RC25468 (WAT1405-042) May 15, 2014
Computer Science

IBM Research Report

A System Software Approach to Proactive Memory-Error Avoidance

Carlos H. A. Costa, Yoonho Park, Bryan Rosenburg, Chen-Yong Cher, Kyung Dong Ryu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

A System Software Approach to Proactive Memory-Error Avoidance

Carlos H. A. Costa   Yoonho Park   Bryan Rosenburg   Chen-Yong Cher   Kyung Dong Ryu
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
{chcost, yoonho, rosnbrg, chenyong, kryu}@us.ibm.com

***Abstract*—Today's HPC systems use two mechanisms to address main-memory errors. Error-correcting codes make correctable errors transparent to software, while checkpoint/restart (CR) enables recovery from uncorrectable errors. Unfortunately, CR overhead will be enormous at exascale due to the high failure rate of memory. We propose a new OS-based approach that proactively avoids memory errors using prediction. This scheme exposes correctable error information to the OS, which migrates pages and offlines unhealthy memory to avoid application crashes. We analyze memory error patterns in extensive logs from a BG/P system and show how correctable error patterns can be used to identify memory likely to fail. We implement a proactive memory management system on BG/Q by extending the firmware and Linux. We evaluate our approach with a realistic workload and compare our overhead against traditional CR. We show that our approach increases application resiliency without introducing significant overhead and allows checkpointing requirements to be greatly relaxed.**

I. Introduction

Frequent system failure due to an increased number of memory errors is a major obstacle to scaling current HPC technology [1]. These errors require the system to be rebooted and any running applications to be restarted, either from scratch or from previous checkpoints [2]. Error-correcting code (ECC) is the traditional approach to addressing memory errors. ECC uses extra, redundant memory cells to detect, and in many cases to correct, memory errors before the errors lead to memory corruption and system failure. However, as systems scale up, unprecedented transistor and component density, aggressive power efficiency requirements, and new fabrication techniques will impose new challenges to system reliability that ECC may not be able to address.

Memory failures are caused by soft or hard errors. Soft errors are transient errors caused by external particles such as neutrons and alpha particles. These particles can flip bit values in the transistors. Hard errors are permanent, recurring errors caused by wear-and-tear and aging of transistors and metals. They are characterized by phenomena such as electro-migration, voltage drop, Negative- or Positive-Biased Temperature Instability (NBTI/PBTI), Hot Carrier Injection (HCI), Time-Dependent Dielectric Breakage (TDDB), and other deterioration processes. In the next decade, hard errors are expected to be the dominant source of errors while the soft error rate is expected to decrease due to advances in transistor technology such as the transition to FinFET. This trend can already be observed in current systems in which there is a significantly larger than expected number of hard errors compared to soft errors [3], [4].

ECC mechanisms can address both soft and hard errors. Correctable memory errors, those for which the redundant ECC information is sufficient to reconstruct the correct memory content, can be handled without restarting the system or applications. Uncorrectable errors are those that are detected by the ECC mechanism but for which the redundant information is insufficient for correction. Uncorrectable errors generally require a system restart. Silent errors, those that escape even detection by the ECC mechanism and result in outright memory corruption, are so damaging that ECC mechanisms are designed to make them extremely rare, even in very large-scale systems. This paper is primarily concerned with ECC-detected memory errors, both correctable and uncorrectable.

While soft errors are intrinsically unpredictable, hard errors can be described as a dynamic function of manufacturing variations, surrounding conditions, and workload phases. Early signs of degradation manifest as a pattern of correctable errors. Correctable errors are typically logged by Reliability-and-Availability Services (RAS) subsystems in HPC systems, but are transparent to upper software layers including the OS and applications. Several studies point to the predictability of memory hard errors [5], [6]. These studies expose the potential benefits in the early identification of *unhealthy* memory. These studies also show how the occurrence of correctable memory errors is concentrated in a small fraction of memory in few nodes. The benefits of proactively avoiding failing memory are discussed in [3]. While the study projects that simple policies guiding the retirement of memory pages would be sufficient to avoid a large number of correctable errors, it does not provide a practical model and mechanism to apply these policies.

There are two main challenges that must be addressed to enable proactive avoidance of failing memory. The first challenge is to enable real-time access to corrected-error information. The second is to efficiently analyze the error information, recognize patterns that indicate likely imminent failures, and take steps to avoid using the failing memory areas. Addressing these challenges requires changes to the typical RAS infrastructure deployed in HPC systems. Non-fatal errors should continue to be captured and logged by the RAS infrastructure but they should also be exposed to the OS. Non-fatal error events must be presented in a way that allows the OS to take proactive actions to avoid fatal errors. The OS can then apply a failure prediction algorithm to the live stream of error events and distinguish between harmless error patterns and error patterns that are likely to lead to failure.

We propose, implement, and evaluate a solution for exploring correctable memory error patterns for proactive memory management within the OS. Our solution consists of a generic and adjustable memory-failure prediction algorithm for DRAM, an OS mechanism implementing failure prediction, and proactive memory management. Our solution is implemented as a working prototype on Blue Gene/Q (BG/Q). In order to develop a prediction algorithm, we used previous studies and publicly available data on failure events in HPC systems. Our algorithm is based on error patterns we identified in raw BG/P RAS logs that were collected from Intrepid at Argonne National Laboratory [7] and cover a period of seven months of real workload. All the data is available in the USENIX Computer Failure Data Repository (CFDR). We describe the steps taken to process the error logs and report on our analysis of memory-related events and on how the analysis led to a generic and adjustable algorithm for DRAM that can be applied to live event streams.

Our prototype consists of changes to the native BG/Q RAS support, a memory health monitor, and changes to the BG/Q Linux kernel. In order to expose detailed information about correctable errors, we changed the mechanisms in the BG/Q firmware used for error logging. Error information can be intercepted and delivered to the OS through a custom interrupt interface. The interface provides access to information about the memory involved in correctable errors including physical addresses and error types. We also implemented an error emulation mechanism at the firmware level. This mechanism can inject errors via the hardware memory controller that are then processed as if they were real errors. The mechanism allows us to evaluate our solution in realistic, repeatable scenarios.

The memory health monitor is responsible for handling interrupts caused by correctable error events, applying the failure prediction algorithm, and triggering mitigation actions if necessary. The mitigation actions involve the migration of data away from, and the retirement of, pages for which failure has been predicted. We use the page soft-offlining feature that exists in Linux but was not previously enabled in BG/Q Linux. The features allows the retirement of a page through non-disruptive migration of data from the page and its removal from mapped memory.

The goal of this paper is two-fold. First, we demonstrate the feasibility of our solution by showing how emulated errors from existing logs can be avoided in a practical setup. We measure the overhead and the effectiveness for different scenarios. We evaluate the number of errors avoided as a function of the amount of memory offlined and the performance overhead due to page migration. Second, we evaluate the benefits of our mechanism for the improvement of overall system resilience to memory failures. We present numbers for Mean-Time-Between-Failure (MTBF) improvement and compare our approach to a checkpoint/restart mechanism for a representative workload. We show that significant resilience improvement can be obtained with negligible overhead compared with traditional mitigation techniques.

