodic iteration.

Sensor data queries posted to IPMI using the FreeIPMI library are blocking in nature. A querying thread blocks on the response from a remote BMC without querying any other BMC. Using multiple threads allows several BMCs to be queried in an overlapped manner. Furthermore, these multiple worker threads provide some level of tolerance to node failures. If a node stops responding to IPMI requests, the worker thread handling this IPMI query will be blocked until the operation times out. However, the other threads can progress with their tasks, thereby limiting the delay induced by the node failure. The performance impact of having multiple worker threads is evaluated by means of experimentation in Section IV.

B. Sensor State Analysis

Table I shows an example of the data collected by FreeIPMI for a given node in a single iteration. In addition to sensor name, information about a sensor such as its *Type*, its current *Value*, the *Unit*, and its *State* are fetched. The *State* of a sensor is decided by FreeIPMI, based on its current value and a sensor-specific threshold. These thresholds and the FreeIPMI behavior can be customized for each node using the `freeipmi_interpret_sensor.conf` configuration file. From the sample readings listed in the table, we can notice that some sensors are in the *Critical* state and report the value to be "0". This indicates that the particular hardware sensor unit is not available. This case is quite frequent and can be safely ignored.

For fault-tolerance purposes, the most relevant information is the state change of a sensor rather than its actual value.

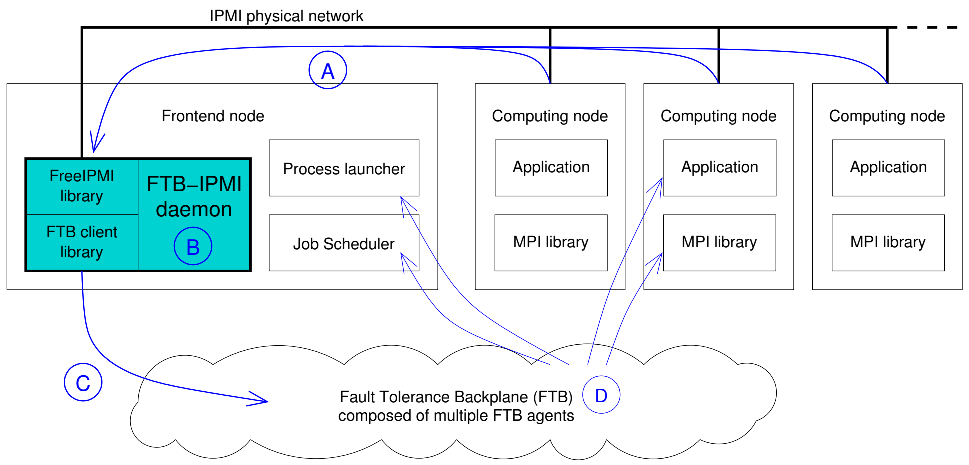

Fig. 3: FTB-IPMI Work-flow

Sensor Name	Type	State	Value	Unit
CPU1 Temperature	Temperature	Nominal	25.00	C
CPU2 Temperature	Temperature	Nominal	26.00	C
TR1 Temperature	Temperature	Critical	0.00	C
TR2 Temperature	Temperature	Critical	0.00	C
VCORE1	Voltage	Nominal	0.94	V
VCORE2	Voltage	Nominal	0.94	V
+1.5V_ICH	Voltage	Nominal	1.53	V
+1.1V_IOH	Voltage	Nominal	1.10	V
+3.3VSB	Voltage	Nominal	3.22	V
+3.3V	Voltage	Nominal	3.24	V
+12V	Voltage	Nominal	12.10	V
VBAT	Voltage	Nominal	3.22	V
+5VSB	Voltage	Nominal	4.96	V
+5V	Voltage	Nominal	4.99	V
P1VTT	Voltage	Nominal	1.14	V
P2VTT	Voltage	Nominal	1.14	V
+1.5V_P1DDR3	Voltage	Nominal	1.50	V
+1.5V_P2DDR3	Voltage	Nominal	1.50	V
FRNT_FAN1	Fan	Nominal	9840.00	RPM
FRNT_FAN2	Fan	Critical	0.00	RPM
FRNT_FAN3	Fan	Nominal	9840.00	RPM
FRNT_FAN4	Fan	Nominal	9520.00	RPM
CPU1_ECC1	Memory	Nominal	N/A	N/A
CPU2_ECC1	Memory	Nominal	N/A	N/A
Chassis Intrusion	Physical Security	Critical	N/A	N/A

TABLE I: FTB-IPMI Sensor Readings from a single compute-node on Cluster A (See Section IV)

Based on this observation, FTB-IPMI is designed to maintain the history of states that each sensor shifts through. This allows for easy detection of state changes and also provides a framework for history-based failure prediction.

