SAND2011-0533C

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

David Fiala, Frank Mueller
North Carolina State University
{dfiala|fmuelle}@ncsu.edu

Christian Engelmann
Oak Ridge National Laboratory
engelmannc@ornl.gov

Rolf Riesen, Kurt Ferreira
Sandia National Laboratory†
{rolf|kbferre}@sandia.gov

ABSTRACT

Faults have become the norm rather than the exception for high-end computing on clusters with 10s/100s of thousands of cores, and this situation will only become more dire as we reach exascale computing. Exacerbating this situation, some of these faults will not be detected, manifesting themselves as silent errors that will corrupt memory while applications continue to operate but report incorrect results. This paper introduces RedMPI, an MPI library residing in the profiling layer of any standards-compliant MPI implementation. RedMPI is capable of both online detection and correction of soft errors that occur in MPI applications without requiring any code changes to application source code. By providing redundancy, RedMPI is capable of transparently detecting corrupt messages from MPI processes that become faulted during execution. Furthermore, with triple redundancy RedMPI additionally "votes" out MPI messages of a faulted process by discarding and replacing corrupted results with corrected results from unfaulted processes. We present an experimental evaluation of RedMPI on an assortment of applications to demonstrate the effectiveness and assess overheads associated with this approach.

RedMPI experimental results reveal overheads between 13% and 62% depending on the desired level of redundancy and MPI application communication patterns. Fault injection experiments establish that RedMPI is not only capable of successfully detecting injected faults, but can also correct these faults while carrying a corrupted application to successful completion without propagating invalid data. To our knowledge this is the first design, implementation and evaluation of a runtime system to detect and correct silent data corruption for high-end computing systems.

*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.

†Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

1. 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) [32], (ii) 1-5% of disk drives die per year [26] and (iii) DRAM errors occur in 2% of all DIMMs per year [32], which is more frequent than commonly believed.

Table 1: Reliability of HPC Clusters [17]

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 1, 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 (such as cores, memory and 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 [25], 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 [5, 6] 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.

Table 2: 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%

A study by Sandia National Lab from 2009 [12] shows rapidly decaying useful work for increasing node counts (see Table 2). 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 3 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 restarts.

Table 3: 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 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 [12]. 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 3) and four times the number of messages [12]. But once restart and rework overheads exceed 50%, redundancy is actually *cheaper* than traditional C/R at large core counts.

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, will force an instant reboot after detection since such faults cannot be corrected by ECC. 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 100,000+ DIMMs) [19].

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 [18], 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 [23, 1, 37, 38, 35, 2, 14, 21, 28, 29, 30, 31, 36].

Today, the frequency of bit flips is no longer believed to be dominated by single-event upsets due to radiation from space [27] 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 [32]. With SDCs occurring at significant rates, not every bit flip will result 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 3.

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).

Contributions: The main contribution of this work is the design and implementation of efficient and transparent mechanisms for fault tolerance (FT) in large-scale HPC environments for SDC detection and correction. The key to success is to not only to rely on techniques to react to failures imposing restart overheads but to also sustain failures such that computation progresses seamlessly 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?

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 silent data corruption (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 will 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.

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.

2. DESIGN

In this paper we present 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 corrupt messages, or run in triple redundant mode and also correct faulty messages. 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.

It is important to note where and how SDC faults are detected. RedMPI solely analyzes the content of MPI messages for any possibility of divergence between replicas during communication. When a divergence is detected, the result deemed to be invalid will be thrown out 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 will eventually be 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 will detect it when the transmission occurs, independent of when or how the SDC originated.

2.1 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, one needs to verify that a point-to-point message (e.g., `MPI_Isend`) sent from an MPI process is identical to the message sent by other replica processes 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 usage. 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 presents the time lost on the receiver side before the application can proceed. For this reason, RedMPI's protocols use a receiver-side verification method resulting in faster message delivery with considerably reduced message latency.

2.2 Assumptions

RedMPI does not protect messages over a transport layer such as TCP or InfiniBand, and assumes the transport it utilizes to be reliable. Due to this assumption, RedMPI does not handle corruption due to transport failure. An unreliable network could cause undefined behavior and deadlock RedMPI.