The rest of the paper is organized as follows. In Section II we describe how we derived a generic prediction algorithm by analyzing a raw RAS log. We present in detail the method used to process and interpret the logs, along with the spatial and correlation analysis that yielded the steps in the algorithm. In Section III we describe our OS mechanism to predict and avoid memory errors. In Section III-B we describe the

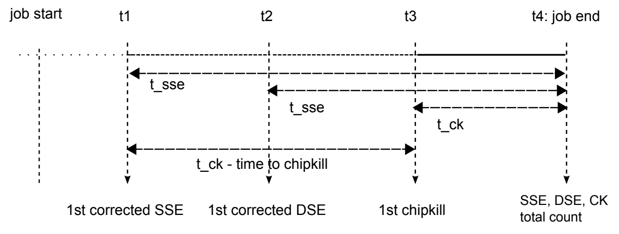

Fig. 1. Correctable memory error reporting format in BG/P.

prototype implementation, including the changes to the BG/Q infrastructure and the enabling and testing of memory page migration/offlining in Linux on BG/Q. In Section IV we present our evaluation of the effectiveness of the proposed solution. In Section V we discuss related work and in Section VI we discuss conclusions and future work.

II. Memory Health and Failure Predictability

Identifying early signs of memory health degradation is necessary to mitigate the impact or avoid a failure. While previous studies show strong statistical evidence of the correlation between correctable errors and failures, little has been done to demonstrate how this information can be exploited in system software. Spatial and temporal distribution of errors and their correlation can be used to separate healthy and unhealthy memory chips. We define unhealthy chips as those with deteriorating memory error patterns. A common progression for an unhealthy chip is multiple single-bit errors, followed by multi-bit and/or non-trivial errors, until an eventual uncorrectable error causes a failure.

Spatial correlation indicates the likelihood of experiencing a failure after seeing a pattern of errors in bits that are near each other physically. For hard errors, the frequency with which a correctable error is experienced is a function of the degradation process and the access pattern. We show how to capture both the degradation process and the access pattern in a simple algorithm, that when applied to a stream of correctable errors can be used to anticipate when memory is unhealthy and a failure is imminent. The failure prediction algorithm is derived from an analysis of error logs from Intrepid, a BG/P system at Argonne National Laboratory. These are publicly available logs used in previous studies and are comparable to error patterns in different systems [3].

Intrepid consists of 40 racks containing a total of 40,960 nodes. Each node has four PowerPC 450 cores and 40 DRAM chips providing 2 GB of addressable memory. ECC can correct single- and double-symbol errors, and it includes Chipkill error correction that can tolerate complete failure of any single DRAM chip by reconstructing data from redundant data pieces kept in the remaining chips [8]. The error correction capabilities and logging formats found in BG/P are similar to those found in BG/Q, the system we use for our prototype. Figure 1 summarizes a sample from a correctable-error report in BG/P. For each job executed, the first correctable error in

TABLE I. Summary of correctable error distribution for nodes and jobs.

Nodes	#	%	Reports/Jobs	#
All	40273		Error reports	68946
w/ memory error	956		Jobs	49500
w/ Chipkill activation	262	27.41		

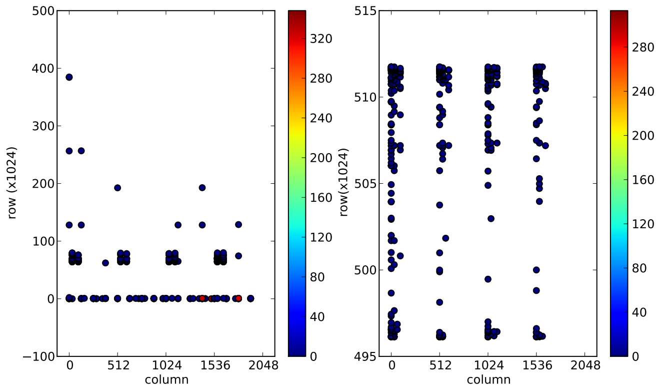

Fig. 2. Example of error address distribution in a memory bank for two nodes activating Chipkill: R34-M0-N13-J24 and R20-M0-N14-J24.

each of the following complexity categories is reported: single-symbol error, double-symbol error, and Chipkill. For each first occurrence, details such as the failing address, DRAM chip, and timestamp, among others, are reported. The total number of occurrences of each type of correctable error is then summarized at the end of the job. In Table I, we show a summary of correctable error occurrences in the BG/P logs. We highlight the incidence of non-trivial error-correction activations (Chipkill), an indication of impending hard errors. Next we show how spatial and temporal characteristics of memory errors can be used in a prediction algorithm.

A. Spatial Correlation

We first show how the spatial distribution of correctable errors can be used to identify nodes with a high chance of developing non-trivial error corrections. We focus on the evolution of error patterns as an indication of predictable memory failure. With respect to the occurrence of correctable errors, two very distinct sets are observed. In one set are nodes that experienced instances of single-symbol errors, but never required a more complex error correction activation such as Chipkill. In another set are nodes that activated Chipkill. While Chipkill is still a correctable error, its repeated occurrence is highly correlated with the occurrence of an uncorrectable error causing a failure (details can be found in Section II-C). The second set of nodes represents 27% of all nodes experiencing correctable errors.

We use spatial error distribution to identify errors that tend to cluster, indicating the development of a faulty memory area. Assuming that such patterns can be related only to hard errors, one approach to identify a faulty area is to identify

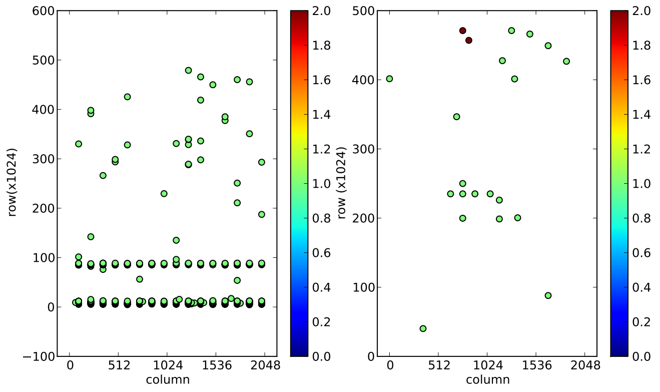

Fig. 4. Example of error address distribution in a memory bank for two nodes not activating Chipkill: R21-M0-N10-J30 and R44-M1-N01-J26.

the repetition of correctable errors in the same address (row and/or column) in a given memory bank. Note that references to "addresses" below are meant to imply the row and column addresses that are relevant for spatial locality.

Figures 2 and 4 show examples from the two distinct node sets, with each failing address plotted in the memory bank space. Each address in the graphs is colored according to the number of repeated error occurrences it has experienced. In Figure 2, there are two examples of nodes that repeatedly activated Chipkill. Conversely, in Figure 4 are two examples of nodes where single-symbol corrections never developed into Chipkills. Most of the nodes that develop Chipkill follow the pattern in Figure 2, which is characterized by repeated error corrections in the same address. Conversely, nodes that do not activate Chipkill experience significantly fewer address repetitions. As proposed in previous studies, the chances of experiencing a Chipkill correction can be obtained as a function of whether correctable errors are observed in the same address. Figure 3 shows the probability of experiencing at least one Chipkill correction after seeing at least one repetition at the same address for all jobs in the error logs. Figure 5 shows the prediction accuracy. While there is some predictability, the sole use of spatial distribution implies a probability no higher than 20% and a prediction coverage of 45%. These numbers are in line with previous studies. Next we show that by combining spatial and temporal error distribution, specifically the correctable error rate, the prediction accuracy can be made significantly higher.

