Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing

David Fiala, Frank Mueller
NCSU
Raleigh, NC
{*dfiala,fmuelle*}*@ncsu.edu*

Christian Engelmann
Oak Ridge Natl Lab
Oak Ridge, TN
engelmannc@ornl.gov

Kurt Ferreira, Ron Brightwell
Sandia Natl Labs
Albuquerque, NM
kbferre@sandia.gov

Rolf Riesen
IBM
Dublin, Ireland
rolf.riesen@ie.ibm.com

***Abstract*—Faults have become the norm rather than the exception for high-end computing on clusters with 10s/100s of thousands of cores. Exacerbating this situation, some of these faults remain undetected, manifesting themselves as silent errors that corrupt memory while applications continue to operate and report incorrect results.**

This paper studies the potential for redundancy to both detect and correct soft errors in MPI message-passing applications. Our study investigates the challenges inherent to detecting soft errors within MPI application while providing transparent MPI redundancy. By assuming a model wherein corruption in application data manifests itself by producing differing MPI message data between replicas, we study the best suited protocols for detecting and correcting MPI data that is the result of corruption.

To experimentally validate our proposed detection and correction protocols, we introduce RedMPI, an MPI library which resides in the MPI profiling layer. RedMPI is capable of both online detection and correction of soft errors that occur in MPI applications without requiring any modifications to the application source by utilizing either double or triple redundancy.

Our results indicate that our most efficient consistency protocol can successfully protect applications experiencing even high rates of silent data corruption with runtime overheads between 0% and 30% as compared to unprotected applications without redundancy.

Using our fault injector within RedMPI, we observe that even a single soft error can have profound effects on running applications, causing a cascading pattern of corruption in most cases causes that spreads to all other processes. RedMPI's protection has been shown to successfully mitigate the effects of soft errors while allowing applications to complete with correct results even in the face of errors.

I. Introduction

In High-End Computing (HEC), faults have become the norm rather than the exception for parallel computation on clusters with 10s/100s of thousands of cores. Past reports attribute the causes to hardware (I/O, memory, processor, power supply, switch failure etc.) as well as software (operating system, runtime, unscheduled maintenance interruption). In fact, recent work indicates that (i) servers tend to crash twice a year (2-4% failure rate) [1], (ii) 1-5% of disk drives die per year [2], (iii) DRAM errors occur in 2% of all DIMMs per year [1], which is more frequent than commonly believed, and (iv) large scale studies indicate that simple ECC mechanisms alone are not capable of correcting a significant number of DRAM errors [3].

Table I
Reliability of HPC Clusters

System	# CPUs	MTBF/I
ASCI Q	8,192	6.5 hrs
ASCI White	8,192	5/40 hrs ('01/'03)
PSC Lemieux	3,016	9.7 hrs
Google	15,000	20 reboots/day
ASC BG/L	212,992	6.9 hrs (LLNL est.)

Even for small systems, such causes result in fairly low mean-time-between-failures/interrupts (MTBF/I) as depicted in Figure I [4], and the 6.9 hours estimated by Livermore National Lab for its BlueGene confirms this. In response, long-running applications on HEC installations are required to support the checkpoint/restart (C/R) paradigm to react to faults. This is particularly critical for large-scale jobs; as the core count increases, so does the overhead for C/R, and it does so at an exponential rate. This does not come as a surprise as any single component failure suffices to interrupt a job. As we add system components (cores, memory, disks), the probability of failure combinatorially explodes.

For example, a study from 2005 by Los Alamos National Laboratory estimates the MTBF, extrapolating from current system performance [5], to be 1.25 hours on a petaflop machine. The wall-clock time of a 100-hour job in such as system was estimated to increase to 251 hours due to the C/R overhead implying that 60% of cycles are spent on C/R alone, as reported in the same study. More recent investigations [6], [7] revealed that checkpoint/restart efficiency, *i.e.*, the ratio of useful vs. scheduled machine time, can be as high as 85% and as low as 55% on current-generation HEC systems. Recent work by Sandia [8] shows rapidly decaying useful work for increasing node counts (see Table II). Only 35% of the work is due to computation for a 168 hour job on 100k nodes with a MTBF of 5 years while the remainder is spent on checkpointing, restarting and then partial recomputation of the work lost since the last checkpoint. Figure III shows that for longer-running jobs or shorter MTBF (closer to the ones reported above), useful work becomes *insignificant* as most of the time is spent on

Table II
168-HOUR JOB, 5 YEAR MTBF

# Nodes	work	checkpt	recomp.	restart
100	96%	1%	3%	0%
1,000	92%	7%	1%	0%
10,000	75%	15%	6%	4%
100,000	35%	20%	10%	35%

restarts.

The most important finding of the Sandia study is that **redundancy in computing can significantly revert this picture.** By doubling up the compute nodes so that every node N has a replica node N', a failure of primary node N no longer stalls progress as the replica node N' can take over its responsibilities. Their prototype, rMPI, provides dual redundancy [8]. And *redundancy scales*: As more nodes are added to the system, the probability for simultaneous failure of a primary N *and* its replica rapidly decreases. Of the above overheads, the recompute and restart overheads can be nearly eliminated (to about 1%) with only the checkpointing overhead remaining — at the cost of having to deploy twice the number of nodes (200,000 nodes in Figure III) and four times the number of messages [8]. But once restart and rework overheads exceed 50%, redundancy is actually *cheaper* than traditional C/R at large core counts.

Table III
100K NODE JOB, VARIED MTBF

job work	MTBF	work	checkpt	recomp.	restart
168 hrs.	5 yrs	35%	20%	10%	35%
700 hrs.	5 yrs	38%	18%	9%	43%
5,000 hrs,	1 yr	5%	5%	5%	85%

The failure scenarios above only cover a subset of actual faults, namely those due to fail-stop behavior or at least detectable by monitoring of hardware and software. Silent data corruption (SDC) is yet a different class of faults. It materializes as bit flips in storage (both volatile memory and non-volatile disk) or even within processing cores. A single bit flip in memory can be detected (with CRC) and even mitigated with error correction control (ECC). Double bit flips, however, forces an instant reboot after detection since ECC cannot correct such faults. While double bit flips were deemed unlikely, the density of DIMMs at Oak Ridge National Lab's Cray XT5 causes them to occur on a daily basis (at a rate of 1-2 per day for 75,000+ DIMMs) [9].

Meanwhile, even single bit flips in the processor core remain undetected as only caches feature ECC while register files or even ALUs typically do not. Significant SDC rates were also reported for BG/L's unprotected L1 cache [10], which explains why BG/P provides ECC in L1. Nvidia is making a similar experience with its shift to ECC in their Fermi GPUs. Yet, hardware redundancy, such as Tandem/HP's NonStop architecture remains extremely costly [11], [12], [13], [14], [15].

Today, the frequency of bit flips is no longer believed to be dominated by single-event upsets due to radiation from space [16] but is increasingly attributed to fabrication miniaturization and aging of silicon given the increasing likelihood of repeated failures in DRAM after a first failure has been observed [1]. With SDCs occurring at significant rates, not every bit flip results in faults. Flips in stale data or code remain without impact, but those in active data/code may have profound effects and potentially render computational results invalid without ever being detected. This creates a severe problem for today's science that relies increasingly on large-scale simulations. Redundant computing can detect SDCs where relevant, i.e., when results are impacted. While detection requires dual redundancy, correction is only feasible with triple redundancy. Such high levels of redundancy appear costly, yet may be preferable to flawed scientific results. Triple redundancy is also cheaper than comparing the results of two dual redundant jobs, which would be the alternative at scale given the amount of useful work without redundancy for large systems from Table III. Overall, the state of HEC requires urgent investigation to level the path to exascale computing — or exascale HEC may be doomed as a failure (with very short mean times, ironically).

A. Modeling Redundancy

Elliott *et al.* [17] combine partial redundancy with checkpointing in an analytic model and experimentally validate it. Results indicates that for an exascale-size machine, more jobs can utilize a cluster under redundancy than would be possible with checkpointing without redundancy for the same number of cores, i.e., redundancy increases capacity computing in terms of HPC job throughput.

We used this model in combination with Jaguar's system MTBF of 52 hours [18] (equivalent to a node MTBF of 50 years) to assess the viability of redundancy. Consider a 128 hour job (without checkpointing). We then assess the time required for such a job without redundancy (1x), dual redundancy (2x) and triple redundancy (3x) at different node counts under weak scaling and with an optimal checkpoint interval to minimize overall execution (see Fig. 1). At 18,688 nodes (Jaguar), marked as line C, single-node (1x) runs are about 7% faster than dual (2x) and 20% faster than triple (3x) redundancy. The problem is that a job at 1x will have no indication if it had been subjected to an SDC. Consider Jaguar's double bit error rate of once a day again [19], which is silently ignored (to increase system availability) as it cannot be corrected. Scientists will not know if their outputs were affected, i.e., if outputs are flawed (incorrect science problem).

Let us consider dual redundancy at half the node count

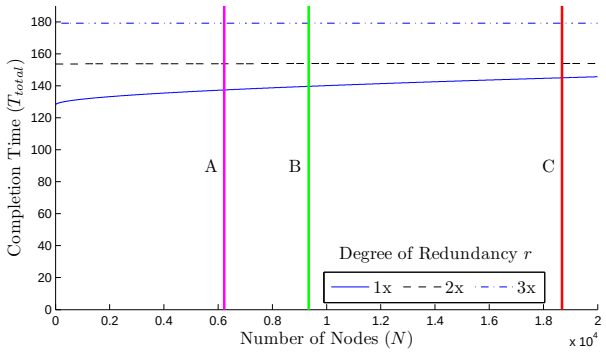

Figure 1. Modeled Time to Completion with Redundancy

of Jaguar (line B in Fig. 1). In order to ensure absence of SDCs, a user would have to run a single redundant (1x) job twice for a total time of about 280 hours (twice 140) vs. a dual redundant (2x) job at twice the number of nodes (full Jaguar size, line A) with 155 hours. Hence, dual redundancy, results in nearly half the wall-clock time if SDC detection is a requirement for verification and validation of a job's results.