Many SDCs that occur will affect the data sections of running applications, but there remains a chance that corruption could change the code section instead. While RedMPI makes every attempt to continue uninterrupted execution when corruption occurs, if the thread of execution for a process diverges from the other replicas then it may not be possible to maintain identical MPI communication patterns, which would lead RedMPI to fault.

3. 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*, which intercepts MPI function calls and directs them to RedMPI. The profiling layer provides a standard API that allows libraries to wrap all MPI calls and add additional or replacement logic in place of the original functionality.

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 will need to be launched. For example, to launch an MPI job that normally requires 128 processes will instead require 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 1 shows how triple redundancy may appear within an MPI application expecting a size of three.

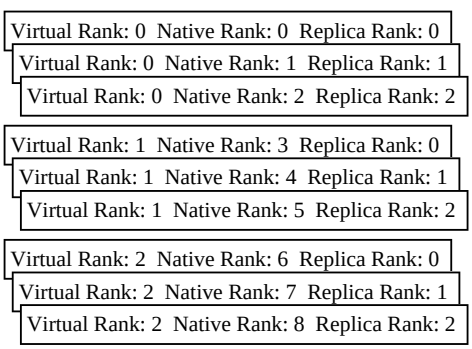


Figure 1: Internal Ranks of a 3 Process MPI Application with Triple Redundancy

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.

3.1 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 or group created within the MPI application another map will be created to track ranks within the new group.

Native to virtual mappings:

$virtual\_rank = native\_rank \bmod virtual\_size$
$replica\_rank = native\_rank / virtual\_size$

Virtual to native mapping:

$native\_rank =$
$\quad virtual\_rank + (replica\_rank \times virtual\_size)$

Figure 2: Rank Mapping Formulas

Internally, mappings can be translated using a formula (Figure 2) or by storing the data in a lookup structure. The formula given provides a simple, deterministic method with low memory requirements, but it is not capable of providing fine-tuned control of rank mapping. By using a custom rank map file and passing it 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.

3.2 Message Corruption Detection and Correction

3.2.1 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 would send three messages where one message goes to each replica receiver as demonstrated by Figure 3. This means that for a degree of 2 or 3 the number of messages actually sent for a single `MPI_Isend` would be 4 or 9, respectively. On the receiving side, each receiver would listen for a message from each sender replica and place such messages in separate receive buffers.

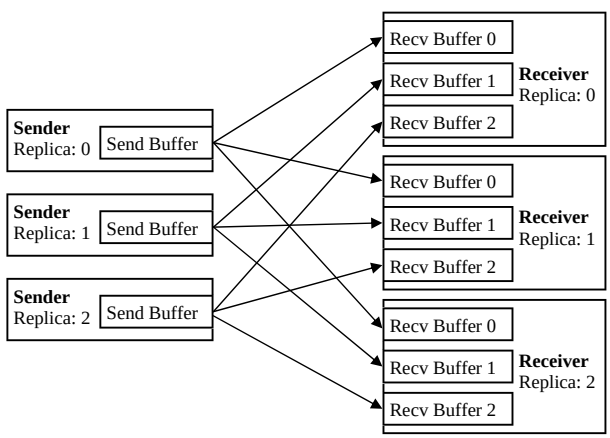


Figure 3: All-to-all Method Overview

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 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 4 visualizes this process.

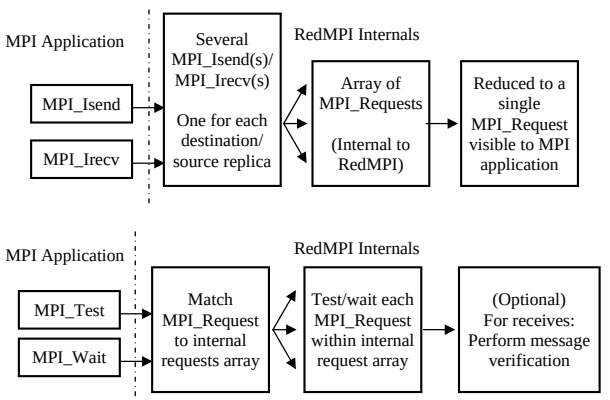


Figure 4: All-to-all Function Overrides

Following an `MPI_Isend` or `MPI_Irecv`, an MPI application will usually complete 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 imposes 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. Alternatively, a request from an `MPI_Irecv` requires extra steps in order to verify the messages received from each replica.