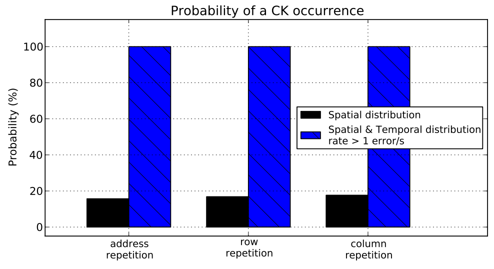

Fig. 3. Probability of observing a Chipkill event based on spatial and temporal distribution of repeated correctable errors for all jobs.

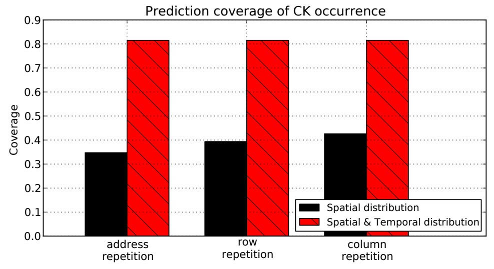

Fig. 5. Coverage of Chipkill prediction based on spatial and temporal distribution of repeated correctable errors for all jobs.

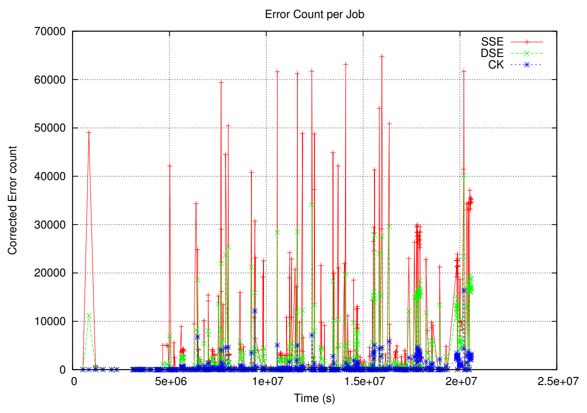

Fig. 6. Example showing correlation of correctable error counts: total error counts for all jobs run on node R04-M1-N01-J17.

B. *Temporal Correlation*

We use timing information from the BG/P logs in two ways. We first calculate for each correctable error category, an error rate based on the first occurrence of an error and the total number of errors reported at the end of each job. We also use error timestamps to measure the average time between the first occurrence of a single-symbol error and a potential Chipkill correction.

For each node, we calculate the error rate for each job, and based on whether the first address report is an address repetition, try to determine whether that specific job developed at least one Chipkill. The right-hand side bars in Figure 3 show the updated probability when spatial and temporal correlation is taken into account. We found that if any kind of repetition is followed by an error rate greater than 1 error/second, all the observed jobs developed Chipkill. However, not all jobs with Chipkill had an early repetition and an error rate greater than 1 error/second. The right-hand side bars in Figure 5 show the updated coverage when error rate is taken into account. While this analysis does not guarantee full coverage, it shows that the observation of the first occurrence of a correctable error and the frequency of any errors that follow allow the anticipation of complex error activation with high accuracy.

Figure 3 shows the probability of seeing at least one Chipkill correction per job executed. While one instance of this non-trivial error correction pattern is already an indication of a complex error scenario, its constant repetition can be associated with a high probability of future uncorrected error. A repetition of Chipkill corrections is typically preceded by a repetition in single-symbol errors and double-symbol errors. Figure 6 shows the error counts for single-symbol errors, double-symbol errors, and Chipkills for all jobs run on one of the nodes in Intrepid. In this example, most of the jobs with a high Chipkill count are also preceded by a high number of single- and double-symbol errors. Calculating Pearson's coefficient for the correlation between double-symbol errors and Chipkill rates, we found that the error rates are fairly correlated, with a coefficient of 0.88. With the steps in the previous analysis, we are able to predict whether a Chipkill correction will occur when we start a job. The correlation between double-symbol errors and Chipkill lets us predict the cases where repeated Chipkills are more likely.

A question that remains is the time between the first correctable error and a Chipkill. This determines how much time there is for proactive action to be taken, and when it should be taken to avoid multiple and complex errors leading to failure. Knowing the time of the first single-symbol error and the first Chipkill, this distribution can be calculated for all nodes and jobs with Chipkills. Table II shows the time distribution. It reveals that in the majority of cases (63%), the first Chipkill is experienced within one minute of the first single-symbol error. When almost all jobs execute for longer than a minute, it is clearly advantageous to take actions to avoid failures while jobs are still running.

TABLE II. TIME ELAPSED BETWEEN FIRST SINGLE-SYMBOL ERROR OBSERVED AND FIRST CHIPKILL CORRECTION.

t < 1s (%)	**1s < t < 1m (%)**	**1m < t < 1h (%)**	**1h < t < 1 day (%)**
26.21	63.57	9.51	0.69

C. *Chipkill and Uncorrected Errors*

In our analysis we focus on anticipating repeated non-trivial corrected errors that result in Chipkill. While this type of error is still correctable, its repetition has been shown to be a clear sign of future failure. As shown in previous studies, the presence of a multiple-bank or multiple-rank fault increases the probability of experiencing an uncorrected error by 350x and 700x, respectively [4]. The majority of nodes with uncorrected errors, 83%, first experienced repeated corrected errors from an existing fault. In our analysis, there were no instances of uncorrected memory failures that were not preceded by at least one correctable error.

The anticipation and avoidance of repeated non-trivial error correction has multiple benefits. It first allows us to avoid faulty memory before it is too late to prevent a failure. On the other hand, the repeated activation of a costly and complex error correction can be seen as an efficiency problem in itself. In fact, due to its high area and power overhead, avoiding Chipkill correction in some cases has significant benefits. In the next section, we show how the analysis presented in this section can be captured in a memory-failure prediction algorithm and how we incorporated it into an OS.

III. PROACTIVE MEMORY MANAGEMENT

In this section we propose an operating system mechanism to monitor, anticipate, and proactively mitigate the impact of predictable memory failures. Our approach consists of monitoring memory degradation and reallocating memory proactively to avoid faulty memory. Our goal is to dynamically migrate data residing in memory pages that are becoming faulty without disrupting running processes. We transparently avoid unhealthy memory before it causes a failure, allowing nodes to keep running in the presence of faulty memory.

A. *Design*

The analysis in the previous section can be applied in real-time to predict the probability of an uncorrectable error. A mechanism that dynamically keeps track of addresses reported in correctable errors can determine whether any given error is part of a spatial repetition. When a repetition is detected, the monitoring of past correctable errors can then be used to calculate an error rate. These steps can be applied in

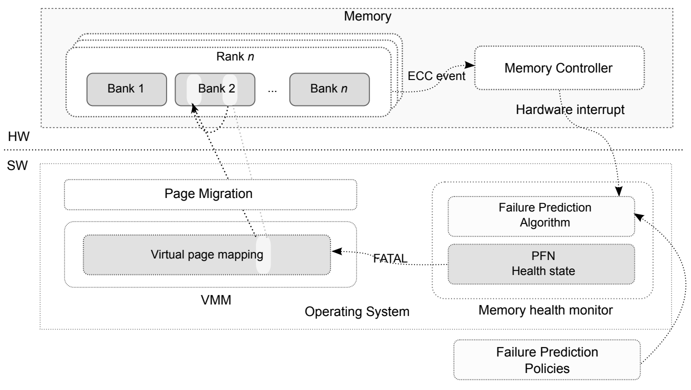

Fig. 7. Memory health monitoring mechanism.

an algorithmic fashion, triggering mitigation actions when certain situations are encountered. This model is intrinsically parametric as error rate and repetition policy can be adjusted considering accuracy, coverage, and the cost of false positives.