Consider triple redundancy at a third of Jaguar's node count (line A). Running two jobs at 1x takes about 276 hours (twice 138), a dual redundant (2x) job 145 hours and a triple redundant job 180 hours. The output of the two 1x jobs differs, a third run would be required (assuming that two of them produce correct results). If the dual redundant (2x) job detects an error, it also needs to be rerun. The triple redundant (3x) job, in contrast, can correct errors so that no reruns would be needed.

At exascale core counts of one million and a node count of 100,000 (swim lane 1 [20], [21]), dual redundancy would have the lowest cost (lower than single job at 1x). The additional cost of SDC correction at triple redundancy adds another 14% overhead in wall-clock time, with the benefit of no repeated runs for SDCs. This is based on the assumption of Jaguar's system MTBF (52 hours), even though the MTBF could be much smaller given that double bit errors for a 128 Petabyte system would occur every four minutes (compared to one a day today on Jaguar) [19]. SDC detection, if not correction, may thus become essential at exascale.

B. Contributions

The main contributions of this work are (i) the design of a novel silent data correction detection / correction methods and (ii) a study on the challenges and costs of performing SDC protection using redundancy. By utilizing redundancy, our key to success is to not only rely on reactive resilience requiring restart overheads but to sustain failures with forward computational progress without a need to restart. In this context, our work is addressing the following research questions:

1) What are the protocols best suited to realize SDC detection and correction at the communication layer?
2) What is the cost of different levels of redundancy with respect to application runtime overhead?
3) How does quickly and critically does data corruption spread across communicating MPI processes?
4) Is redundancy effective at identifying and isolating corrupted data while still producing correct results?

Answering these questions, our work makes the following major contributions: (1) We contribute the design and implementation of protocols for SDC detection and correction at the communication layer. (2) We demonstrate the capabilities and assess the cost of redundancy to (a) detect SDC and (b) recover from such corruption in experiments on a real system. As SDCs are being observed for 10k+ cores and also due to smaller fabrication sizes, C/R techniques fail to uncover SDCs, which can render the output of scientific computations incorrect without knowledge of application scientists. While dual redundancy can detect SDCs, triple redundancy can actually correct them through voting. We study the benefits and limitations of the spectrum ranging from no redundancy over dual to triple redundancy in terms of overhead and computing/interconnect resource costs. A key challenge is to limit the overhead for SDC detection by reducing the relevant footprint of computational results, which we explore. (3) We assess the resilience of HEC jobs to faults through injection. Hardware and software failures can be studied through injection, which is in a native environment on an actual cluster. (4) We develop a live SDC tracking and reporting framework to investigate the effects of SDCs on applications in terms of their rate of taint (corruption) progression spreading from node to node via MPI communication. Further, we use this framework to evaluate several application responses to fault injection and classify three types of observed behavior that result in invalid data being generated. In summary, this work contributes to fault detection and recovery in significantly advancing existing techniques by controlling levels of redundancy intervals in the presence of hardware and software faults.

II. Design

This work presents RedMPI, an MPI library that is capable of both detecting and correcting SDC faults. RedMPI creates "replica" MPI tasks for each "primary" task and performs online MPI message verification intrinsic to existing MPI communication. The replicas compare received messages, or hashes, from multiple senders and can thus detect if a process's communication data has been corrupted.

RedMPI can run in double redundant mode and detect divergent messages between replicas. Such messages are indicative of corruption due to the fact that replicas will be run in a deterministic manner. When RedMPI is run in triple redundant mode, it gains the additional potential to also correct faulty messages from a corrupted replica. RedMPI

supports additional levels of redundancy for environments where multiple near-simultaneous faults can occur during data transmission. A voting algorithm is used to determine which of the received messages are correct and should be used by all receivers.

To detect SDCs, RedMPI solely analyzes the content of MPI messages to determine divergence between replicas during communication. Upon divergence, the result deemed to be invalid is discarded on the receiver side and transparently replaced with a known "good" value from another replica.

A different SDC detection approach would be to constantly compare the memory space of replicas' processes and compare results. Such an approach suffers from excessive overhead due to constant traversals of large memory chunks, overhead due to global synchronization to ensure that each process is paused at the exact same spot during a memory scan, and the communication required for replicas to compare their copy of each memory scan while looking for differences. In this case, if corruption is detected, it is not feasible to correct the memory while the application is running as this could interfere with application-side writes to the same memory region. This, in turn, could necessitate a rollback of all tasks to the last "good" checkpoint (assuming that checkpointing was also enabled).

By instead focusing on the MPI messages themselves, we have cut our search area down to only data that is most critical for correctness of an MPI application; i.e., we argue communication correctness is a necessary (but not sufficient) condition for output correctness. Moreover, should an SDC occur in memory that is not immediately communicated over MPI, the fault is eventually detected as the corrupted memory may later be accessed, operated on, and finally transmitted. The same principle holds true for data that became corrupted while residing in a buffer or any other place in memory. If the SDC is determined to eventually alter messages, then RedMPI detects it during transmission, independent of when or how the SDC originated.

It is very important to note that RedMPI is designed to protect an an entire application from SDC by using replication. RedMPI is not designed to protect an interconnect. By assuming that an application's most critical data is communicated during/after computation, we have effectively reduced the scope to data that gets communicated and may be compared between replicas to ensure consistency. A process receiving corrupted data that affects important calculations will eventually result in message correction so that uncorrupted replicas are guaranteed to have received correct data, only.

A. Point-to-Point Message Verification

The core of the RedMPI's error detection capabilities are designed around a reliable, verifiable point-to-point communication protocol. Specifically, a point-to-point message (e.g., `MPI_Isend`) sent from an MPI process must be identical to the message sent by other replicas for any given rank. Upon successful receipt of a message, the MPI application is assured that the message is valid (not corrupted).

Internally, a verification message may take the form of a complete message duplicate that is compared byte by byte. Alternatively, since MPI messages may be large, it is in many cases more efficient to create a unique hash of the message data and use the hash itself for message verification to reduce network bandwidth. Message data verification can be performed at either the sender or the receiver.

Let us first consider the case of sender-side verification. To perform verification at the sender, all of the replicas need to send a message to communicate with each other and verify their content (through some means) before sending the verified data to the receiving replicas. However, this approach incurs added latency and overhead for each message sent due to the time taken to transmit between replicas and to perform internal verification messages. Additionally, it is best to optimize for the critical path; i.e., assuming that a sent message tends to not be corrupted and that all senders have matching data. A sender-side approach is subject to additional overhead for every message sent at both sending and receiving nodes. Specifically, while every sent message is treated as suspect, the time required for the senders to agree that each of their own buffered messages is correct represents the time lost on the receiver side before the application can proceed. For this reason, RedMPI's protocols use receiver-side verification resulting in faster message delivery with considerably reduced message latency.

B. Assumptions

RedMPI does not protect messages over a transport layer, such as TCP or InfiniBand, and assumes the transport to be reliable. An unreliable network could cause undefined behavior and deadlock RedMPI. Fortunately, protection in the transport layer is well understood and is a common feature of modern communication fabrics, e.g. hamming codes or checksums for correction/retransmission.

As the primary focus of this work is to investigate redundancy as a means to protect application data, RedMPI only attempts to protect against corruption in data and not application code or instructions. RedMPI does not protect MPI I/O functionality, but orthogonal work [22] could be combined with RedMPI to also cover this aspect.

Deterministic ordering of messages between replicas (for wildcard receives, MPI_Waitall/any calls or reduction orders) is ensured via back-channel communication (see Section III).

In this paper, we frequently reference RedMPI's `MPI_Isend` and `MPI_Irecv`. Note that RedMPI does, in fact, support 53 standard MPI functions including collectives via interposing, such as `MPI_Send` and `MPI_Recv`, around

their non-blocking equivalents within RedMPI. For brevity, details of these and other supported functions are omitted.

III. Implementation

RedMPI provides the capability of soft error detection for MPI applications by online comparison of results of nearly identical replica MPI processes. To an MPI developer, the execution of replica processes of their original code is transparent as it is handled through MPI introspection within the RedMPI library. This introspection is realized through the *MPI profiling layer* that intercepts MPI function calls and directs them to RedMPI. The profiling layer provides a standard API allowing libraries to wrap MPI calls and add additional or replacement logic for API calls.

To understand how RedMPI functions internally, it is first important to understand how redundancy is achieved within RedMPI. When launching an MPI job with RedMPI, some *multiple* of the original number of desired processes needs to be launched. For example, to launch an MPI job that normally requires 128 processes instead requires 256 or 384 processes for dual or triple redundancy, respectively. RedMPI handles redundancy internally and provides an environment to the application that appears to only have the originally required 128 processes.

The primary difference between replica MPI processes is a replica rank that distinguishes redundant processes. For example, for an application to run with three replicas (triple redundancy), it would be started with three times as many MPI ranks as usual. Internally, the number of ranks visible to the MPI application would be divided by three where each redundant rank carries an internal replica rank of 0, 1, or 2. Figure 5 shows how triple redundancy may appear within an MPI application expecting a size of three.

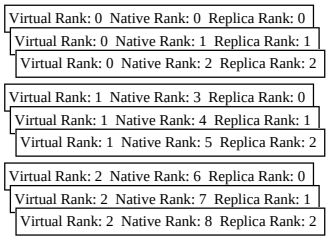

Figure 5. Triple Redundancy

The actual rank assigned to a process by `mpirun/mpiexec` is referred to as the native rank. The rank that is visible to an MPI process via the `MPI_Comm_rank` API is referred to as the virtual rank. Likewise, the size returned by `MPI_Comm_size` is referred to as the virtual size. The number of replicas running per virtual rank describes the redundancy of the application and is referred to as the replication degree. Within RedMPI, a mapping structure is stored in each process that allows the forward and reverse lookup of any processes' native rank, virtual rank, or replica rank.