One point of interest is how the replica receive buffers are allocated. To reduce storage overheads, the first receive buffer is always a pointer to the MPI application's actual receive buffer. Any additional buffers required in response to an increase in the degree of replication is internally allocated by RedMPI. This approach not only saves buffer space, but it also avoids the need to ever copy a message from a RedMPI buffer to the application's buffer. An exception to this is made if RedMPI detects that the first replica receive buffer was determined to be corrupted, in which case a memory copy is necessary to provide a corrected copy of the message to the MPI application.

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 by computing a `SHA1` hash of each replica receive buffer and then comparing the hashes themselves. Using a hash ensures that message data is not read multiple times, which would incur excessive overhead for large messages. Under normal conditions, when no corruption is detected, the extra buffers are freed and control is returned to the MPI application.

If, during message verification, a buffer mismatch is detected, RedMPI will mitigate 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. If the first buffer was verified as correct but a later buffer was not, then it is not necessary to initiate a copy as the MPI application will only access the first buffer.

3.2.2 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 will later be used to verify each receiver's message.

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. For example, MsgPlusHash will internally utilize multiple send and receive `MPI_Request` handles, but the MPI application will only ever receive one such `MPI_Request` handle.

To check for 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 as well as a hash from an alternate replica. From an efficiency standpoint, it is not necessary to send two full messages since a hash provides sufficient means 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 is compared with a hash from a different replica. In the likely event that these hashes match, the receiver can be assured that its message is correct, i.e., no corrective action needs to be taken.

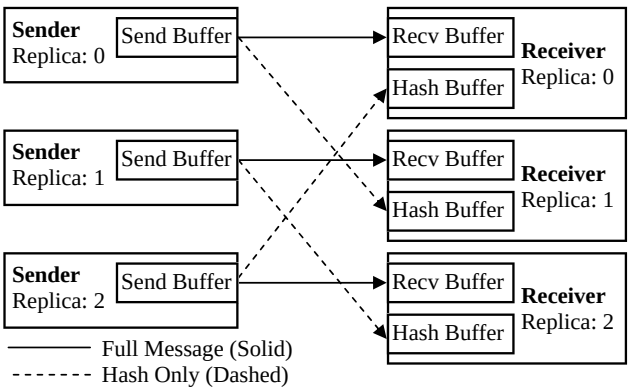


Figure 5: MsgPlusHash Method Overview

As shown in Figure 5, 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 will wrap 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 will transmit 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 will affect 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 5 that both receiver replicas 1 and 2 will be affected. In this particular case, receiver replica 1 will have received a corrupt message, but a good hash since sender replica 0 was not corrupted. Conversely, receiver replica 2 will have 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 will always be 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 the hash was corrupted. By analyzing the communication pattern, it is obvious that the replica which has a higher replica rank will always have the corrupt message with a bad hash. Therefore, the two adjacent replicas communicate with one another to to determine which of them holds a correct message. For this reason, after this handshake, the higher replica rank transmits a correction message to the lower ranked replica to complete the correction.[1]

Corrupted Adjacent Replica Discovery.

After a process encounters a message and hash mismatch, it will initiate a protocol to discover which of its adjacent replicas are also in this state. For each adjacent rank also actively trying to discover a potentially corrupted process, the other rank will engage in the discovery protocol since its message and hash did match. Such another rank is entirely unaware of the corruption elsewhere. 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.

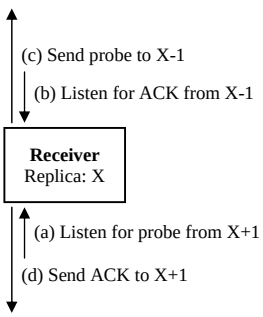


Figure 6: MsgPlusHash Correction Protocol

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 within the RedMPI interposition layer.