Figure 7 illustrates our approach. A health tracking module is responsible for interpreting reports of correctable errors, applying a prediction algorithm, and triggering a page migration action upon the identification of unhealthy memory. The monitor acts as a high-level interrupt handler for interrupts generated by correctable errors. These interrupts are silent machine checks generated by the memory controller, or any kind of custom interrupt interface for notification of correctable errors. In our approach, every interrupt that is the result of a correctable error is handled by the health monitor.

In our design, we identify faulty memory with the granularity of page frame numbers (PFNs). This approach (detailed later) facilitates the implementation of page migration. A PFN is the physical address divided by the physical page size. The memory health monitor applies an algorithm derived from the analysis of memory health degradation and marks memory pages according to specified policies. These policies refer to the number of spatial repetitions and the error rate threshold that are associated with an imminent failure. All error events, including those detected through self-testing mechanisms such as memory scrubbing, are captured. The target pages for migration are selected from those pages that have not experienced any previous error, which indicates the target pages are healthy. Figure 8 summarizes the algorithm used by the memory health monitor.

For every interrupt, we calculate the PFN of the reported address. We track the health state of every page that has at least one correctable error. We define three states: healthy, unhealthy, and fatal. The algorithm relies on two hash tables for tracking reported addresses and PFNs as shown in Figure 9. In the address table, every address has a corresponding error count. In the PFN table, every page has a last error time, an error rate, and a health state. When an interrupt occurs, we calculate the PFN from the address. If this is the first error for the page, we add the PFN to the PFN table and the address to the address table. If this is not the first error for the page, we check if the address is a repetition by looking up the address in the address table. If the address is a repetition (steps 11–18), we calculate the error rate using the last occurrence of an error. If the error rate threshold specified by the policy is crossed, the health state of the page is updated to fatal. Otherwise, the

```
Require: physical address, last occurrence of error, error type
 1: addr ← physical address
 2: Get PFN for addr
 3: Search PFN in pages hash table (PHT)
 4: if PFN ∉ PHT then
 5:     Get time and error type
 6:     Add PFN to PHT
 7:     Add addr to address hash table (AHT)
 8: else
 9:     Search addr in AHT
10:     if addr ∈ AHT then
11:         Increment current repetition count
12:         Get current time and last occurrence in PFN
13:         Calculate error rate $E_r$
14:         if $E_r >$ error threshold then
15:             Update health state of PFN to fatal
16:         else
17:             Update health state of PFN to unhealthy
18:         end if
19:     else
20:         Update health state of PFN to healthy
21:         Add addr to AHT
22:     end if
23: end if
```

Fig. 8. Algorithm for failure prediction.

health state is updated to unhealthy. If the address is not a repetition (steps 20-21), the health state is updated to healthy.

Updating the health state of a page to fatal triggers a page migration action. This action relies on a specific implementation for memory allocation and support for dynamic memory remapping. In the next section, we describe how we implemented this in our prototype.

B. Implementation and Prototype

We implemented a prototype in the Linux kernel that runs on BG/Q nodes. A BG/Q node has 17 cores, each with 4 hardware threads, and 16 GB of physical memory [9], [10]. A BG/Q system consists of I/O and compute nodes. Compute nodes usually run a lightweight kernel called the Compute Node Kernel (CNK), but for this work we used a version of the Linux 3.4 kernel that has been ported to run on both I/O and compute nodes and is the basis for a hybrid Linux/CNK research effort [11]. The kernel is available at [12].

There are four major components in our prototype. We first reconfigured the BG/Q interrupt controller so that correctable memory errors are presented as non-critical interrupts rather

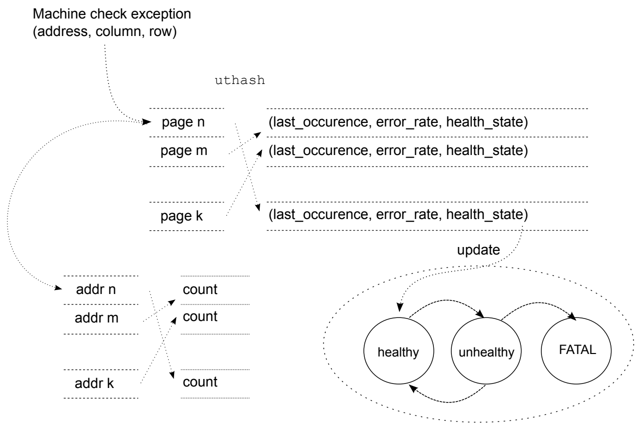
Machine check exception (address, column, row)
uthash
page n
page m
page k
(last_occurence, error_rate, health_state)
(last_occurence, error_rate, health_state)
(last_occurence, error_rate, health_state)
update
addr n
addr m
addr k
count
count
count
healthy
unhealthy
FATAL

Fig. 9. Mechanism to monitor the health states of memory pages.

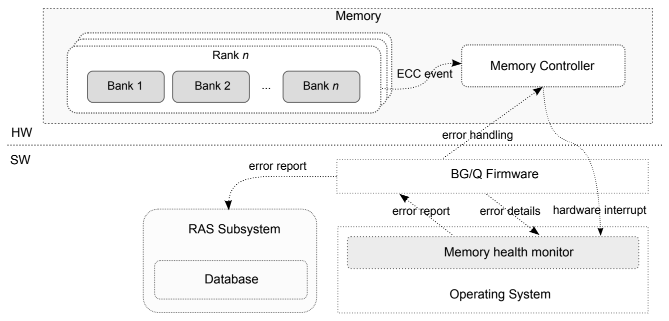

Fig. 10. BG/Q firmware modifications for health monitoring.

than as machine checks, in order to expose correctable errors to Linux. Second, we added a custom interrupt handler that retrieves error details in cooperation with the firmware, applies the failure prediction algorithm, and triggers page migration and offlining when the health state of a page changes to fatal. Third, we enabled memory hotplug support in the Linux kernel we are using in our experiments. Fourth, for validation and evaluation purposes, we implemented an error injection mechanism to replay error patterns. The injection mechanism consists of modifications to the firmware and the firmware interface. New firmware routines are used to set error condition bits, which trigger interrupts, in failure isolation registers. These new firmware routines can be called from the OS. Below we detail the implementation of each of these components.

1) Interrupt Controller Reconfiguration: Ordinarily, BG/Q firmware handles correctable errors transparently. Errors encountered by the hardware generate machine check exceptions that are handled by the firmware, which retrieves details from the hardware. The firmware processes, aggregates, and interprets error conditions. Error reports are delivered to the control system and logged to a database. BG/Q Reliability-and-Availability Services (RAS) uses a hard-coded threshold for correctable errors. When this threshold is crossed, an unrecoverable machine check is generated. The OS is interrupted and terminates in a fatal condition when an unrecoverable error is detected or when the correctable error threshold is reached.

We reconfigured the BG/Q interrupt controller to divert and expose correctable errors. There are two levels of interrupt controllers in Blue Gene/Q. There is one Global Event Aggregator (GEA) and 17 Processing Unit Event Aggregators (PUEAs), also called Blue Gene Interrupt Controllers (BICs). There is one PUEA/BIC per core. Our health monitor requires all correctable errors to be exposed to Linux. We reconfigured the PUEA/BIC to present memory errors as non-critical interrupts, handled by Linux, rather than as machine checks, handled by firmware. Although memory errors are now handled by Linux, we still use the firmware routines that retrieve error details and reset hardware mechanisms involved in their reporting. We also invoke the original firmware handler for memory errors so that the errors are reported to the native control system via the standard RAS infrastructure.