A. Rank Mapping

When launching an MPI job, the mapping of native ranks may be specified on the command line with either a custom map file or by specifying a flag to indicate the desired virtual size. When a virtual size is specified on the command line, RedMPI automatically generates a structure that maps native ranks to a virtual rank of $[0 \ldots virtual\_size-1]$ and assigns replica ranks of $[0 \ldots (native\_size/virtual\_size)-1]$. Additionally, for each communicator, group, or MPI topology created within the MPI application, another map is created to track ranks within the new group.

Internally, mappings can be translated using a formula or by storing the data in a lookup structure. The formula method provides a simple, deterministic mapping function with low memory requirements, but it is not capable of providing fine-tuned control of rank mapping. By using a custom rank map file passed to RedMPI during startup, the user has the capability to specifically designate which virtual ranks are mapped to a native rank. This is advantageous in particular when the user desires to put replica processes on the same physical host or on neighboring hosts with low network latency. If a custom map file is omitted, the mapping formula is used to build the initial structure upon startup.

B. Message Corruption Detection & Correction

1) Method 1: All-to-all: RedMPI's first receiver-side protocol, All-to-all, supports both message verification and message voting to ensure that the receiver discards corrupted messages. The All-to-all method requires that each MPI message sent is transmitted from all sender replicas to each and every receiver replica. Thus, for a redundancy degree of three, each sender sends three messages where one message goes to each replica receiver as demonstrated by Figure 2. This means that for a degree of 2 or 3 the number of messages sent for a single `MPI_Isend` is 4 or 9, respectively. Each receiver listens for a message from each sender replica and places such messages in separate receive buffers.

Such message verification requires each sender to send $degree$ messages for each MPI send encountered. This is realized by interposing `MPI_Isend` via RedMPI using the MPI profiling layer. The new `MPI_Isend` routine determines all replicas for the virtual rank of a message's destination. For each such replica, RedMPI performs a non-blocking send with a payload of the entire message and records the `MPI_Request` for each pending send. Upon completion, the overridden `MPI_Isend` returns back to the MPI application a single `MPI_Request` that can later be used by `MPI_Test` or `MPI_Wait`. In a similar manner, `MPI_Irecv` is interposed by RedMPI to look up all replicas

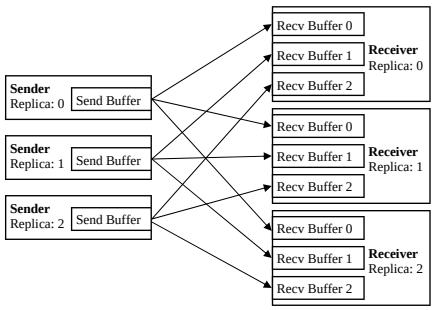

Figure 2. All-to-all Method Overview

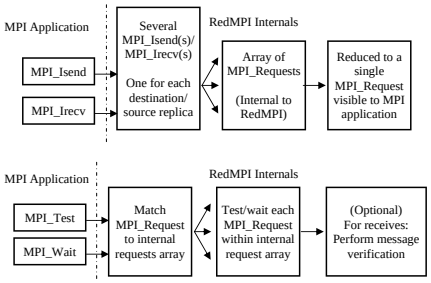

Figure 3. All-to-all Function Overrides

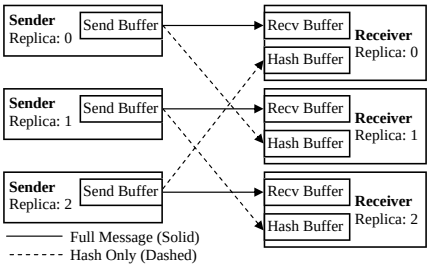

Figure 4. MsgPlusHash Method Overview

of the source's virtual rank and internally posts a non-blocking receive for a message from each replica. Every receive is stored into a different, temporary buffer entry. Again, all `MPI_Request` handles originating from non-blocking receives are recorded internally, but only a single `MPI_Request` is returned to the MPI application. Figure 3 visualizes this process.

Following an `MPI_Isend` or `MPI_Irecv`, an MPI application usually completes these requests with an `MPI_Test` or `MPI_Wait`. RedMPI interposes these functions as it needs to test not just the single `MPI_Request`, but rather impose a test for each array element of internal MPI requests corresponding to sends/receives from all replicas. The `MPI_Request` is looked up and the test or wait is performed on all outstanding requests. If the test or wait was performed on a request from an `MPI_Isend`, then no further action from RedMPI is required once the requests complete. Only a request from an `MPI_Irecv` requires extra steps to verify message reception from each replica.

When an MPI application receives a message, RedMPI internally waits for all replica MPI receive requests to finish during an `MPI_Test` or `MPI_Wait` before verifying the data. The actual verification occurs before `MPI_Test` or `MPI_Wait` return to the MPI application, but after all replica receives arrive. Verification is performed via memory comparison or computing a `SHA1` hash of each replica receive buffer and then comparing these hashes.

If during message verification a buffer mismatch is detected, RedMPI mitigates in a manner dependent on the degree of replication. With replication degree of two, it is impossible to determine which of the two buffers is corrupt. Hence, an error is logged noting corruption detection, but no corrective action may proceed since the source of corruption is indeterminate. With a replication degree exceeding two, buffers are compared and corrupted messages are voted out upon mismatch with the simple majority (of matching messages). In this event, RedMPI ensures that the MPI application's receive buffer contains the correct data by copying one of the verified buffers if necessary.

2) Method 2: Message Plus Hash (MsgPlusHash): The MsgPlusHash (message plus hash) corruption detection and correction method provides a key performance enhancement over the All-to-all method by vastly reducing the total data transfer overhead per message and the number of messages in the general case. Similar to the All-to-all method, MsgPlusHash performs message verification solely on the receiver end. The critical difference is that MsgPlusHash sends one copy of a message originating from an `MPI_Isend` in addition to a very small hash message. This change in protocol allows each sending replica to transmit their message only once, while the additional hash message is later used to verify each receiver's message. MsgPlusHash's contribution is a reduction of messages and thus bandwidth required from n^r to simply $n * r$ where n is the number of messages sent and r is the degree of replication.

Internally, the MsgPlusHash method interposes `MPI_Isend`, `MPI_Irecv`, `MPI_Test`, and `MPI_Wait` similarly to the All-to-all method previously discussed. The following logical overview of MsgPlusHash outlines how the MsgPlusHash implementation differs, while the same level of transparency is provided to MPI applications as for All-to-all. E.g., MsgPlusHash internally utilizes multiple send/receive `MPI_Request` handles, but the MPI application only ever receives one such `MPI_Request` handle.

To detect message corruption, the minimum requirement is a comparison between two different sources. Additionally, the most likely scenario (critical path) is for corruption to not exist. The MsgPlusHash method takes full advantage of these facts by only receiving a single copy of any message transmitted plus a hash from an alternate replica. From an efficiency standpoint, it is not necessary to send two full messages since a hash suffices to verify data correctness without imposing overheads of full message retransmission. Once the full message is received, a hash of the message is generated at the receiver and compared with a hash from a different replica. In the likely event that hashes match, the receiver can be assured that its message is correct, i.e., no corrective action is taken.

As shown in Figure 4, each sender replica must calculate where to send its message and where to send a hash of its message. The actual message's destination is simply

calculated by finding the receiver with the same replica rank as the sender. The hash message's destination is calculated by taking the sender's replica rank and adding one. In the event that the destination replica rank exceeds the replication degree, the destination wraps around to replica rank 0. This pattern provides a simple and elegant solution to ensure each receiver always gets a copy of the full message plus a hash of the message from a different sender replica over a ring of replicas.

In the event that the message's hash does not match the received hash, it is necessary to determine if either the message is corrupt or if the received hash was produced from a corrupt sender. In any case, if a sender becomes corrupt, it transmits both the corrupted message and a hash of the corrupted message to adjacent receiving replicas. It is important to realize that a single corrupt sender affects both receivers. For example, with a replication degree of three where the middle sender (replica 1) transmits a corrupted message, we can see from Figure 4 that both receiver replicas 1 and 2 are affected. In this particular case, receiver replica 1 has received a corrupt message, but a good hash since sender replica 0 was not corrupted. Conversely, receiver replica 2 has received a valid message, but a hash of a corrupted message. In this scenario, both receiver replicas 1 and 2 cannot yet determine if their message is corrupt, but they are both aware that one of their senders was in fact corrupted. Additionally, receiver replica 0 is unaware of any corruption since both message and hash matched on arrival. If the replication degree had only been two, a corrupt error would be logged at this point, but no corrective action would be available. With larger replication degrees, in contrast, a corrupted message can be corrected.

MsgPlusHash message correction is a multi-step process that takes place on the receiver replicas that have been flagged with potential corruption. In this event, there are always two adjacent receiver replicas that are aware of corruption since both are affected by the same corrupt sender replica. Yet, these receivers cannot easily identify whether their message or their hash was corrupted. By analyzing the communication pattern, it is obvious that the replica with a higher replica rank always contains the corrupt message with a bad hash. Therefore, the two adjacent replicas communicate with one another to determine which of them holds a correct message. After this handshake, the higher replica rank transmits a correction message to the lower ranked replica to complete the correction. (Ranks wrap around 0 and the highest rank in these situations.)

Corrupted Adjacent Replica Discovery: After a process encounters a message and hash mismatch, it initiates a handshake protocol to discover which of its adjacent replicas are also participating in a discovery. For each adjacent rank also actively trying to discover a potentially corrupted process, the other rank engages in the discovery protocol since its message and hash did not match. Any rank that did not obtain a mismatched message and hash is entirely unaware of the corruption elsewhere and thus not participate in the this protocol. In order for the two searching processes to find each other, they both attempt to send a probe to the rank below them (replica rank - 1) while simultaneously issuing a receive probe from the rank above them (replica rank + 1). After one of the processes receives a probe, an acknowledgment is returned. Figure 6 depicts this process. The highest rank's X+1 wraps around to rank 0 and vice versa.