The nature of the discovery process creates a problem in that two unique SDCs detected by adjacent replicas at separate times may in fact send probe messages that are received

[1]Replica rank 0 wraps around to the adjacent replica of the highest degree.

in a later discovery. RedMPI accounts for this possibility by using hashes to identify whether a probe pertains to the SDC at hand. Probes that are unrelated to the current discovery process will be 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 copy of the message in their buffers and the MPI application may proceed.

3.3 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 will be 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 replica with rank zero to actually perform a real `MPI_Wtime` call. Since all replica ranks will 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`. Unlike `MPI_Wtime`, a probe may result in inconsistent results amongst 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.

Collectives.

Collective operations in MPI pose a unique challenge for corruption detection and correction. The first issue is the lack of non-blocking collectives in the MPI standard as of now. (While this short-coming is being addressed in the forthcoming MPI 3 standard by adding non-blocking collectives, our effect extends to detection and correction within the current state of collectives in MPI 2.) 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 and correct messages at the granularity of individual processes. RedMPI's solution to both issues is to map all collectives onto point-to-point operations. Via an implementation of collectives as point-to-point messages, all of the corruption detection benefits are realized by reusing the existing methods presented previously. RedMPI's goal is not to directly create the most efficient implementation of collectives over point-to-point messages, as this would replicate existing functionality of any MPI runtime without adding to the research contributions. Besides, such an effort would be nontrivial without the knowledge of the underlying communication transports. Instead, RedMPI provides a reliable and effective implementation of the original MPI collectives in a fashion that allows existing codes to enjoy the benefits of the RedMPI corruption detection methods.

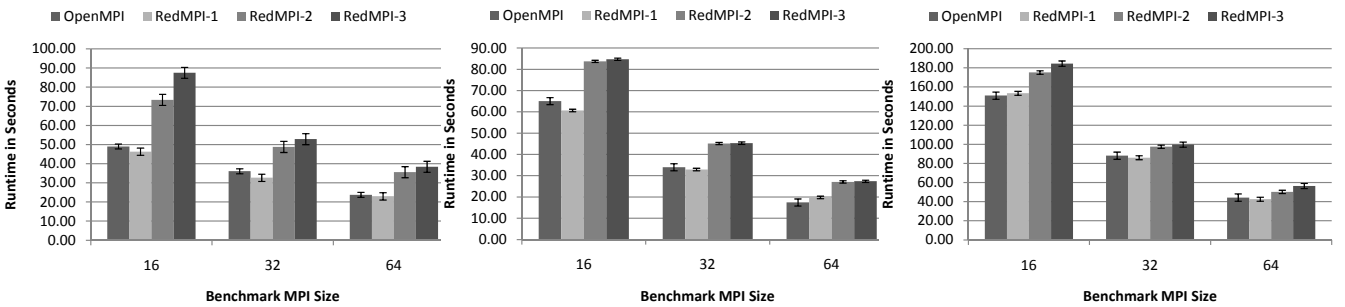
4. FAULT INJECTION FRAMEWORK

To experimentally determine the effect of corruption and verify corrective actions, a fault injector needs to be devised that can reliably 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. The fault injector designed 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 (as to not affect the MPI application itself), 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. Notably, RedMPI is agnostic to the data type of the message. This allows the injector to calculate the total number of bits within the entire message regardless of type or count before picking a random 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.

5. EXPERIMENTAL FRAMEWORK

This section describes the experiments we conducted and the computing environment on which they were performed. For benchmarking and testing purposes we deployed RedMPI on a medium sized cluster at North Carolina State University and utilized up to 24 nodes for the purposes of benchmarking. Each compute node consists of a 2-way SMPs with AMD Opteron 6182 (Magny Core) processors of 8 cores per socket (16 cores per node). Each node contains 32 gigabytes of memory. To provide networking support,



(a) CG (b) FT (c) LU

Figure 7: Comparison of OpenMPI with various RedMPI configurations on NAS Parallel Benchmarks

each node is connected via 1000Mbps Ethernet for user interactions and management. MPI transport is provided by 40Gb/s InfiniBand. To maximize the compute capacity of each node, we ran up to 16 processes per node.