2) Interrupt Handler and Health Monitor: Figure 10 illustrates our implementation. Every correctable error generates a non-critical interrupt, which is then handled by our interrupt handler. To collect error details, our handler calls the firmware's memory-error handler, which is normally invoked from the machine check handler. This provides details such

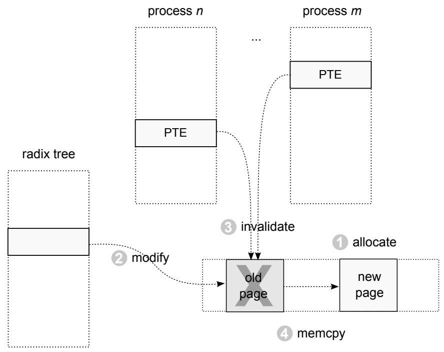

Fig. 11. Page migration in the Linux kernel.

as the type of correctable error, physical address, and timing. As a side-effect of calling the firmware handler, the error is reported to the RAS infrastructure and logged to a database.

We apply the algorithm for failure prediction every time an interrupt is handled. We implemented the algorithm using uthash code [13]. The hash tables grow dynamically, with memory for new items allocated with the GFP_ATOMIC flag. For every error encountered, the handler is called, resulting in an update of the health state of each memory page. When a memory page is moved to the fatal state, a page migration is triggered.

3) Page Migration in Linux: Memory hotplug is a recent feature in Linux that has been available since version 3.2. Physical memory can be added/removed by adding/removing DIMMS. Logical memory can be added/removed through onlining/offlining. Memory hotplug allows the dynamic update of the page table and virtual memory maps. We rely on memory hotplug for its memory offline feature, allowing us to migrate data and avoid the use of faulty memory.

Memory hotplug uses a non-linear memory model, SPARSEMEM, which allows Linux to handle discontiguous memory. We enabled SPARSEMEM for PowerPC/Book3E, making the memory offline feature available for the BG/Q A2 processor. The memory offline feature provides page *soft-offline* and page *hard-offline*. With page soft-offline, suspicious pages can be dynamically offlined without killing processes using the page. With page hard-offline, pages are offlined and processes using the pages are killed. Hard-offlined pages are also included on a bad page list and retired. Memory offline is performed at a kernel-specific granularity, which is usually the base page size.

We call the soft-offline routine when the health state of a page is updated to fatal. The kernel will then attempt to migrate the contents of the page elsewhere or drop the page if possible. Offlined pages are placed on a bad page list and never reused. Figure 11 illustrates the page migration process. Page migration is implemented in the following steps: (1) allocate a new page, (2) modify the radix tree, (3) invalidate the Page Table Entry (PTE) descriptors pointing to the old page, and (4) copy page contents from the old page to the new page. In the current implementation, not all pages are migratable. Structures such as kernel/text/stack and kmalloc() are examples of un-removable memory.

4) Error Emulation Through BG/Q Firmware: To evaluate our approach, we implemented an error emulation mechanism

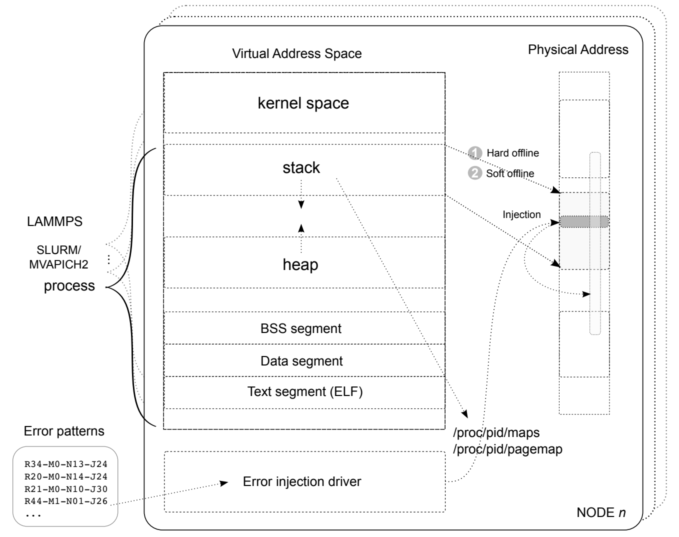

Fig. 12. Approach for virtual-physical address mapping considering the worst-case scenario of errors affecting the application stack.

in which simulated errors are indistinguishable, as far as the OS is concerned, from real errors. Such an emulation mechanism allows us to replay known error patterns, like those found in BG/P logs, and evaluate the behavior of our memory health monitoring system.

We added new routines to BG/Q firmware that force the hardware to report various types of memory errors. We do so by selectively setting bits in the Machine Check Failure Isolation Registers (MCFIRs). These bits are normally set by hardware when an ECC event occurs. By manually setting them, we force the hardware to react, i.e., record error details and generate a machine check exception (or a non-critical interrupt, in our case). This means that as for a real error, the custom interrupt interface is activated when an error is emulated. We also have the capability to set the details about an error. When the hardware drills down for detailed information it will find the specified error type and address we injected. While no actual manipulation has been done to data in memory, the emulation approach makes the hardware react realistically.

We created a Linux sysfs interface to allow a daemon to drive error injection. In this approach, injection routines can be called from userspace and errors can be emulated as other applications run. In the next section, we describe how we use this approach to evaluate our mechanism.

IV. Experimental Evaluation

We evaluated our mechanism and implementation by measuring the performance overhead it imposed on applications and by estimating the overall resilience improvement it accomplished. To demonstrate feasibility, we showed that critical memory pages can be migrated dynamically without disrupting applications. Costs are evaluated in terms of variation in application running times when memory pages are migrated and of the amount of memory offlined. The overall resilience improvement was measured in terms of effectiveness in avoiding failures, and we compared our results against traditional checkpoint/restart support.

A. Benchmark and Parallel Execution Environment

We used LAMMPS from the ASC Sequoia Benchmark Codes [14] in our evaluation. The benchmark mimics "particle-like" systems such as molecular dynamics simulations. We used the 3D Lennard-Jones melt benchmark script from LAMMPS version 1Feb14. In order to adjust running time and memory footprint, we varied the number of iterations and problem size (reflected in the number of atoms). The benchmark has built-in support for scalable checkpoint/restart, which allowed us to make a direct comparison with an efficient, existing mechanism.

The benchmark aims to predict parallel efficiency for large systems. In LAMMPS, several processes are run on each multicore node in a hybrid manner, each process using OpenMP to exploit multiple threads and coordinating with other processes via MPI. In our experiments we ran the benchmark both on a single node with OpenMP and multiple nodes with MPI support. For multiple nodes, we deployed SLURM (Simple Linux Utility for Resource Management) [15] across nodes running our modified Linux. For process coordination, we rely on MVAPICH2 (MPI-3 over OpenFabrics-IB, OpenFabrics-iWARP, PSM, and TCP/IP) [16] on top of SLURM.

B. Error Injection and Reproduction of Error Patterns

We developed an error injector that reads the BG/P logs and replays the error pattern for each node following time and spatial distribution. The implementation allows time to be accelerated given a specified time scale. For our experiments, we use a time scale of 1:1200 seconds, allowing error patterns that span months to be reproduced in a time frame of hours. In this way, we evaluated a worst case scenario, which we take into consideration in our conclusions and comparisons. When reproducing errors, we used the following four cases:

Base: Idealistic case in which no errors are experienced. The error monitoring and migration mechanism is never activated in this case.