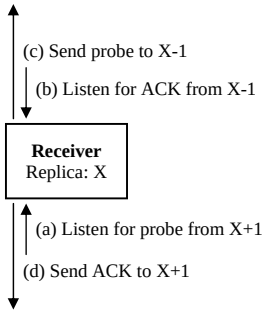

Figure 6. MsgPlusHash Correction

In part (a) of Figure 6, the process posts a non-blocking receive to listen for a probe from above. Next, in part (b), the process posts a non-blocking receive to listen for an acknowledgment from the process below. With the receives in place, part (c) posts a non-blocking send as a probe to the rank below. The probe contains a copy of the received message's hash as a means to match this particular probe on the other end. At this point, the process waits for either the probe or acknowledgment requests to complete as they both result in different outcomes. If a probe message is received then the process can immediately assume that it is the lower-ranked replica and, as such, has a copy of the corrupted message due to the communication patterns. This lower rank then sends an acknowledgment (see part (d)) to signal that the discovery is complete. Meanwhile, if a process receives an acknowledgment instead of a probe message then that rank immediately assumes that it is the higher ranked process with a valid copy of the original message. In both cases, once discovery has completed, any outstanding sends and receives that were posted but left incomplete are now canceled through MPI from RedMPI.

The nature of the discovery process creates a problem: Two unique SDCs detected by adjacent replicas at separate times may send probe messages that are received in a later discovery. RedMPI handles this case by using hashes to identify whether a probe pertains to the SDC at hand. Probes that are unrelated to the current discovery process is safely discarded until an expected probe hash arrives.

With discovery complete, the higher ranked replica sends a full copy of the original, validated message to the lower

rank. The lower rank receives a copy of this message within the application's buffer while overwriting the copy that originally was received in a corrupted state. Once this transfer completes, all replicas hold a validated message copy in their buffers and the MPI application may proceed.

Notice that RedMPI may sustain an unlimited number of corrupt messages from a sender provided that the degree of redundancy is greater than two and that no other replica of the same virtual rank becomes corrupt simultaneously. By our design, corruptions remain isolated to a single process, even if never corrected.

C. MPI Operations

Deterministic Results: RedMPI relies on keeping replica processes running with approximately equal progress in execution. As replicas execute in a deterministic manner, we guarantee that all MPI messages are sent in exactly the same frequency, order, and message content. There are, however, a few factors that might derail the replicas leading to non-deterministic results that would leave RedMPI inoperable, which has to be precluded. In particular, care was taken to ensure any MPI routine with the potential to diverge in execution progress of replicas is instead replaced with logic that provides the same results across all replicas.

One notable MPI routine with the potential to induce divergence is `MPI_Wtime` function. Not only is `MPI_Wtime` extremely likely to return a different value between separate processes and separate hosts, but its usage may guarantee different outcomes across processes especially if used as a random number seed. The divergence problem is solved by allowing only the rank zero replica to call `MPI_Wtime`. Since all replica ranks call `MPI_Wtime` at about the same time, the first replica simply sends a copy of its result to the others, which is then returned to the MPI application.

Another MPI routine with similar potential is the `MPI_Iprobe` function. Unlike `MPI_Wtime`, a probe may result in inconsistent results among replicas due to networking delays. It is possible that all but one replica received a message. To prevent results of `MPI_Iprobe` from diverging, the lowest ranking replica performs a real `MPI_Iprobe` for the requested message. Following the non-blocking probe, the lowest rank then sends a copy of the results to all higher ranking replicas. If `MPI_Iprobe` returned no message, then every replica simply reports that no message was found. Otherwise, if the lowest rank did report probing a message then each higher rank enters a blocking `MPI_Probe` to wait until their copy of the message arrives. As every replica has the same communication pattern, they are guaranteed to return from `MPI_Probe` quickly if the probed message had not, in fact, already arrived. Many other MPI functions such as `MPI_Testany`, `MPI_Testall`, `MPI_Waitany`, and `MPI_Waitall` are handled similar to `MPI_Iprobe` as previously described.

Random number generation may be considered a source of divergence, but in our experience many applications seed their random number generator with `MPI_Wtime`, which has already been protected. If this is not the case for an application then other protective measures would be required to ensure consistency between replicas or the application's source may need to be slightly modified to conform.

Some MPI operations (*e.g.*, `MPI_Recv`, `MPI_Iprobe`) may specify wildcards for their source or tag parameters (*i.e.*, `MPI_ANY_TAG`, `MPI_ANY_SOURCE`). To ensure that replicas do not receive messages from differing nodes in a receive and to ensure that probe results return the same source and tag, RedMPI handles these wildcards differently than regular operations when they are encountered during program execution. When a wildcard is detected, only the lowest ranking replica actually posts the wildcard while the other replicas await an envelope containing the source and tag from the lowest replica in a manner similar to how `MPI_Wtime` is handled. Additional care must be taken to ensure that only the lowest rank replica can post receives until the wildcard has been resolved. Full technical details are omitted due to space. Overall, RedMPI fully supports wildcard sources, wildcard tags, or both in combination.

Mathematical operations such as those in `MPI_Reduce` will execute in the same order across replicas independent of their virtual rank. For this reason, if a floating point reduction is performed in a tree then although differing virtual ranks might perform different floating operations, our virtual replicas will always perform the same operations in order.

Collectives: MPI collectives pose a unique challenge for corruption detection/correction. First, there is a lack of non-blocking collectives in the MPI-2 standard (to be addressed in the future MPI-3 standard). Without non-blocking collectives, it is impossible to overlap collective operations. Thus, it is possible to sustain a faulty process in a collective that does not participate or encounter other unforeseen problems. These issues may cause other participants to become non-responsive ("hang") or fail ("crash"). A second critical issue with native MPI collectives is the inability to detect/correct messages at the granularity of individual processes.

RedMPI supports two solutions for collectives:

1) Built-in linear collectives that map all collectives to point-to-point messages inside of RedMPI: Our linear collectives are portable to any MPI implementation and reside entirely in the profiling layer. These collectives are not necessarily performance oriented for large scale usage as we tried to avoid direct replication of existing MPI functionality.
2) An Open MPI collectives module that acts as a wrapper to other existing collective modules, such as `linear` or the more efficient `tuned`: Our customized module resides within Open MPI and translates all point-to-point operations that would normally

occur without any RedMPI instrumentation into operations that call the RedMPI library. By redirecting collective communication through RedMPI we are able to exploit the enhanced performance of the native collectives while still providing SDC protection. Our solution utilizes the Open MPI wrapper module during experiments. The same algorithmic solution can also be applied to other MPI implementations in place of the portable but less efficient RedMPI linear collectives.

The normal MPI communication stack is shown in Figure 7. However, when RedMPI is enabled along with its counterpart collectives wrapper module, both the PMPI layer and collectives have additional instrumentation as shown in Figure 8. All MPI requests from the application are now directed through the PMPI RedMPI library residing between the application and Open MPI. Application point-to-point communication is interposed in to its redundant/deterministic counterparts and sent to Open MPI's point-to-point communication layer. Application collectives are handled quite differently; a collective is routed through RedMPI and then to Open MPI's collective modules. Normally each Open MPI collective module would directly communication with the point-to-point communication layer, but this would circumvent RedMPI's point-to-point communication protocol for SDC protection. To solve this challenge, the RedMPI collective module intercepts point-to-point communication that is generated within Open MPI's collectives and redirects it to the RedMPI layer where it can be protected similar to normal application communication. This allows RedMPI to efficiently take advantage of Open MPI's more featureful and topologically-aware collectives functionality.

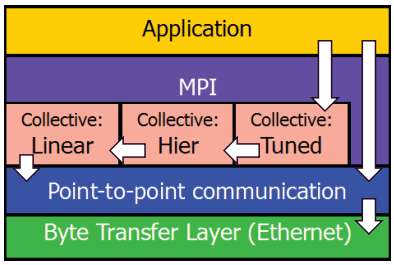

Figure 7. MPI stack without RedMPI's collectives module

IV. Fault Injector Design

As the research goals of this work include detecting and protecting applications from silent data corruption, an integrated fault injection tool is required to evaluate the effectiveness of RedMPI to detect and correct memory errors during execution. Additionally, the same fault injector can later be used to monitor the adverse effects of SDC on

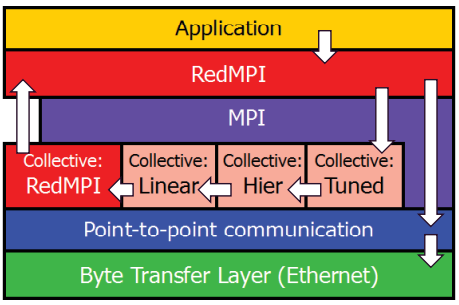

Figure 8. MPI stack with RedMPI's collectives module and PMPI layer

running applications when RedMPI is not actively protecting them.

To experimentally determine the effect of corruption and verify corrective actions, our fault injector was designed to produce data corruption in a manner resembling naturally occurring faults. Namely, single bit flips undetected by ECC are of interest (e.g., within an arithmetic-logic unit of a processor) when their effects eventually propagate into a message transmission over MPI. Alternatively, also of interest is SDC due to multiple bit flips in main memory resulting in a corrupted bit pattern that ECC is unable to detect / correct.