To benchmark the overheads associated with RedMPI's various communication changes, the NAS Parallel Benchmarks (NPB) is evaluated for various RedMPI configurations and varying number of processes. Of the two SDC methods proposed, we solely report benchmark results for the MsgPlusHash method as it provides a more efficient communication protocol by design. To provide meaningful metrics, each experiment assesses the run time for regular, unaltered OpenMPI, RedMPI without any redundancy or SDC (RedMPI-1), RedMPI with dual redundancy (RedMPI-2), and RedMPI with triple redundancy (RedMPI-3). Each experiment is run 8 times with the average presented in our results.

Note that RedMPI-1 is not a typical scenario we would expect for normal operation, but the experimental results are of interest in order to demonstrate the performance effect of null interpositioning and the overhead imposed by RedMPI's linear collectives on our benchmarks. For this reason, RedMPI-1 bridges the gap between regular OpenMPI and RedMPI-2/3 results.

To gauge SDC sustainability when RedMPI is active with redundancy, we randomly 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 following computation.

6. RESULTS

We will first analyze the runtime results of our various RedMPI configurations as shown in Figure 7. The results are broken down by benchmark with three different Benchmark MPI sizes (number of processors) shown. In all cases, we assessed the performance for each benchmark under "class C" problem sizes (inputs). We report the wall-clock time for each benchmark execution averaged over 8 iterations but also provide minimum and maximum runtimes depicted as error bars, which illustrate the effect of background services and other OS activity.

The CG benchmark (Figure 7(a)) ran with the highest overhead of the benchmarks considered with an average overhead of 45% for dual redundancy and 62% for triple redundancy relative to a plain, unaltered OpenMPI application run. FT (Figure 7(b)) incurred a 39% average overhead for dual redundancy and 40% for triple redundancy. Finally, LU (Figure 7(c)) had the least overhead with a 13% average for dual and 21% for for triple redundancy.

An interesting point to note is the slight performance increase in RedMPI-1 compared to plain OpenMPI. As RedMPI-1 provides no redundancy, we believe that our performance of collectives is responsible for this slight performance jump. By using a simple linear point-to-point mapping of collectives, we avoid the overhead associated with using a more advanced communication pattern (e.g., trees) employed by OpenMPI to improve performance on larger-scale jobs. Since our job sizes did not span large numbers of nodes or specialized communication topologies, we believe that our circumstances allowed for improved performance given our cluster and job sizes. On large-scale systems with thousands of nodes we expect that the opposite will occur as RedMPI does not (yet) support advanced collective transmission topologies or transport-specific optimizations.

Next, we analyze the effectiveness of the SDC detection and correction protocols. We ran the fault injector with two different corruption frequencies: $1/5,000,000$ messages and $1/2,500,000$ messages to provide a relatively high likelihood that we would encounter an injection while running the CG benchmark with 64 processes (virtual ranks) and a replication degree of three. 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 with no 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, however, 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(s) 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. By doubling the odds for an injection, we observed much more dramatic events. On average, we received 2.5 injections per run with

about 3,377 invalid messages per run as a result. Nevertheless, RedMPI carried all but two runs to a successful completion with verification. Of the two runs that failed, they likely fell subject to:
(a) a simultaneous SDC in two messages between replicas
or
(b) an SDC that subsequently caused changes in data-dependent control paths resulting in differing MPI calls. (RedMPI depends on non-divergent control flow between replicas of the same virtual rank.)
In our experiment, we believe that case (a) occurred with two nodes becoming corrupt and sending differing messages. The receiving replicas would receive 3 unique messages contents (2 unique corrupt messages and 1 correct). Unable to distinguish which node is corrupt, RedMPI would be forced to fail. In this type of situation when each run incurs several thousand bad messages, the chance for recovery decreases as more senders encounter faults of type (a) or (b) above.