Case 1: Node with unhealthy memory with several and widespread repeated errors, including repeated Chipkill correction. This pattern is correlated with a high probability of failure. The error pattern in node R20-M0-N14-J24 (Figure 2.a) is an example of this case.

Case 2: Node with unhealthy memory with several errors concentrated in few rows. The error pattern in node R34-M0-N13-J24 (Figure 2.b) is an example of this case.

Case 3: Node with healthy memory, with early signs of degradation. Correctable errors are experienced, but no repetition is observed. The error pattern in node R21-M0-N10-J30 (Figure 4) is an example of this case.

C. Survivability

We first show that a node and running application survive when critical memory is migrated. Figure 12 illustrates our approach. Once started, the error injector first finds the virtual address range used by the stack of a specified running process. We use the stack in an attempt to evaluate a disturbance in a highly and frequently used memory area. The injector uses the

TABLE III. RESOURCE UTILIZATION OF LAMMPS IN A SHORT SINGLE-NODE (4 OPENMP THREADS) RUN WITH PAGE MIGRATION.

scenario	real	user	sys	sys incr. (%)
baseline	23m 26.18s	1h 33m 40s	0m 3.02s	
case 1	23m 27.55s	1h 33m 42ss	0m 3.63s	**20.20**
case 2	23m 27.83s	1h 33m 45s	0m 3.66s	**21.19**
case 3	23m 26.22s	1h 33m 40s	0m 2.96s	**-1.98**

TABLE IV. RESOURCE UTILIZATION OF LAMMPS IN A LONG SINGLE-NODE (4 OPENMP THREADS) RUN WITH PAGE MIGRATION.

scenario	real	user	sys	sys incr. (%)
baseline	3h 55m 05s	15h 39m 41s	31.54s	
case 1	3h 55m 05s	15h 39m 41s	32.05s	**1.62**
case 2	3h 55m 06s	15h 39m 46s	30.73s	**-2.56**
case 3	3h 55m 06s	15h 39m 47s	31.56s	**0.06**

information provided by Linux through `/proc/pid/maps` and `/proc/pid/pagemap`. The virtual memory region used by the stack is retrieved from `maps`. The physical page frame that each virtual page is mapped to is then retrieved from `pagemap`. The injector then iterates over the address range and finds a dirty mapped memory page. It injects the first error in the error pattern to this address and subsequent errors by maintaining the spatial and time distribution.

In a first step, we first proved the criticality of pages found through this process. For LAMMPS running on a single node, we *hard-offline* the first page identified. If the kernel hard-offlines a page that is being used, its owner will be killed. Our action consistently caused the kernel to kill LAMMPS. In the second step, we *soft-offline* the first page identified. When the kernel soft-offlines a page the contents of the page are migrated to a new page transparently. LAMMPS survived the soft-offline of the first page identified.

In a parallel execution with multiple nodes, independent instances of the injector run on each node (Figure 12). They reproduce different error patterns from the logs, according to a specified selection criteria. Each process started by SLURM for the parallel execution of LAMMPS across the nodes is found and the same steps applied. In a parallel execution, the failure of a node halts the whole execution.

D. Performance Overhead

We evaluated the impact of tracking errors and performing page migration on resource utilization and execution time. In our experiments, based on the analysis of prediction accuracy and coverage in Section II, we offline any page that has an error address repetition with a frequency greater than 1 error per second. The implementation of the migration mechanism is non-blocking, meaning that other threads are able to make progress while the OS is offlining a memory page. Then we investigate how the individual cost per migration translates into variations in resource utilization and completion times for runs with different sizes on both single and multiple nodes.

1) Single-Node Scenario: We collected system and user times taken by a single-node run of LAMMPS for each of the cases. In order to have a first short run, we reduced the number of iterations in the input script. Table III shows the variation in resource utilization in a short run (<45 minutes, with a memory footprint of 20 MB), running with 4 OpenMP threads. User time in the table is the sum of times taken by multiple threads. During the execution, 451 errors were injected. The table reveals a negligible variation in the real times taken for completion, while system activity increased (≈ 20% in cases 1 and 2), showing the additional work the OS performs to interpret errors and migrate pages. Since the migration operation is non-blocking, the impact to user time is negligible. An analysis of the number of pages migrated and the amount of memory taken is shown in detail in the next section.

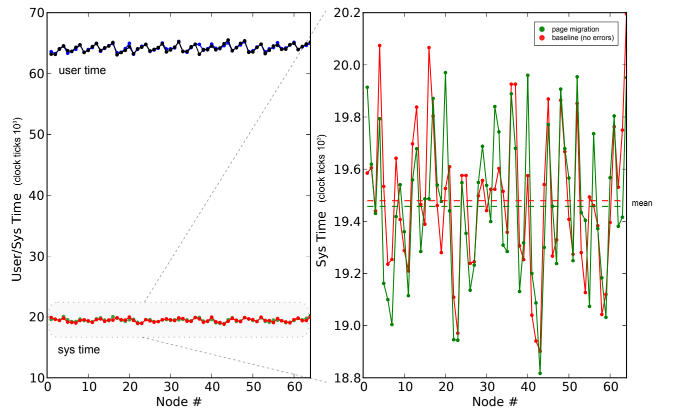

Fig. 13. Resource utilization across multiple nodes: baseline and page migration comparison for short LAMMPS run on 64 nodes.

TABLE V. SUMMARY OF RESOURCE UTILIZATION ACROSS MULTIPLE NODES: BASELINE AND PAGE MIGRATION COMPARISON FOR LONG RUN OF LAMMPS ON 64 NODES, 6 DIFFERENT RUNS.

	real time	incr. (%)	mean. user time (ticks 10^3)	incr. (%)	mean sys time (ticks 10^3)	incr. (%)
baseline	293m46.131s		1736.48		8.424	
avg. with page migration	292m58.317s	**-0.27**	1733.74	**-0.16**	8.437	**0.14**

Table IV shows resource utilization for a longer run (4 hours). In this time frame, thousands of errors are handled by the OS. As in the short run, the impact on real and user times is negligible. In this case, however, the variation in system utilization also becomes negligible. The main reason for this is that after paying the initial cost of migrating unhealthy pages, page migration activity decreases as the application runs and the final share of total system activity becomes negligible. This happens due to the clustering of memory errors, as discussed before, and the fact that a few unhealthy pages are responsible for the majority of subsequent errors, which are avoided as faulty memory is taken offline.

2) Multi-Node Scenario: In a multi-node environment, we evaluate whether a parallel run survives and how the overall running time is impacted by page migration on each node. In this scenario, LAMMPS runs in parallel on 64 nodes (256 procs, 64 MPI x 4 OpenMP), in 6 independent runs. Each node has an independent instance of the injector. The problem size (number of atoms) was adjusted to increase memory usage to ≈ 200 MB per processor. The injector randomly selects, for each run, an error pattern from the logs of nodes with Chipkill (269 in total). Figure 13 shows the resource utilization for each node in our environment. The baseline scenario, where no errors are experienced, is compared against the scenario where the page migration mechanism is active.