IPMI Sensor Event		FTB Action
State change to *Nominal*	⇒	Publish event (Severity `INFO`)
State change to *Warning*	⇒	Publish event (Severity `WARNING`)
State change to *Critical*	⇒	Publish event (Severity `WARNING`)
Read Error	⇒	Publish event (Severity `CRITICAL`)

TABLE II: FTB-IPMI rules to generate FTB event

Table II summarizes the set of internal rules based on which FTB-IPMI generates FTB events. When a sensor moves to the *Warning* or *Critical* state, an event with severity WARNING (and not FATAL) is generated. We consider that a sensor with a *Warning* or *Critical* state is just an indication of a potential failure and not an actual failure. It is also important to generate an FTB event when a sensor goes back to the *Nominal* state, in order to roll-back or cancel any recently triggered (and possibly incomplete) preventive action.

C. FTB Event Publication

The FTB events are published using the FTB client library which automatically connects to FTB agents and forwards the events to all interested parties. Publishing an event is non-blocking action that can also tolerate failures like network faults.

The details of the published FTB events and their payload are given in Table III. The payload includes as much information as possible, including the hostname and the sensor ID. Components that subscribe to events from FTB-IPMI can build logical prediction engines by correlating events, the history of sensor states and the sensor information that is packed into the payload.

D. Rule-Based Prediction in MVAPICH2

In order to demonstrate this capability, we have developed a rule-based prediction engine prototype within the MVAPICH2 MPI library. This prediction engine uses the information provided by FTB-IPMI to predict impending node failures. The prediction from this engine is used to trigger a proactive process-migration protocol which migrates MPI processes from a failing node to a healthy spare-node. This prediction engine is an FTB-enabled component that subscribes to events from FTB-IPMI in the `FTB.IPMI` namespace and publishes prediction information for the MVAPICH2 library to use in the `FTB.MPI.MVAPICH2` namespace. The `PREDICTOR_NODE_FAILURE` event is published with the failing node's hostname and the sensor state information as

Event Name	Payload	Severity	Description
`IPMI_SENSOR_STATE_NOMINAL`	Hostname,SensorID,Name,Type,Value,Unit,State	INFO	Sensor status changes to the *Nominal* state
`IPMI_SENSOR_STATE_WARNING`	Hostname,SensorID,Name,Type,Value,Unit,State	WARNING	Sensor status changes to the *Warning* state
`IPMI_SENSOR_STATE_CRITICAL`	Hostname,SensorID,Name,Type,Value,Unit,State	WARNING	Sensor status changes to the *Critical* state
`IPMI_ERROR`	Hostname,Error	CRITICAL	Error reading IPMI sensors

TABLE III: FTB Events published by FTB-IPMI

payload. This component gathers state history information from the events published by FTB-IPMI, and applies certain rules to predict failures. One such rule expects at least 3 events of `WARNING` severity from the same sensor to be generated before it can deem that hardware to be dying. This rule ignores `CRITICAL` events from sensors having a reading of “0” in order to curb pseudo-predictions. In case of subsequent `WARNING` events related to the system fan speeds, a failure prediction is not generated unless a rise in CPU-temperatures is observed too. This rule ensures that automatic fan-speed control mechanisms ([14], [15]) employed by modern motherboards do not lead to false-predictions. Similar logical rules can be added to this fabric to increase the accuracy of predictions, and to completely suppress false-alarms. This component can also be ported to any of the FTB-enabled software. Section IV-C shows a demonstration of how this component in MVAPICH2 is used to provide proactive fault-tolerance.

IV. Experimental Evaluation

In this section, we discuss some experimental results that demonstrate the capabilities, and evaluate the performance impacts of FTB-IPMI. The computing cluster used for these evaluations, Cluster A, is a 160-node Linux-based system. Each compute node has eight Intel Xeon cores organized as two sockets with four cores per socket and has 12 GB of memory. They are equipped with InfiniBand QDR Mellanox ConnectX-2 HCAs. The operating system used is Red Hat Enterprise Linux Server release 6 with the 2.6.32-71.el6.x86_64 kernel. The cluster also has a separate Ethernet network for the IPMI system to keep its operations independent of user-traffic on the regular network.