Performance of the SDC correction method has proved to be quite efficient. During SDC correction overhead experiments, we discovered that with as few 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, our experimental runtime average was 0.31 seconds less than the original experiment runtime average that lacked fault injection.

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 sent and the distribution of such messages over nodes. The number of invalid messages sent is highly dependent on the data reuse patterns of an MPI application. For example, an application that never reuses data from a send buffer will only incur a single invalid message in the event that a buffer is corrupt. On the other handle, if an entire application depends on reuse of data being stored in a buffer, then it is possible that the number of invalid messages would quickly exceed the valid messages in this type of program design.

7. RELATED WORK

Since the early 1990s [7], 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, such as through the Berkeley Lab Checkpoint Restart (BLCR) [15] solution, but it is only employed at a few HPC centers. Diskless C/R, *i.e.*, using compute-node memory for distributed checkpoint storage, exists as well, like the Scalable C/R (SCR) library [4], but is rarely used in practice. Message logging, algorithm-based fault tolerance, proactive fault tolerance, and Byzantine fault tolerance have all been researched in the past and are also not available in production HPC systems. Redundancy in HPC, as showcased in this paper, has only been recently explored (see below).

Historically, the primary defense against silent data corruption (SDC) has been error correcting code (ECC) in dynamic random access memory (DRAM). Only very recently, ECC has been deployed in server-market processors, such as in the AMD Opteron, to protect cache and registers as well. Single event upsets (SEUs) [11], *i.e.*, bit flips caused by natural high-energy radiation, are the dominant source of SDC. In today's memory modules and processors, single-error correction (SEC) double-error detection (DED) ECC protects against SEU as well as single event multiple upset (SEMU) scenarios. Chipkill [8] offers additional protection against wear-out and complete failure of a memory module chip by spanning ECC across chips. Manufacturers will continue exploring mitigation strategies and will be able to continue to deliver products with certain soft error resilience. However, high reliability for the latest-generation processors and memories comes at a price in terms of chip space and power consumption, and still may not be as good as today's [16]. Redundancy may provide more extensive SCD protection, especially considering the expected increase in SECDED ECC double-error rates.

Studies primarily done at Los Alamos National Laboratory (LANL) focused on analyzing the probability and impact of silent data corruption in HPC environments. One investigation [22] showed that a Cray XD1 system with an equivalent number of processors as the ASCI Q system, *i.e.*, ∼18,000 field-programmable gate arrays (FPGAs) with 16.5 TB SECDED-ECC memory, would experience one SDC event within 1.5 hours due to the high vulnerability of the FPGAs. Ongoing work at LANL focuses on radiating new-generation processors, flash, and memory with neutrons at the Los Alamos Neutron Science Center (LANSCE) to measure vulnerability and efficiency of protection mechanisms. Another study [3] at Lawrence Livermore National Laboratory (LLNL) 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. These investigations not only point out a high SDC rate when scaling up HPC systems, but also the severe impact SDC has. More extensive SDC protection is needed to assure application correctness at extreme-scale.

In general, modular redundancy (MR) transparently masks any errors without the need for rollback recovery. In case of SDC, detection is achieved through comparison and recovery is performed by majority voting. MR has been used in information technology, aerospace and command & control [34]. Recent software-only approaches [13, 33] focused on thread-level, process-level and state-machine replication to eliminate the need for expensive hardware. The sphere of replication [24] concept describes the logical boundary of redundancy for a replicated system. Components within such a sphere are protected; those outside are not. Data entering it (input) is replicated, while data leaving (output) is compared. The work in this paper relies on this concept directly in the all-to-all and indirectly in the MsgPlusHash protocol.

A recent analysis [10] studied the impact of deploying redundancy in HPC systems. System availability is a standard metric used in the information technology industry and is based on mean-time to failure (MTTF) and mean-time to recovery (MTTR): A = MTTF / (MTTF + MTTR). Redundancy can significantly increase system availability and correspondingly lower the needed component reliability, *i.e.*, the component rating by the number of nines in the component's availability rating (e.g. 99.9% as 3-nine rating). Redundancy applied to a single computer allows to decrease

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 applied to each compute node in a HPC system with 1 million nodes allows lowering the node rating from 7 to 3 nines with 2 million dual-redundant nodes and to 2 nines with 3 million triple-redundant nodes. Redundancy essentially offers a trade-off between component quality and quantity. The work presented in this paper permits this trade-off.