TABLE VI. EFFECTIVENESS AND MEMORY OVERHEAD WITH PAGE MIGRATION: SINGLE-NODE SHORT RUN.

scenario	SSE avoid.(%)	DSE avoid.(%)	CK avoid.(%)	memory retired	% of app. memory
case 1	27.3	23.9	**71.5**	512 KB	2.52%
case 2	86.69	91.97	**91.87**	832 KB	4.10 %
case 3	0	n/a	n/a	0	0

TABLE VII. EFFECTIVENESS AND MEMORY OVERHEAD WITH PAGE MIGRATION: 64 NODES, 6 DIFFERENT RUNS.

scenario	SSE avoid.(%)	DSE avoid.(%)	CK avoid.(%)		memory retired	% of app. memory
best average	76.34	71.42	**80.42**	max.	1152 KB	0.6
worst average	71.28	63.39	**74.80**	min.	64K KB	0.03
avg. run	76.07	68.65	**76.42**			

The left-hand side shows the user and system times. The right-hand side shows system time in more detail. Table V summarizes the variation observed in resource utilization in multiple long runs (5 hours each). A very small variation in user and system time is observed, which considering its magnitude can be better explained by other sources of noise than by a disturbance caused by page migration. On average the change in system and user times are 0.14% and -0.16%, respectively, meaning that page migration does not introduce any measurable overhead. This is also observed in the real times. The total wall-clock time taken to complete the parallel run is almost the same with page migration (a variation of -0.27%). These results show that for a parallel run several other sources of overhead such as I/O and synchronization make the cost of page migration insignificant.

E. *Effectiveness*

The effectiveness of the page migration mechanism is evaluated in terms of the number of repeated errors (specifically Chipkills) that are avoided and the amount of memory that is taken offline. We calculate the number of errors avoided by counting the errors that would affect a memory page that has been proactively offlined, and compare this to the total number of errors for each category. While we observed a negligible overhead in application running times, the remaining question concerns how much memory overhead page migration incurred in each of the previous scenarios and cases.

1) Single-Node Scenario: Table VI shows the number of errors avoided for each error category and the amount of memory retired for a single-node run of LAMMPS with a memory footprint of 20 MB per process. The numbers are consistent with the coverage expected by the calculation in Section II and show that the majority of Chipkill errors can be avoided. The effectiveness depends on how errors cluster in space and time to trigger page migrations. The effectiveness in avoiding repeated Chipkill were 71% and 92% in cases 1 and 2, respectively. Case 3 does not activate Chipkill. With a small memory footprint, pages offlined represent 2.5% and 4.1% of memory used by the application, respectively. These numbers confirm that very few memory pages account for the vast majority of repeated errors.

2) Multi-Node Scenario: Table VII shows the statistics for the effectiveness of page migration in multiple parallel runs (6 in total), where error patterns are randomly selected. In this scenario, each node has its own statistics based on the error pattern used, and all nodes are affected by Chipkill, as described previously. The table shows the worst average (lowest effectiveness) and best average (highest effectiveness) for the 64 nodes in a given run, and the average for the 6 multiple runs. We show that on average, a significant 76% of all repeated Chipkills could be avoided by our mechanism. In the best case average, 80% effectiveness is achieved with only 0.6% of the application memory being offlined. Even the worst case average shows an effectiveness of 74%. We see that the conclusions drawn from single-node executions hold true for parallel executions.

3) Overall Resilience Improvement: As discussed in Section II, repeated Chipkill correction can be correlated with failure with a probability of 83%. We use this correlation to project how the overall resilience can be improved through our approach. For a parallel run of LAMMPS with real error patterns from BG/P logs, we showed that on average 76.42% of repeated Chipkill errors could be avoided. This implies the avoidance of 63.43% of memory failures, which shows that a significant number of memory failures can be avoided with negligible overhead to the application.

4) Comparison with Traditional Checkpoint/Restart: In [17], it is shown that application-level Checkpoint/Restart (CR) techniques are often more efficient than system-level autonomous CR. To evaluate our approach, we chose LAMMPS because it has a widely used application-level CR that we can quantitatively compare against our proactive error-avoidance method. We leverage the built-in, application-specific CR support in LAMMPS to measure CR overhead on BG/Q. We use the same input script for a run of approximately 5 hours for a total of 1000 simulation steps using 64 nodes. The CR support allows multiple writers, up to one per node. We set the most efficient scenario of 64 nodes writing checkpoint files concurrently. In a run without CR support, the application takes 294 minutes to complete. Enabling checkpointing at an interval of 199 steps for a total of 9 checkpoints, the application takes 308 minutes. We explicitly use an odd number of steps as checkpoint interval to avoid unnecessary CR overhead at the end of execution. The CR overhead is calculated to be (308-294)/9 $\approx$ 100 seconds per checkpoint, at an interval of 1960 seconds. We also measure the restart time by comparing the execution time of application runs that are restarted from the checkpoints to the no-CR run. The restart time is measured in the best case to be 117 seconds.

By assuming that the measured checkpoint interval is optimal for the application that runs forever, we can calculate the corresponding MTBF as described in [18], [19] and the associated performance overhead in terms of checkpoint overhead, loss of work overhead, and restart overhead. In [18], [19], the formula for optimal interval is expressed as $\approx \sqrt{2 \times mtbf \times ckp_{time}}$, which leads to MTBF = $(1960^2)/(2 * 100)$ $\approx$ 19000 seconds or 5.3 hours. The checkpoint overhead is 100/1960= 5.1%. The restart overhead is 117 seconds, or 117/19000= 0.6%, a small overhead. We compute the loss of work overhead using the excess life model described in [20]. We assume the failure distribution using the aforementioned error logs in which we normalized to the MTBF of 19000 seconds, and conservatively assume that a restart is triggered only when the number of Chipkill (CK) errors on a node reaches its maximum count of 65,535. The loss of work overhead is calculated to be 6.3%. Therefore, the total CR overhead is 5.1+0.6+6.3= 12.0% for a system with an MTBF of 5.3 hours. In comparison, our approach achieves a negligible overhead as shown in Table V. At the accelerated time scale of 1:1200 seconds, these cases are equivalent to a system with an MTBF of 8 minutes. The comparison shows that our approach can mitigate extremely high memory failure rate at a significantly lower performance overhead with respect to CR.

V. Related Work

Mechanisms for memory fault avoidance and handling have been deployed at various levels of the software stack, ranging from application and runtime to OS schedulers. Our work is related to a series of previous studies on memory failure prediction and proactive techniques to avoid failure. The predictability of memory failure is exploited in [6], [4], [3], [21]. In [6], [21] error logs from HPC systems are analyzed for common failure sources. They present evidence that memory and core failures are the main failure sources and explore correlation in their occurrences. While showing that these failures can be predicted, they do not explore mechanisms to take advantage of prediction. In [4], [3], memory error patterns in DRAM from different large systems are analyzed, showing that hard errors are the dominant errors in DRAM. A theoretical projection for errors avoided with OS support for bad-page retirement is presented in [3]. The study suggests better OS support for exploring these patterns but does not propose or implement one. The analysis in our work is closely related to the approach in these studies, but diverges in the way it is used. We depart from a typical offline approach to use error pattern recognition for real-time prediction and adaptation.

There are implementations of OS-based features to anticipate the impacts of memory failures. Mcelog [22] is a daemon for x86 Linux systems that monitors and interprets machine check messages. The daemon can be configured to trigger actions, including bad-page offlining. Bad-page offlining occurs when a certain number of correctable errors are experienced. Similar support implemented in Solaris 9 is presented and discussed in [23]. These studies also try to proactively avoid memory that shows early signs of failure. However, they rely solely on static error counting and do not explore real-time observation of spatial and temporal error distribution. Our work implements an algorithmic and dynamic approach to monitor errors, migrate data, and offline pages. In addition, these approaches have not been designed for and evaluated in an HPC environment.