A. Resource Utilization

In these experiments, the CPU utilization of FTB-IPMI was profiled. Figure 4 compares the real-time CPU usage of an instance of FTB-IPMI which monitors 128 compute nodes using 128 and 64 worker threads. The initial spike that goes up to 39.4% and 24.6% respectively, is from the operation of spawning all the worker threads during startup. Subsequent smaller spikes are from multiple iterations of sensor sweeps performed by FTB-IPMI. For the purpose of experimentation, each iteration is separated by a 2-second delay. In a typical installation, this iteration delay has to be decided based on the MTBF of the system being monitored, and the tolerable latency with which failure predictions are needed to take appropriate preventive measures. The effect of having multiple client nodes assigned to a each worker thread is seen in the case of 64-threads, where each thread has to monitor two hosts from the tasklist. As discussed in Section III-A, having

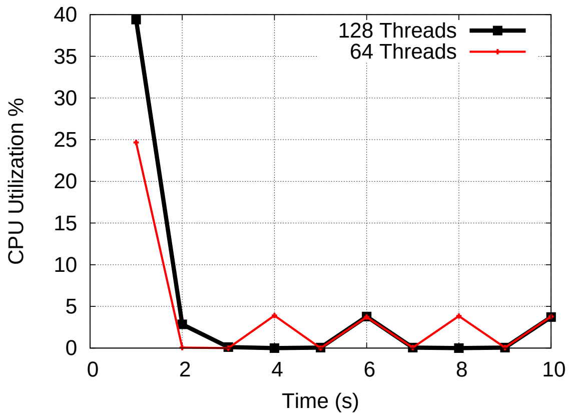

Fig. 4: Real-Time CPU Usage

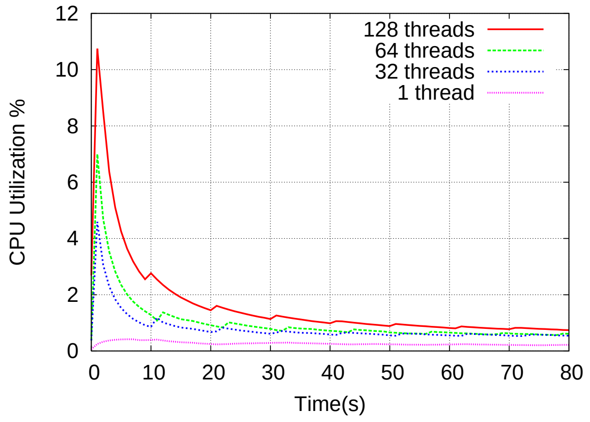

Fig. 5: Average FTB-IPMI CPU Usage

multiple-threads allows several remote BMCs to be queried in an overlapped manner. This explains the additional spikes in the 64-threads case. In this configuration, about 3.7% of available CPU resources on the node hosting FTB-IPMI are used during a single iteration of querying the sensors.

However, to fully capture the CPU utilization of FTB-IPMI service, the average CPU usage of the daemon was measured over time, and is plotted as a graph in Figure 5. For this experiment, the iteration delay was set to 10 seconds and the CPU utilization values were measured for up to 10 iterations. This graph shows an initial spike as well, which can be attributed to the phase where multiple worker threads are spawned. The average CPU utilization increases marginally with increasing number of threads. Considering the case with 128 threads for instance, as execution progresses over the 10 iterations, the average CPU usage stabilizes at about 1.5%. This would be much lesser in a realistic deployment which will have a much larger delay between iterations. This indicates that the FTB-IPMI service has a very low system-utilization footprint and does not let other services and libraries starve for resources. This will not affect any compute-intensive user-applications that run on the compute nodes either, as the daemon runs only on the head-node or management-node of a cluster.

Network resources are critical in a supercomputing environment. However, the communication between different IPMI BMCs used by FTB-IPMI is carried-out over a separate Ethernet network. This ensures that the IPMI out-of-band monitoring traffic does not congest the network that is used for communication by the user/application. Given this exclusivity of the network resource, we have not profiled its usage for the purpose of this paper.

B. Scalability

The ability to scale to a large number of nodes was one of the design goals of FTB-IPMI. In order to demonstrate the scalability of our tool, we conducted several experiments by varying the number of worker threads used, and the number of compute nodes monitored.