Our fault injector, which is built to co-exist with RedMPI, specifically targets MPI message send buffers to ensure that each injection actually impacts the MPI application while simultaneously reaching message recipients. When activated, the fault injector is given a frequency of $1/x$ during launch, which is the probability that any single message may become corrupted. By using a random number generator with a state internal to RedMPI (without effect on the MPI application), the injector randomly picks messages to corrupt. Once targeted for corruption, RedMPI selects a random bit within the message and flips it prior to sending it out. RedMPI is agnostic to the data type of the message, i.e., the injector calculates the total number of bits within the entire message regardless of type or count before picking a bit to flip.

Note that not only does the fault injector flip a bit in the send buffer, but it actually modifies the application's memory directly. If the MPI application accesses the same memory again, further calculations based on that data will be invalid with a high probability of causing further divergence from non-corrupted replicas.

A. Targeted Fault Injections

Memory corruption faults may also be specifically targeted to occur within specific sets of replicas, MPI ranks, application timesteps, or frequency. For instance, as we will later investigate the effects of SDC on our experimental applications when SDC errors are allowed to propagate without any protection enabled, our targeted faults will be limited to only one set of replicas such that in dual redundancy half of the nodes may serve as "control" replicas that never

experience faults while the other "experiment" replicas will receive faults. By modifying MPI applications to report back to RedMPI whenever they reach a new timestep, we can also target faults to occur at very specific points in execution such as defining a desired timestep for an SDC injection to occur.

V. Experimental Framework

We deployed RedMPI on a medium sized cluster and utilized up to 96 nodes for benchmarking and testing. Each compute node consists of a 2-way SMPs with AMD Opteron 6128 (Magny-Cours) processors of 8 cores per socket (16 cores per node) with 32 GB RAM per node. Nodes are connected via 1Gbps Ethernet for user interactions and management. MPI transport is provided by a 40Gb/s InfiniBand fat tree interconnect. To maximize the compute capacity of each node, we ran up to 16 processes per node.

When launching RedMPI jobs, we map replica processes so that they do not reside on the same physical nodes. This type of mapping is preferred as a fault on a node will not affect multiple replicas of the same process simultaneously (i.e., due to localized power failures for a whole rack).

A. Time Overhead Experiments

RedMPI allows applications to utilize transparent 2x or 3x redundancy. While the physical cost of redundancy is known (2x or 3x the number of tasks), the additional cost in terms of wall-clock time should be investigated to determine what types of costs are expected, if any. To determine the cost of redundancy in terms of time, we run a variety of applications demonstrating differing scaling, processor counts, communication patterns, and problem sizes.

For our timing experiments, we solely report benchmark results for the MsgPlusHash SDC method as it provides a more efficient communication protocol than All-to-all by design. To provide meaningful metrics, each experiment assesses the run time for regular, unaltered Open MPI (referred to as *1x* in the results tables), RedMPI with dual redundancy (*2x*), and RedMPI with triple redundancy (*3x*). Note that the size reported is the size without redundancy in all experiments and results. Hence, a size 512 job in our results with triple redundancy is actually running across 1536 processors.

We assess both strong and weak scaling when evaluating overheads associated with RedMPI. For each weak scaling application, the input data size remains constant for each process no matter how many processes are run. In contrast, strong scaling applications have a constant problem size for a given class that varies the amount of input data each process receives when jobs of differing sizes are run. In effect, we expect strong scaling applications to reduce the amount of data and computation required per process as the number of processors increases.

Our test suite of weak scaling applications includes LAMMPS, ASCI Sweep3D, and HPCCG. LAMMPS is a popular molecular dynamics code that we evaluate with two different problems, "chain" and "chute". Sweep3D is a neutron transport code. Finally, HPCCG is a finite elements application from the Sandia National Labs Mantevo Project. It was chosen because of its use of `MPI_ANY_SOURCE` to demonstrate RedMPI's capability to handle non-deterministic MPI operations.

The strong scaling NAS Parallel Benchmarks (NPB) are also evaluated with varying problem class sizes and number of processes. We use the NPB suite to demonstrate how varying the communication-to-computation ratio affects RedMPI in some cases.

VI. Results

Tables IV-XII report execution time for the benchmarked applications. Every application was run with three different MPI sizes. For all of the cases except one, we conducted experiments with 128, 256, and 512 processors for the baseline. The uninstrumented (no RedMPI) version of each application is shown under the *1x* column while the *2x* and *3x* columns represent dual and triple redundancy, respectively, under RedMPI. The final two columns represent the percent overhead incurred by adding dual or triple redundancy relative to the baseline. Runs with redundancy use two or three times as many processes as the uninstrumented baseline runs. Performance is subject to cache effects when running the same application with RedMPI. This effect may vary between degrees of redundancy. This is evident for results that indicate small negative overheads (speedup under redundancy) when averaged, such as in Table IV.

We first analyze the runtime results of the weak scaling applications in Tables IV-VII. LAMMPS with input *chain* was run with a dataset size of 32x40x20 for 512 processors and scaled down proportionally for 256 and 128 processors, which explains the relatively consistent runtimes. Likewise, LAMMPS with input *chute* was given a dataset size of 320x88 for 512 processors and also scaled proportionally. Sweep3D had an input size of 320x40x200 and HPCCG had an input size of 400x100x100.

These applications performed very well with RedMPI, i.e., in most cases the RedMPI overhead was not perceptible due to a well balanced communication to computation ratio that weak scaling allowed us retain despite increasing the number of processors as we scaled up the benchmarks with RedMPI.

To demonstrate the effectiveness of RedMPI's wildcard support, HPCCG was chosen as it makes use of `MPI_ANY_SOURCE` receives. Since RedMPI requires special handling for wildcards, the overheads incurred may vary based on how long it takes the replicas to receive an envelope message that resolves the wildcard. When wildcard resolution is completed quickly, very little performance penalty is seen as in the *2x* results with size 128 (see Table VII). Conversely, when wildcard resolution takes a relatively long time, then RedMPI forces MPI to receive all messages in an

Table IV
LAMMPS INPUT CHAIN.SCALED

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128	240.5	241.34	242.54	-3.8%	-3.3%
256	244.39	244.61	245.25	0.1%	0.4%
512	250.93	251.89	256.11	0.4%	2.1%

Table V
LAMMPS INPUT CHUTE.SCALED

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128	137.50	138.38	139.01	0.6%	1.1%
256	138.26	140.43	140.00	1.6%	1.3%
512	139.19	140.22	140.67	0.7%	1.1%

Table VI
SWEEP3D

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128	390.30	389.49	393.05	-0.2%	0.7%
256	428.17	427.53	431.20	-0.1%	0.7%
512	488.08	488.93	494.09	0.2%	1.2%

Table VII
HPCCG

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128	99.79	99.76	125.75	0.0%	26.0%
256	99.64	128.83	131.02	29.3%	31.5%
512	126.36	146.19	152.26	15.7%	20.5%

Table VIII
NPB CG

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128-D	201.42	205.87	215.51	2.2%	7.0%
256-D	127.21	132.61	136.64	4.2%	7.4%
512-D	70.10	77.54	83.67	10.6%	19.4%

Table IX
NPB EP

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128-D	72.31	72.63	72.74	0.4%	0.6%
256-E	579.94	581.02	581.27	0.2%	0.2%
512-E	289.80	290.83	291.30	0.4%	0.5%

Table X
NPB FT

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
32-C	117.45	117.95	118.68	0.43%	1.05%
64-C	68.82	68.62	71.77	-0.29%	4.29%
128-D	222.75	228.76	234.97	2.70%	5.49%

Table XI
NPB LU

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128-D	361.78	379.90	375.44	5.0%	3.8%
256-D	179.78	191.97	195.55	6.8%	8.8%
512-D	102.90	115.07	121.01	11.8%	17.6%

Table XII
NPB MG

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128-E	339.17	340.41	429.67	0.4%	26.7%
256-E	168.56	170.68	171.48	1.3%	1.7%
512-E	66.97	68.35	69.29	2.1%	3.5%

unexpected queue. This can degrade performance.

The remaining NPB benchmarks are strong scaling applications subjected to input classes C, D, and E, where E is the largest size. For each class, is equally distributed across all MPI processes. For example, the CG and LU benchmarks were both run with the same class size *D* for 128, 256, and 512 size jobs. We can see that as the number of processes increases, the baseline runtime decreases since there is less computation per process to perform. In turn, as the computation is decreased per process the amount of communication incurred does, in fact, increase. Tables VIII and XI clearly demonstrate how the overhead of RedMPI increases as the per-process communication overshadows the per-process computation. Hence, to keep RedMPI overheads reasonable, it is important to choose input classes such that the ratio of communication to computation is balanced, e.g., as seen for input sizes for EP and FT. For the FT, we were unable to run class E with size 256 and 512 because our experimental setup did not have enough memory available to hold class E. Thus, we chose to run FT with smaller class sizes and report smaller runs.

Overall, RedMPI's runtime overheads are modest for well-behaved applications that can be scaled to exhibit a fair communication to computation ratio.

VII. Fault Injection Studies

RedMPI's fault injector provides two key opportunities for specifically analyzing silent data corruption faults within the scope of running MPI applications. We will use RedMPI to answer these questions:

(1) **Propagation:** Does SDC affect applications messages and correctness when no protection mechanisms (such as RedMPI's voting) are available? How quickly do SDC injections propagate to other processes via communication? Do corrupted processes further disrupt other processes in a cascading manner by sending invalid, divergent MPI messages as compared to the correct execution of a job?

(2) **Protection:** When utilizing triple redundancy with RedMPI, are SDCs successfully detected and corrected? Do applications still complete with correct answers even in the face of SDC injections?

A. SDC Propagation Study

Our first study investigates whether leaving message data unprotected in the face of SDCs does in fact lead to incorrect results. This happens when a single SDC injection in one process will later spread to other nodes causing an overwhelming cascade of invalid data.