Checkpoint/restart techniques have been commonly used to recover from memory failures. Failure prediction has been used to guide the determination of optimized checkpointing intervals in reactive fault management techniques [24], [25], [2]. While related in the aspect of trying to anticipate memory failures, we try to avoid failures before they happen. In this way, our approach differs fundamentally from CR by avoiding most of the overhead related to frequent checkpointing, work loss, and restart times. Unlike CR, which can resume application execution after failures in a broad range of faulty components such as processors and power supplies, our approach is limited because it currently targets only memory failures; therefore it cannot completely eliminate the need for CR. In some severe cases of memory errors, such as errors at the memory chip or link levels, our approach will not be effective; instead, these cases are detected by memory scrubbing tests and mitigated through CR on a spare node or a shared memory chip. However, our approach avoids memory failures and therefore reduces the frequent CR overhead. While ramdisk-based CR such as [25], [26] may have comparable speed and memory overhead with respect to our approach, their use of additional memory is also subjected to potential memory failures and therefore can also benefit from our proactive memory error avoidance approach.

There are also other proactive schemes exploiting health monitoring capabilities for failure prediction. They target different levels, such as the migration of processes or entire nodes. Fault-tolerant MPI masks memory failures in a parallel execution through replication or migration [27], [28]. Our approach is completely transparent to the MPI stack and does not require any changes to the application. The migration of entire nodes has been also shown to be feasible in HPC as shown in [29]. In our approach, by operating at the kernel level, we achieve intrinsically lower overheads compared to mechanisms at higher layers in the software stack.

VI. Conclusions and Future Work

We have shown how correctable error information can be used by the OS in real time to migrate data, offline faulty memory pages, and avoid repeated memory errors and failure. By implementing a prototype in Linux running on a BG/Q system, we have shown a working and transparent solution that is a viable alternative to traditional reactive mechanisms. Reproducing realistic error patterns in extensive logs from a BG/P system, we have shown that most of the repeated errors that affect a parallel run of a representative benchmark can be avoided with negligible performance overhead and a negligible amount of memory offlined. We have shown experimentally a coverage of 76% of repeated error avoidance, implying the avoidance of 63% of failures caused by memory. Using the built-in Checkpoint/Restart (CR) support in LAMMPS, we have shown that the performance overhead imposed by CR is significantly higher than that of our approach.

As future work, we plan to study the impact of repeated error avoidance on performance and power consumption. In this work we showed feasibility through an error emulation approach that does not impact the hardware cost of performing complex error correction. We plan to deploy and evaluate our mechanism in a production system, in a scenario where multiple real memory errors are expected and can be potentially avoided. Another approach we are considering is an accelerated fault-injection experiment to evaluate the effectiveness of our technique under real-world failures of memory. We envision new opportunities for memory error protection by showing that complex error correction in the field can be replaced with software support. We also envision extending the proactive reliability management paradigm to cover other components such as network links, processors, and power supplies. In a different direction, we plan to understand how to compensate for cases where our mechanism cannot avoid a failure. We plan to leverage studies showing tolerance for error corruption in application data, and use our migration mechanism to rearrange memory in such a way that suspect memory is used only to hold non-critical data. In the case where critical data cannot be protected, a proactive notification would allow applications to prepare for an imminent crash. We envision this support as an application-OS cooperation interface, allowing data criticality to be expressed by the application and notifications to be delivered to the application.

VII. Acknowledgments

We would like to thank Kyu-Hyoun Kim and Paul Coteus from the IBM T. J. Watson Research Center for sharing with us their expertise and details of the Blue Gene memory subsystem and error protection mechanisms and providing valuable feedback. We also thank the Computing, Communications, and Networking Division at LANL and the USENIX CFDR maintainers for collecting and publicly releasing the error logs used in this work.

References

[1] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, Sep. 2009.

[2] S. Chakravorty and L. V. Kale, "A fault tolerance protocol with fast fault recovery," in *International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–10.

[3] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 40, no. 1. New York, New York, USA: ACM, Mar. 2012, p. 111.

[4] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2012, pp. 1–11.

[5] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *International Conference on Dependable Systems and Networks*. IEEE, Jun. 2013, pp. 1–12.

[6] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 193–193–204–204, Jun. 2009.

[7] Z. Zheng, L. Yu, W. Tang, Z. Lan, R. Gupta, N. Desai, S. Coghlan, and D. Buettner, "Co-analysis of RAS log and job log on Blue Gene/P," in *International Parallel and Distributed Processing Symposium*. IEEE, May 2011, pp. 840–851.

[8] IBM Blue Gene team, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, no. 1/2, pp. 199–220, Jan. 2008.

[9] M. Gschwind, "Blue Gene/Q," in *International Conference on Supercomputing*. New York, New York, USA: ACM, Jun. 2012, p. 245.

[10] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, alan Gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar. 2012.

[11] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. W. Wisniewski, "FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment," in *International Symposium on Computer Architecture and High Performance Computing*. IEEE, Oct. 2012, pp. 211–218.

[12] "FusedOS." [Online]. Available: https://github.com/ibm-research/fusedos/

[13] "uthash." [Online]. Available: http://troydhanson.github.io/uthash/

[14] "ASC Sequoia benchmark codes." [Online]. Available: https://asc.llnl.gov/sequoia/benchmarks/

[15] "SLURM: A highly scalable resource manager." [Online]. Available: https://computing.llnl.gov/linux/slurm/

[16] "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE." [Online]. Available: http://mvapich.cse.ohio-state.edu/overview/mvapich2/

[17] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "McrEngine: A scalable checkpointing system using data-aware aggregation and compression," in *International Conference on High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE, 2012, pp. 17:1–17:11.

[18] J. T. Daly, "A model for predicting the optimum checkpoint interval for restart dumps," in *International Conference on Computational Science*. Springer–Verlag, 2003, pp. 3–12.

[19] ——, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, Feb. 2006.

[20] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Păun, and S. Scott, "An optimal checkpoint/restart model for a large scale high performance computing system," in *International Symposium on Parallel and Distributed Processing*. IEEE, Apr. 2008, pp. 1–9.

[21] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how HPC systems fail," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Jun. 2013, pp. 1–12.

[22] "mcelog: Advanced hardware error handling for x86 Linux." [Online]. Available: http://www.mcelog.org/

[23] P. Carruthers, Z. Totari, and M. Shapiro, "Assessment of the effect of memory page retirement on system RAS against hardware faults," in *International Conference on Dependable Systems and Networks*. IEEE, 2006, pp. 365–370.

[24] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into HPC systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2012.

[25] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2010, pp. 1–11.

[26] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. D. Panda, "A 1 PB/s file system to checkpoint three million MPI tasks," in *International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 143–154.

[27] R. Rajachandrasekar, X. Besseron, and D. K. Panda, "Monitoring and predicting hardware failures in HPC clusters with FTB-IPMI," in *International Parallel and Distributed Processing Symposium Workshops and PhD Forum*. IEEE, May 2012, pp. 1136–1143.

[28] G. E. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer–Verlag, Jan. 2000, pp. 346–353.

[29] F. Mueller, C. Engelmann, and S. Scott, "Proactive process-level live migration in HPC environments," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2008, pp. 1–12.