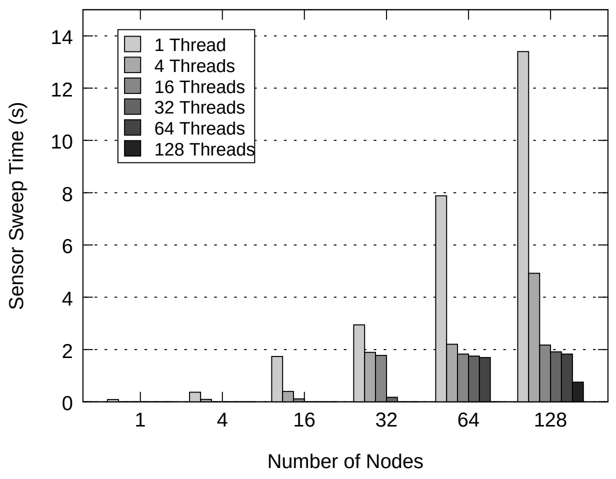

Fig. 6: Scalability with Multiple Threads

Figure 6 provides a comprehensive summary of how FTB-IPMI scales as the number of nodes, and number of threads increases. In this experiment, the number of worker threads used are either equal to, or lesser than the number of nodes in the hostlist. As the trends in the results indicate, even as the number of nodes increase geometrically, the sensor sweep times increase just sub-linearly.

To get a deeper understanding of how the multi-threaded design of FTB-IPMI enhances the scalability of the service, we varied the number of threads used to query sensors across 128 compute nodes, from 1 to 128 (see Figure 7). The time taken for the execution of a single iteration drops significantly when multiple threads are used. Adding more threads beyond a certain point does not help reduce the sweep time either. This is due to contention at the thread level, where resource allocations get multiplexed. A single iteration of FTB-IPMI to read all the sensor readings from 128 nodes takes just 0.75 seconds with 128 threads.

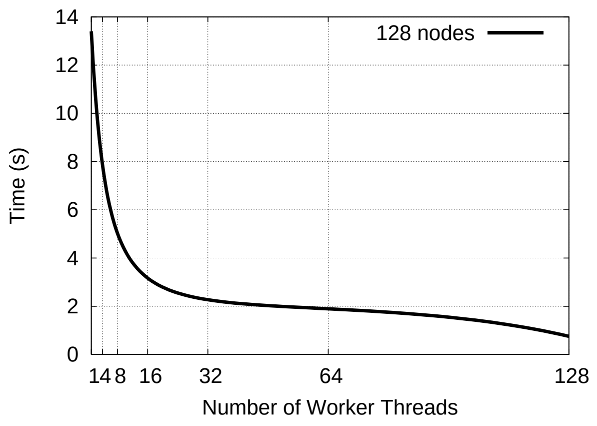

Fig. 7: Execution times for Single Iteration

C. Proactive Process Migration in MVAPICH2

Section III-D describes the fault-prediction engine integrated into the MVAPICH2 library. To study how this component interacts with the FTB ecosystem and system software to provide proactive fault-tolerance, we ran the Tachyon [16] MPI ray-tracing application on 128 nodes of Cluster A. Multiple IPMI events indicating a rise in CPU Temperatures (`WARNING` severity) were injected into the FTB-IPMI service in order to simulate a potential failure. On seeing three consecutive `WARNING` events from IPMI, the prediction engine realizes that one of the predefined rules have been satisfied, and generates a `PREDICTOR_NODES_FAILURE` FTB event, with information about the failing node as payload. This event is then published in the FTB namespace, and delivered to all components that have subscribed to events from the prediction engine. The `mpirun_rsh` process manager in MVAPICH2, which is one of these subscribers, uses the prediction event as a cue to initiate the process migration protocol which moves processes from the failing node to a healthy spare node.

In Figure 8, the X-Axis marks the progress of the application execution in seconds and Y-Axis marks the computation progress in terms of percentage completed. The vertical bars represent either a checkpoint in progress, or a preemptive process migration in progress. The labels marked by pointers from these bars indicate the standardized FTB events that are published by the MPI library for other system components to make use of when taking preventive or reactive measures to handle failures. In this experiment, the MPI library is configured to record periodic checkpoints of the application. In the absence of a fault-prediction mechanism, the application will be rolled-back to a particular checkpoint in case of a failure, from which point the execution resumes during recovery. However, with the help of the failure prediction generated by the rule-based prediction engine, the process-migration protocol is triggered, on completion of which the job resumes execution from the same point at which it was suspended. This protocol significantly improves application resiliency with a minimal overhead incurred to the application performance.