As described in Section IV-A, we run RedMPI with dual redundancy and assume two sets of replicas. The first set of replicas is a control set and will not receive any SDC injections. The control set will execute normally and should produce correct results upon completion. Our second set of replicas is the test set, which becomes the victim when SDC faults are injected. Further, during these experiments we also disable RedMPI's corrective capabilities. RedMPI still detects divergent messages between the control and test set replicas, but allows application progress to continue. RedMPI tracks live statistics on applications running in this environment such as:

- which processes receive SDC injections;
- which processes send bad messages and where;
- which processes receive corrupt messages and how that corruption further spreads to and from nodes that were indirectly tainted by a bad message; and
- aggregate data at the granularity of a single application-defined timestep.

This type of reporting is made possible by redundancy and is considered a new technique for live application analysis and correctness since we do not need to actually log data for later viewing; this is advantageous for long-running applications

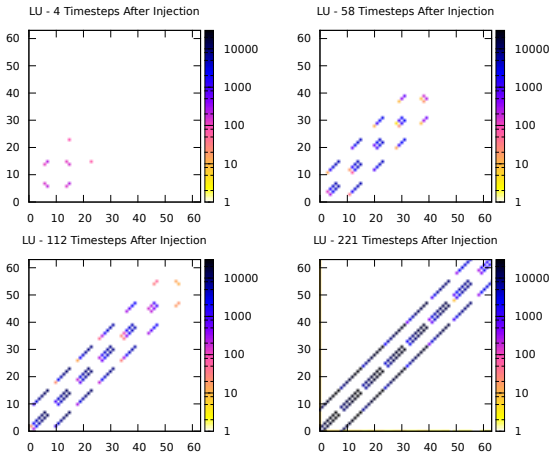

Figure 9. NPB LU Corrupted Communication Patterns

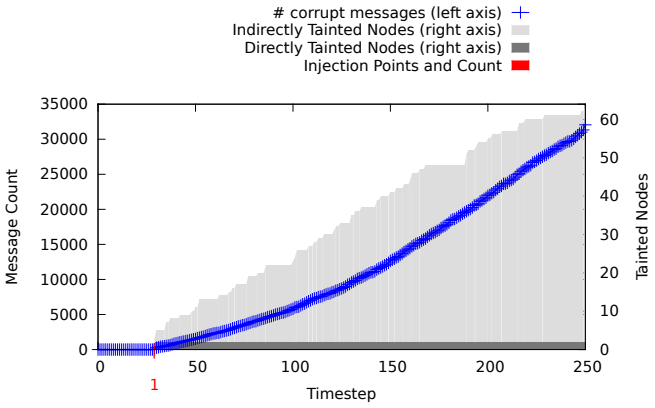

Figure 10. NPB LU Overview of Corrupt Nodes and Messages

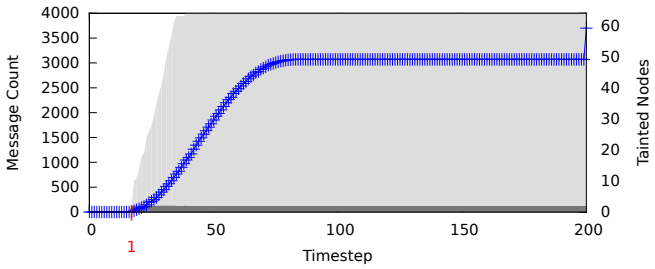

Figure 11. NPB BT Overview of Corrupt Nodes and Messages

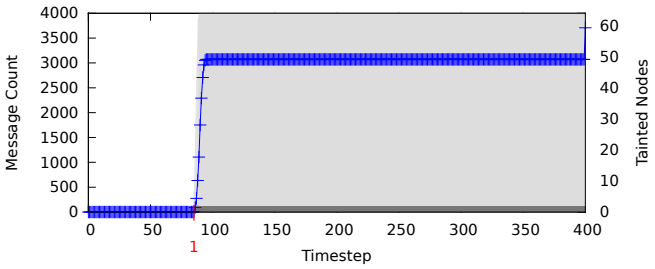

Figure 12. NPB SP Overview of Corrupt Nodes and Messages

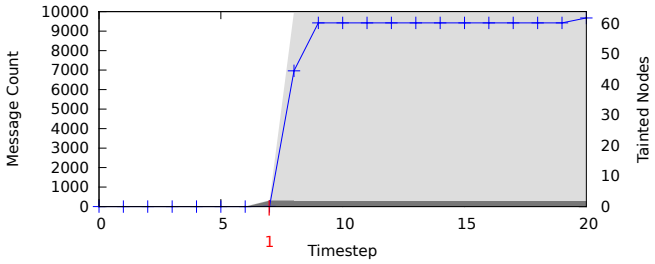

Figure 13. NPB MG Overview of Corrupt Nodes and Messages

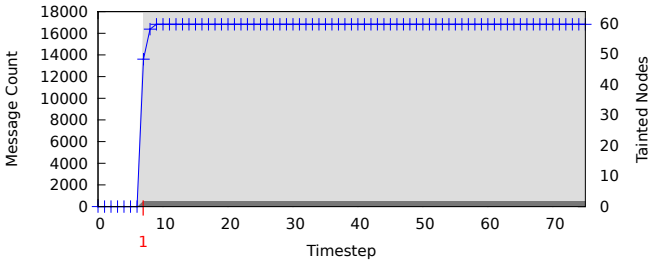

Figure 14. NPB CG Overview of Corrupt Nodes and Messages

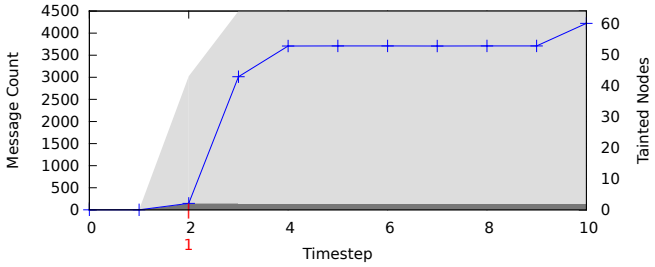

Figure 15. ASCI Sweep3D Overview of Corrupt Nodes and Messages

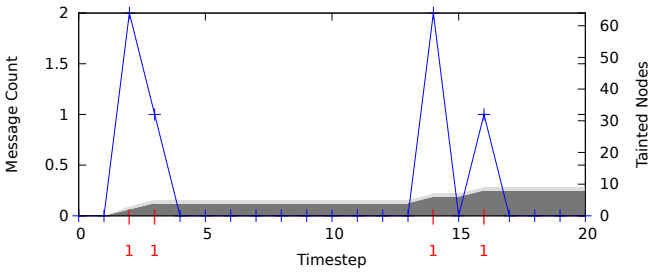

Figure 16. NPB FT Overview of Corrupt Nodes and Messages

or when bandwidth is high and logging to disk for offline analysis is undesirable.

Figures 10-16 show how SDC injection(s) spread causing bad messages and cascading tainting of other processes via reception of corrupt messages. The x-axis of these graphs denotes progress in the form of application-reported timesteps. The blue line correlates with the left vertical axis, which denotes the number of bad messages that were cumulatively received by all MPI processes in any given timestep. The gray filled areas match to the right axis and denote the number of MPI processes that receive an SDC injection themselves (direct tainting) or become corrupt indirectly (indirect tainting/light gray) by receiving a corrupt message from a previously corrupted sender themselves. Combined, both of the gray filled areas indicate how many of the MPI processes operating in a single job have become corrupt over time.

For all of the experiments except NPB FT (Figure 16), only a single SDC injection was performed. NPB FT was subjected to 4 SDC injections. All SDC injections were randomly inserted using the aforementioned RedMPI fault injector. In all cases we ran the experiments with 64 MPI processes under dual redundancy (64 control set processes + 64 test set processes).

These 7 graphs indicate three disjoint trends in response to SDC injections. First, the *progressive* trend is characterized by Figures 10 and 11 as a single injection that does not spread immediately. Instead, *progressive* applications often communicate with their grid neighbors resulting in a sphere of corruption that grows outward. For example, Figure 9 shows a heatmap indicating nodes (both on the x-and y-axes) that communicate tainted messages with one another due to an SDC. The figures depict timesteps 4 (one timestep after first injection), 58 (more indirectly tainted messages), 112 (most nodes tainted), and 221 (all nodes tainted) after the SDC injection. As the application progresses, the final heatmap shown is actually what a heatmap of normal communication. This indicates that eventually virtually all communication has succumbed to invalid data due to a single SDC injection.

The next trend we identify is the *immediate/explosion* case. Of the experiments reported, Figures 12-15 fall in this category. For these applications we noticed that a single injection resulted in corruption that spread across all nodes usually within two timesteps of the initial SDC. The most common reason for the *immediate* trend is the use of collectives or some communication pattern that tends to exchange messages between all nodes in a short period of time. Heatmaps of this trend are not provided as they essentially mirror the full communication pattern of the application almost immediately after the injection occurs.

Our third identified trend is the *localized* case. In Figure 16, NPB FT is one such application where injections result in just a few invalid messages. This occurs when the corrupted data is neither reused nor retransmitted, which in turn keeps the sphere of corruption relatively isolated to the processes that were tainted by a direct SDC injection. In this experiment we targeted FT with multiple injections to demonstrate how applications that fit the *localized* trend typically do not result in large aggregates of tainted messages or nodes. Although we did not receive a high degree of corrupt messages or spreading for FT, it is important to note that all benchmarks (including NPB FT) failed to pass their internal verification or complete with correct results that matched their "control" counterparts.

In summary, by observing that just a single SDC injection can induce a profound effect on all communicating processes, we conclude that protecting applications at the MPI message level is an appropriate method to detect, isolate, and prevent further corruption. Had RedMPI's protection not been purposefully disabled for this study, then all of the SDC injections would have been isolated from spreading and no bad messages could have been received by other processes.

B. RedMPI SDC Protection Study

To gauge SDC sustainability when RedMPI is active with redundancy, we inject faults into the running benchmarks to determine if the faults are detected and if correction succeeds. Additionally, we experimentally determine if the fault corrections allow the benchmarks to complete their self-verification process successfully.

Next, we analyze the effectiveness of SDC detection and correction protocols. We ran the fault injector with a corruption frequency of $1/5,000,000$ messages to ensure a relatively high likelihood for an injection while running the CG benchmark with 64 processes (virtual ranks) and a replication degree of three (192 physical processes). Note that we restricted the intentional corruption injections to only occur on replicas with a replica rank of zero to control the experiment. This ensures that at least two of the three replicas do not receive an injection in all cases. Thus, voting results in a valid outcome so that invalid messages can be corrected. During ten experiments with a frequency of $1/5,000,000$, we encountered one occasion with two injections, four occasions with a single injection, and five occasions without injections. In every run except one, the corruption resulted in a single bad message that was successfully detected and corrected by the receiving replicas. In one event, a single injection cascaded resulting in 6,242 bad messages originating from the corrupted sender. Nevertheless, the receiving replicas were able to correct the messages as they arrived. Eventually, the corrupted node ceased to send corrupted messages as the application finished traversing data structures until the fault was no longer touched. In these experiments, the applications progressed until completion and successfully passed their built-in verification at the end of processing.

Following that experiment, we performed injections with a frequency of $1/2,500,000$ in another ten runs that were not limited to a single group of replicas. By doubling the odds for an injection and removing the process selection restriction, we detected a significantly larger number of faults. On average, we received 2.5 injections and several thousand invalid messages per run as a result. Nonetheless, RedMPI carried all but two runs to a successful completion with verification. Of the two runs that failed, we observed that when two of the three replicas simultaneously transmitted a corrupt message over RedMPI, it was detected. The voting process is then forced to fail. In this case, RedMPI aborts because voting becomes impossible with three unique messages and hashes. Statistically, as more processes are added to a job, the likelihood of two processes with the same virtual rank becoming corrupt decreases. Therefore, observing results similar to this particular controlled experiment decreases as job sizes and replication degrees increase. Nonetheless, the longer a process remains in an invalid state (i.e., sending bad messages), the longer the correction features of RedMPI are impaired. However, it is important to note that RedMPI still forces a job to abort if it does detect corruption across two or more nodes while utilizing triple redundancy.

Performance of SDC correction has proved to be quite efficient. During SDC correction overhead experiments, we discovered that with as few as three injections we were able to produce nearly 100,000 invalid messages from corrupted senders. The receiving replicas were able to successfully detect and correct each invalid message while effectively generating no perceived overhead. In fact, while running 20 experimental iterations to gauge the protocol overhead of correcting MsgPlusHash messages during injection, the runtime was 0.31 seconds less than the original experiment runtime without fault injection on average.

Realistically, we do not expect to encounter such a high number of naturally occurring SDCs for a small environment such as our benchmarking cluster. The actual overhead incurred due to SDC correction is a function of the number of invalid messages and the distribution of such messages over nodes. The number of invalid messages is highly dependent on the data reuse patterns of an MPI application. For example, an application that never reuses send buffer data only incurs a single invalid message in the event that a buffer is corrupt. On the other hand, if an entire application depends on reuse of data stored in a buffer, the number of invalid messages could quickly exceed the valid messages in this type of program design.

As empirical evidence of the success of RedMPI, we discovered that RedMPI was detecting and correcting MPI transmission errors during our timing benchmarks even though we had not activated our fault injection module. While investigating, we learned that RedMPI had been properly detecting and correcting faulty memory that was later confirmed to be producing errors, which could not be corrected by ECC alone. These problems were occasionally (but not always) visible through the Linux EDAC monitoring module of the memory controller. Interestingly, EDAC was unable to consistently detect all of the errors that RedMPI detected. Using RedMPI, we discovered which of the 1536 compute cores in the experiment was faulting and were able to reproduce similar experiments that consistently produced faulty MPI messages on this node before removing it from the production system. Without RedMPI, this failing hardware may have gone unnoticed for some time.

VIII. Related Work

Since the early 1990s [23], fault tolerance in large-scale HPC systems is primarily assured through application-level checkpoint/restart (C/R) to/from a parallel file system. Support for C/R at the system software layer exists, e.g., via Berkeley Lab Checkpoint Restart (BLCR) [24] or diskless C/R via Scalable C/R (SCR) [25]. Message logging, algorithm-based fault tolerance, proactive fault tolerance, and Byzantine fault tolerance have all been researched in the past. Redundancy in HPC, as showcased in this paper, has only been recently explored.

Historically, the primary defense against SDC has been ECC in memory. In today's memory modules and processors, single-error correction (SEC) double-error detection (DED) ECC protects against single event upset (SEU) bit flips as well as single event multiple upset scenarios. Chipkill offers additional protection against wear-out and complete failure of a memory module chip by spanning ECC across chips but Bose Chaudhuri-Hocquenghem (BCH) encoding provides better energy-delay characteristics [26]. Software redundancy may provide more extensive SDC protection, especially considering the expected increase in SECDED ECC double-error rates.

Pure software-based solutions [27] try to protect against memory corruption without extending hardware ECC. However, they cannot provide perfect coverage to all memory and are subject to job failure if just a single process terminates due to a fault. In contrast, redundancy for SDC correction survives single process faults more gracefully.

Studies primarily done at Los Alamos National Laboratory focused on analyzing the probability and impact of silent data corruption in HPC environments. One investigation [28] showed that a Cray XD1 system with an equivalent number of processors as the ASCI Q system would experience one SDC event every 1.5 hours. Another study [29] at Lawrence Livermore National Laboratory investigated the behavior of iterative linear algebra methods when confronted with SDC in their data structures. Results show that linear algebra solvers may take longer to converge, not converge at all, or converge to a wrong result.

Modular redundancy has been used in information technology, aerospace and command & control systems [30]. Recent software-only approaches [31], [32] focused on

thread-level, process-level and state-machine replication to eliminate the need for expensive hardware. The sphere of replication [33] concept describes the logical boundary of redundancy for a replicated system. Components within such a sphere are protected; those outside are not.

Recent work [34] studied the impact of deploying redundancy in HPC systems. Redundancy can significantly increase system availability and correspondingly lower the needed component reliability. Redundancy applied to a single computer decreases the MTTF of each replica by a factor of 100-1,000 for dual redundancy and by 1,000-10,000 for triple redundancy without lowering overall system MTTF. If a failed replica is recovered through rebooting or replacing with a hot spare, replica node MTTF can be lowered by a factor of 1,000-10,000 for dual and by 10,000-100,000 for triple redundancy. Redundancy essentially offers a trade-off between component quality and quantity. Our work in this paper permits this trade-off.

Another compelling study [8] uses an empirical assessment of how redundant computing improves time to solution. The simulation-driven study looked at a realistic scenario with a weak-scaling application that needs 168 hours to complete, a per-node MTTF of five years, a fixed five minutes to write out a checkpoint, and a fixed ten-minute time to restart. Checkpointing is performed at an optimal interval. The results show that at 200k nodes, an application spends eight times the time required to perform the work, reducing the throughput of such a machine to just over 10% compared to a fault-free environment. In contrast, using 400k nodes and dual redundancy, the elapsed wall clock time is 1/8 of that for the 200k-node non-redundant case. The throughput of the 400k-node system is four times better with redundant computing than the non-redundant 200k-node system. The prototype detailed in this paper is a step toward achieving this capability.

rMPI [35] is a prototype for redundant execution of MPI applications that uses the MPI profiling interface (PMPI) for interpositioning. rMPI maintains redundant nodes and and each replica duplicates the work of its partners. In case of a node failure, the redundant node continues without interruption. The application fails only when two corresponding replicas fail. The reported impact on actual applications is for the most part negligible [8]. Our work differentiates itself from rMPI in that it takes the research in a new direction using replication to detect and correct silent errors. The protocols necessary for this detection and correction and the performance impacts are quite different.

The modular-redundant Message Passing Interface (MR-MPI) [36] is a similar solution for transparent HPC redundancy via PMPI interpositioning. In MR-MPI, a redundantly executed application runs with $r * m$ native MPI processes, where r is the number of MPI ranks visible to the application and m is the replication degree. Messages are replicated between redundant nodes. The results show the negative impact of the $O(m^2)$ messages between replicas. For low-level, point-to-point benchmarks, the impact can be as high as the replication degree. In realistic scenarios, the overhead can be 0% for embarrassingly parallel or up to 70-90% for communication-intensive applications in a dual-redundant configuration. RedMPI extends beyond the capabilities of MR-MPI by protecting against SDC, lowering the replication overhead, and advancing internal communication protocols (MsgPlusHash). Unlike MR-MPI, RedMPI expands upon MR-MPI's linear collectives by providing internal MPI modifications that exploit native collectives performance.

MPIecho[37][38] is another platform for redundant computing that utilizes the PMPI layer of MPI to provide process replication. MPIecho's primary purpose is to assist in the process of debugging by monitoring hardware performance counters between replicas. Paralleling which part of the code each replica is profiling allows MPIecho to perform traditionally heavyweight debugging techniques that on their own may not scale to 10s of thousands of cores. MPIecho provides further debugging assistance by including features that verify the similarity of all MPI arguments and MPI function return values between replicas. It can also perform send buffer checking between replicas at an additional cost. MPIecho's findings show that runtime overheads range from 0% (low communication) to 103% (high communication applications) when send buffer checking is used with dual redundancy. While MPIecho may support simple buffer checking and is used as a debugging aid, RedMPI's contributions stand separately as a library intended for production use by providing high-performance message verification with an optimal communication protocol.