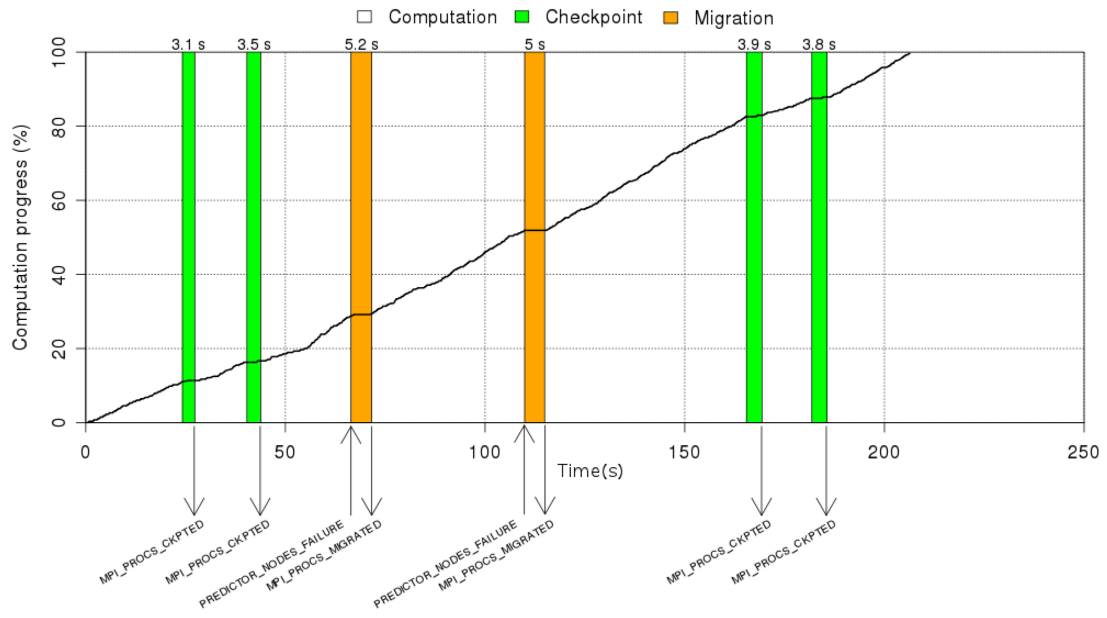

Fig. 8: Prediction-Triggered Preemptive Fault-Tolerance in MVAPICH2

V. Applications of FTB-IPMI

Several High-Performance middleware including MPI libraries, Checkpointing libraries, Network monitoring systems, math libraries, resource managers and parallel applications have been FTB-enabled. Any of these middleware can communicate through the backplane to publish and/or subscribe to fault information. This section discusses the applications of FTB-IPMI service in the context of these FTB-enabled software.

The Message Passing Interface (MPI) is one of the most important programming models in high-performance computing. MVAPICH2, MPICH2, and OpenMPI are three of the most popular MPI implementations that heavily dominate the high-performance computing space. All these MPI libraries have incorporated FTB into their stack. MVAPICH2 and OpenMPI have support for preemptive process migration which relies on a prediction of node failure. On subscribing to events from FTB-IPMI, these libraries can predict impending failures and trigger the migration protocol, thereby providing resiliency to the end-user application.

BLCR, the Berkeley Lab Checkpoint/Restart library for Linux, is the one of the most prominent software available for system-level checkpointing. Several high-level libraries including MPI use it to ensure a certain degree of fault tolerance in their software. BLCR has adopted FTB to propagate failure information and other checkpointing-related events.

The FT-LA software package is a dense linear algebra library that features algorithm-based fault-tolerant routines. The integration of FTB in FT-LA encompasses an event schema that includes events such as the ability to recover from failures, time to completion depending on the number of available spare resources should a failure occur, and notification of successful or unsuccessful recoveries. When a node-failure is predicted using data from FTB-IPMI, the FT-LA library can transfer the complete resource dataset to a reserve node pointed to by the resource manager or MPI library.

SLURM is a popular, open-source resource manager and job scheduler developed by Lawrence Livermore National Laboratory. SLURM is FTB-enabled by means of a new notifier plugin to announce events to FTB. Notifications related to monitoring of resources, scheduling of jobs, and failure events internal to SLURM are supported. The SLURM controller daemon, `slurmctld`, publishes these events to FTB through its various hooks using the notifier plugin. FTB-aware components interested in these events can thus track resource changes, job status and SLURM failures. FTB-IPMI and SLURM can benefit from each other by correlating events from their respective namespaces before predicting a node-failure.

VI. Related Work

There are several services and tools that use IPMI to monitor node health. Ganglia [17] and Nagios [18] are two tools that are used widely for node health monitoring in HPC and Grid computing systems. Ganglia is a scalable distributed monitoring system, where each node monitors itself and sends the data to all other nodes. OVIS 2 [19] is a hierarchical monitoring and analysis tool that collects system health information directly from nodes or from other monitoring solutions, such as Ganglia, for processing using statistical methods for graphical presentation. The FTB-InfiniBand monitoring software (FTB-IB) [20] publishes fault information related to InfiniBand adapter availability/unavailability, activation status of InfiniBand ports, status of InfiniBand adapter local ids and protections keys, and information on subnet manager changes as events to the FTB framework. Although there are several related tools and services, FTB-IPMI is the first of its kind that provides a framework for fault-monitoring and prediction using the Fault-Tolerance Backplane and Intelligent Platform Management Interface technologies. Supermon [21] is another monitoring system for Linux systems, which acts as a performance monitoring server that queries individual compute nodes and gathers the data, thereby minimizing the number of queries that are directly sent to the compute node. The Cluster Systems Management (CSM) [22] tool that is designed for AIX systems enables low-cost management of distributed and clustered IBM power systems by providing a single point-of-control that allows fast responses and consistent policies, updates and monitoring by a small staff.

There is constant research in identifying accurate failure-prediction methodologies. Libby et al. talk about the different features provided by IPMI and discuss how system software can leverage these features to predict failures [23]. Leangsuksun et al. have developed a fault-monitoring system [24] as part of the Open Source Cluster Application Resources (OSCAR) using the Hardware Platform Interface (OpenHPI) [25]. There have also been efforts to predict system failures based on system logs [26]–[28] and Support Vector Machines [29].

Several designs have been proposed in literature for preemptive migration. The MVAPICH2 and OpenMPI MPI implementations have process migration support, based on the Berkeley Lab Checkpoint Restart (BLCR) library. The authors of the work discussed in [30] classify preemptive migration into different categories based on the monitoring capabilities available to the compute nodes. Virtual machine migration using VMM-bypass is also available in the Xen framework [31].

VII. Conclusion and Future Work

In this paper, we have presented the design and implementation of FTB-IPMI, a light-weight service for HPC clusters, which aids in hardware fault-monitoring and fault-information dissemination. In addition to this service, we have also proposed a portable rule-based fault-prediction engine that can be adopted by any FTB-enabled system software to assist preventive fault-tolerance protocols. Experimental results clearly show that the service is scalable and uses minimal system resources during its operation. As part of future work, we plan to enhance the rule-based prediction engine within MVAPICH2 library to reduce false predictions by taking historical information from system logs into account. We also plan on adding support to correlate events from other FTB-enabled components like FTB-IB and job-schedulers to increase the accuracy of failure-predictions. With FTB being supported in a variety of other system architectures such as IBM BlueGene(P and L) and Cray, we would like to explore the possibility of porting FTB-IPMI to these HPC systems.

Acknowledgments

The authors would like to thank Mr. Mark Arnold, our Systems Manager, for his assistance during the development and testing of FTB-IPMI, the anonymous reviewers for their constructive suggestions, and the members of the CIFTS team for valuable inputs and discussions during the design phase of this tool.

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749, #DE-FC02-06ER25755; National Science Foundation grants #CCF-0621484, #CCF-0702675, #CCF-0833169, #CCF-0916302 and #OCI-0926691; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, NVIDIA and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Obsidian, Advanced Clustering, Appro, QLogic, NVIDIA and Sun Microsystems.