VolpexMPI [39] is an MPI library implemented from scratch that offers redundancy internally and uses a polling mechanism by the receiver of point-to-point messages to avoid message replication. If a polled sender (of a replicated sender-receiver pair) fails to respond, a different sender (replica of the original sender) is chosen until the receive is successful. Messages are matched with a logical timestamp to allow for late message retrieval. VolpexMPI achieves close to 80% of Open MPI's point-to-point message bandwidth, while latency of small messages increases from 0.5ms to 1.8ms. Using the NAS Parallel Benchmark suite, there is no noticeable overhead for BT and EP for 8 and 16 processes. SP shows a significant overhead of 45% for 16 processes. The overhead for CG, FT and IS is considerably higher as these benchmarks are communication heavy. VolpexMPI does not provide SDC protection, however, it provides good performance as replication protocols are part of the low-level communication inside the MPI library.

IX. Conclusion

Redundant computing is one approach to detect SDC. This study assesses the feasibility and effectiveness of SDC detection and correction at the MPI layer. We presented

two consistency protocols, explored the unique challenges in creating a deterministic MPI environment for replication purposes, investigated the effects of fault injection in to our framework, and analyzed the costs of performing SDC protection via redundancy.

This study develops a novel, efficient SDC detection and correction protocol (MsgPlusHash) with overheads ranging from 0% up to 30% for dual or triple redundancy depending on the number of messages sent by the application and the communication patterns. In particular, overheads do not change significantly for weak scaling applications as the number of processes is varied. These modest overhead ranges indicate the potential of RedMPI to protect against SDC for large-scale runs.

Our protocol detected and corrected injected faults for processes that continued to completion even when these faults resulted in many thousands of corrupted messages from a sender that experienced one or more SDC faults. In our controlled experiments, injected faults that were targeted to a single set of replicas were successfully isolated from spreading by fixing corrupted messages and allowing the applications to complete without incident. Further, when we injected faults into two or more replicas (of the same rank), RedMPI detected the corruption and was able to abort the application thus preventing invalid results from being reported.

In summary, RedMPI was successful in preventing invalid data from propagating or being transmitted without detection even under extreme scenarios. Our experiments showed profound effects from applications that experience even a single soft error without any form of protection. Without RedMPI, just one injected SDC was observed to quickly spread to other processes and messages, causing the majority of message data to become corrupt, which consistently lead to invalid results at a global scale.

Empirically, RedMPI not only performed exactly as expected in our controlled experiments, but it also pinpointed previously unknown hardware faults on our own experimental cluster nodes that had not been detected until RedMPI alerted us to SDCs occurring on unaltered MPI jobs.

While the cost of double/triple redundancy appears high in terms of power, analytic models show that for large core counts redundancy actually improves job throughput. As both the likelihood of node failure and silent data corruption increases as we scale up HPC systems, the importance of protecting data becomes obvious and available at a low cost when redundancy is already in place to ensure high throughput of mission-critical/high-consequence large scale applications.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grants CNS-1058779, CNS-0958311, DOE grant DE-FG02-08ER25837 and a subcontract from Sandia National Laboratory. Research sponsored in part by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. De-AC05-00OR22725.

REFERENCES

[1] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2009, pp. 193–204.

[2] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *USENIX Conference on File and Storage Technologies*, 2007.

[3] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150989

[4] C.-H. Hsu and W.-C. Feng, "A power-aware run-time system for high-performance computing," in *Supercomputing*, 2005.

[5] I. Philp, "Software failures and the road to a petaflop machine," in *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[6] J. T. Daly, "ADTSC nuclear weapons highlights: Facilitating high-throughput ASC calculations," Los Alamos National Laboratory, Los Alamos, NM, USA, Tech. Rep. LALP-07-041, Jun. 2007.

[7] J. T. Daly, L. A. Pritchett-Sheats, and S. E. Michalak, "Application MTTFE vs. platform MTTF: A fresh perspective on system reliability and application throughput for computations at scale," in *Proceedings of the Workshop on Resiliency in High Performance Computing (Resilience) 2008*, May 2008, pp. 19–22.

[8] K. Ferreira, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Supercomputing*, nov 2011.

[9] O. R. N. Laboratory, "Personal communications," 2010.

[10] L. L. N. Laboratory, "Personal communications," 2007.

[11] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1, 1996, pp. 293–307.

[12] J. R. Sklaroff, "Redundancy management technique for space shuttle computers," *IBM Journal of Research and Development*, vol. 20, no. 1, pp. 20–28, 1976.

[13] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *International Symposium on Microarchitecture*, 1999, pp. 196–207.

[14] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *International Symposium on Computer Architecture*, May 2003, pp. 98–109.

[15] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *International Symposium on Computer Architecture*, 2000, pp. 25–36.

[16] H. Quinn and P. Graham, "Terrestrial-based radiation upsets: A cautionary tale," in *Symposium on Field-Programmable Custom Computing Machines (FCCM) 2005*, Apr. 18-20, 2005, pp. 193–202.

[17] J. Elliot, K. Kharbas, D. Fiala, F. Mueller, C. Engelmann, and K. Ferreirar, "Combining partial redundancy and checkpointing for HPC," in *International Conference on Distributed Computing Systems*, 2012, p. (accepted).

[18] J. Vetter, "Hpc landscape — application accelerators: Deus ex machina?" Sep. 2009, invited Talk at High Performance Embedded Computing Workshop.

[19] A. Geist, "What is the monster in the closet?" Aug. 2011, invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking.

[20] J. Shalf, "Simulation challenge: Exascale planning overview," Aug. 2010, invited Talk at HEC FSIO R&D Workshop.

[21] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J. C. Andre, D. Barkai, J. Y. Berthou, T. Boku, B. Braunschweig, and et al., "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[22] S. Böhm and C. Engelmann, "File i/o for mpi applications in redundant execution scenarios," in *Euromicro International Conference on Parallel, Distributed, and network-based Processing*, Feb. 2012.

[23] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod, "High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development," Whitepaper, Dec. 2009. [Online]. Available: http://www.csm.ornl.gov/~engelman/publications/debardeleben09high-end.pdf

[24] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," in *Journal of Physics: Proceedings of the Scientific Discovery through Advanced Computing Program (SciDAC) Conference 2006*, vol. 46. Denver, CO, USA: Institute of Physics Publishing, Bristol, UK, Jun. 25-29, 2006, pp. 494–499. [Online]. Available: http://www.iop.org/EJ/article/1742-6596/46/1/067/jpconf6_46_067.pdf

[25] G. Bronevetsky and A. Moody, "Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O," Lawrence Livermore National Laboratory, Livermore, CA, USA, Tech. Rep. TR-JLPC-09-01, Aug. 2009. [Online]. Available: http://dx.doi.org/10.2172/964079

[26] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi, "System implications of memory reliability in exascale computing," in *Supercomputing*, 2011, pp. 46:1–46:12.

[27] D. Fiala, K. Ferreira, F. Mueller, and C. Engelmann, "A tunable, software-based dram error detection and correction library for hpc," in *Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids*, Sep. 2011, pp. 110–121.

[28] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer," *IEEE Transactions on Device and Materials Reliability (TDMR)*, vol. 5, no. 3, pp. 329–335, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1545893

[29] G. Bronevetsky and B. R. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the* 21^{st} *ACM International Conference on Supercomputing (ICS) 2008*. Island of Kos, Greece: ACM Press, New York, NY, USA, Jun. 7-12, 2007. [Online]. Available: http://greg.bronevetsky.com/papers/2008ICS.pdf

[30] D. P. Siemwiorek, "Architecture of fault-tolerant computers: An historical perspective," *Proceedings of the IEEE*, vol. 79, no. 12, pp. 1710–1734, 1991. [Online]. Available: http://dx.doi.org/10.1109/5.119549

[31] A. Golander, S. Weiss, and R. Ronen, "DDMR: Dynamic and scalable dual modular redundancy with short validation intervals," *IEEE Computer Architecture Letters*, vol. 7, no. 2, pp. 65–68, 2008. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/L-CA.2008.12

[32] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "PLR: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 6, no. 2, pp. 135–148, 2009. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TDSC.2008.62

[33] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the* 29^{th} *Annual International Symposium on Computer Architecture (ISCA) 2002*. Anchorage, AK, USA: IEEE Computer Society, May 25-29, 2002, pp. 99–110. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ISCA.2002.1003566

[34] C. Engelmann, H. H. Ong, and S. L. Scott, "The case for modular redundancy in large-scale high performance computing systems," in *Proceedings of the* 8^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*. Innsbruck, Austria: ACTA Press, Calgary, AB, Canada, Feb. 16-18, 2009, pp. 189–194. [Online]. Available: http://www.csm.ornl.gov/~engelman/publications/engelmann09case.pdf

[35] R. Brightwell, K. B. Ferreira, and R. Riesen, "Transparent redundant computing with MPI," in *EuroMPI*, ser. Lecture Notes in Computer Science, R. Keller, E. Gabriel, M. M. Resch, and J. Dongarra, Eds., vol. 6305. Springer, 2010, pp. 208–218.

[36] C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *Proceedings of the* 10^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*. Innsbruck, Austria: ACTA Press, Calgary, AB, Canada, Feb. 15-17, 2011.

[37] B. Roundtree, G. Cobb, T. Gamblin, M. Schulz, B. Supinski, and H. Tufo, "Parallelizing heavyweight debugging tools with mpiecho," in *High-performance Infrastructure for Scalable Toolsi, WHIST 2011, Held as part of ICS '11, Tucson, Arizona*, 2011, pp. 803–808.

[38] G. Cobb, B. Roundtree, H. Tufo, M. Schulz, T. Gamblin, and B. de Supinski, "Mpiecho: A framework for transparent mpi task replication," Dept. of Computer Science, University of Colorado at Boulder, Tech. Rep. CU-CS-1082-11, Jun. 2011.

[39] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok, "Volpexmpi: An MPI library for execution of parallel applications on volatile nodes," in *Lecture Notes in Computer Science: Proceedings of the* 16^{th} *European PVM/MPI Users' Group Meeting (EuroPVM/MPI) 2009*, vol. 5759. Espoo, Finland: Springer Verlag, Berlin, Germany, Sep. 7-10, 2009, pp. 124–133. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_19