References

[1] “The InfiniBand Architecture,” www.infinibandta.org.
[2] I. R. Philp, “Software Failures and the Road to a Petaflop Machine,” *Workshop on High Performance Computing Reliability Issues (HPCRI)*, 2005.
[3] “CP2K Molecular Dynamics,” http://cp2k.berlios.de/.
[4] “CPMD Ab-Initio Molecular Dynamics,” http://cpmd.org.
[5] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kal, and K. Schulten, “Scalable molecular dynamics with NAMD,” *Journal of Computational Chemistry*, 2005.
[6] “LS-DYNA Finite Element Software,” www.lstc.com/lsdyna.htm.
[7] J. Peng, J. Lu, K. H. Law, and A. Elgamal, “ParCYCLIC: Finite element modeling of earthquake liquefaction response on parallel computers,” in *International Journal for Numerical and Analytical Methods in Geomechanics*, 2004.
[8] “CIFTS Initiative,” www.mcs.anl.gov/research/cifts.
[9] “MVAPICH2: High-Performance MPI over InfiniBand, iWarp and RoCE,” http://mvapich.cse.ohio-state.edu/.
[10] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, “CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems,” *ICPP*, 2009.
[11] “Intelligent Platform Management Interface Specification v2.0.”
[12] “OpenIPMI,” http://openipmi.sourceforge.net/.
[13] “GNU FreeIPMI,” www.gnu.org/software/freeipmi.
[14] A. Arredondo, P. Roy, and E. Wofford, “Implementing PWM fan speed control within a computer chassis power supply,” in *Applied Power Electronics Conference and Exposition, 2005. APEC 2005. Twentieth Annual IEEE*, 2005.
[15] J. Steele, “ACPI thermal sensing and control in the PC,” in *Wescon/98*, 1998.
[16] J. Stone and M. Underwood, “Rendering of Numerical Flow Simulations Using MPI,” in *Proceedings of the Second MPI Developers Conference*, 1996.
[17] M. L. Massie, B. N. Chun, and D. E. Culler, “The Ganglia Distributed Monitoring System: Design, Implementation And Experience,” *Parallel Computing*, 2003.
[18] “Nagios Infrastructure Monitoring,” www.nagios.org.
[19] J. Brandt, B. Debusschere, A. Gentile, J. Mayo, P. Pebay, D. Thompson, and M. Wong, “Ovis-2: A robust distributed architecture for scalable RAS,” in *IEEE International Symposium on Parallel and Distributed Processing*, 2008.
[20] “FTB-IB: Infiniband Monitoring Software,” www.mcs.anl.gov/research/cifts/docs/files/ftb_api_05_specification.pdf.
[21] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, “Supermon: High-performance monitoring for Linux clusters,” *Proceedings of the 5th Annual Linux Showcase and Conference*, 2001.
[22] “Cluster Systems Management,” http://www-03.ibm.com/systems/software/csm/.
[23] R. Libby, “Effective HPC hardware management and Failure prediction strategy using IPMI,” *Proceedings of the Linux Symposium*, 2003.
[24] C. Leangsuksun, T. Liu1, T. Rao, S. L. Scott, and R. Libby, “A Failure Predictive and Policy-Based High Availability Strategy for Linux High Performance Computing Cluster,” in *5th Linux Cluster Institute Conference*, 2004.
[25] Dague and Sean, “OpenHPI: An Open Source Reference Implementation of the SA Forum Hardware Platform Interface,” in *Service Availability*, 2005.
[26] J. Thompson, D. Dreisigmeyer, T. Jones, M. Kirby, and J. Ladd, “Accurate fault prediction of BlueGene/P RAS logs via geometric reduction,” in *Dependable Systems and Networks Workshops (DSN-W)*, 2010.
[27] M. Shatnawi and M. Ripeanu, “Failure Avoidance through Fault Prediction Based on Synthetic Transactions,” in *Cluster, Cloud and Grid Computing (CCGrid)*, 2011.
[28] Z. Zheng, Z. Lan, B. Park, and A. Geist, “System log pre-processing to improve failure prediction,” in *Dependable Systems Networks, 2009. DSN ’09*, 2009.
[29] E. W. Fulp, G. A. Fink, and J. N. Haack, “Predicting computer system failures using support vector machines,” in *Proceedings of the First USENIX conference on Analysis of system logs*, 2008.
[30] C. Engelmann, G. Vallee, T. Naughton, and S. Scott, “Proactive Fault Tolerance Using Preemptive Migration,” in *Parallel, Distributed and Network-based Processing*, 2009.
[31] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda, “Nomad: migrating OS-bypass networks in virtual machines,” in *Proceedings of the 3rd international conference on Virtual execution environments*, 2007.