An even more compelling study [12] 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 200,000 nodes, an application will spend eight times the amount of 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 400,000 nodes and dual-redundancy, the elapsed wall clock time is 1/8 of than for the 200,000-node non-redundant case. The throughput of the 400,000-node system is four times better with redundant computing than the non-redundant 200,000-node system. The prototype detailed in this paper is a step toward achieving this capability.

rMPI [12] is a prototype for redundant execution of MPI applications. It is a library that gets inserted during link time between an application and the MPI library using MPI's profiling interface (PMPI). Using rMPI, an MPI application is started on up to $2n$ nodes and sees ranks $0 \ldots n-1$. rMPI transparently provides redundancy using the remaining nodes. It maintains each redundant node and duplicates the work of its active partner. In case of a node failure, the redundant node continues without interruption. The application fails only when two corresponding replicas fail. The synchronization protocols and the additional messages incur overhead that is significant in low-level, point-to-point benchmarks. The reported impact on actual applications is for the most part negligible. The overhead for LAMMPS is less than 4%, for SAGE less than 10%, for CTH less than 20% at 2,048 nodes, and for HPCCG less than 5%. RedMPI leverages rMPI technology, such as the PMPI method and the all-to-all message replication, as a follow-on project. RedMPI differentiates itself from rMPI by offering SDC protection and a new low-overhead replication protocol.

The modular-redundant Message Passing Interface (MR-MPI) [9] is a similar solution for transparently executing HPC applications in a redundant fashion. It also utilizes the PMPI to transparently intercept MPI calls from an application and to hide all redundancy-related mechanisms. 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. Partial replication, such as 50%, for tunable resilience is supported. 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 leverages MR-MPI technology, such as its redundant collectives, as a follow-on project. RedMPI extends beyond the capabilities of MR-MPI by protecting against SDC and lowering the replication overhead.

In contrast to rMPI and MR-MPI, VolpexMPI [20] is an MPI library implemented from scratch that offers redundancy internally. It supports around 40 MPI functions 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 the small message latency 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 offers better performance as replication protocols are part of the low-level communication inside the MPI library.

8. CONCLUSION

Redundant computing is one approach to detect SDC. In this paper, we evaluate the feasibility of implementing SDC detection and correction at the MPI layer and report the runtime overhead imposed by this level of protection. We present two consistency protocols and measure their ability to detect injected faults and their impact on application runtimes.

We find that for the second, more efficient, protocol, MsgPlusHash, result in an average overhead ranging between 20% and 60% for triple redundancy and 13% to 45% for dual redundancy depending on the number of messages sent by the application and do not change significantly as the number of processes is varied. These modest overhead ranges indicate the potential of RedMPI to protect against SDC for large-scale runs. Overheads for applications under RedMPI dependent heavily on communication patterns. An application only requiring SDC detection will be sufficient with dual redundancy overhead while the additional correction support requires triple redundancy.

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 experiments, we encountered two events that RedMPI was not able to correct. It is important to note that we have stress-tested RedMPI beyond realistic rates in order to generate extreme SDC rates for presentation of RedMPI's capabilities. Both of the uncorrectable events lead to a deadlock and did not allow the application to proceed with corrupt data but prevented corrupted results from being reported without knowledge of application scientists. Future would can address this by incorporating deadlock detection between sets of replicas, which is feasible for just three replications under triple redundancy. In summary, RedMPI was

successful in preventing invalid data from propagating or being transmitted without detection in even the most extreme scenarios, which could yield invalid application results for an unprotected application.

While the cost of double and triple redundancy is high in terms of power and price, implementing redundancy is not. Detecting and correcting silent data errors may be worth the cost for mission-critical and high-consequence applications such as many large-scale simulations of grand-challenge applications.

9. REFERENCES

[1] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3ghz fifth generation sparc64 microprocessor. In *Design Automation Conference*, pages 702–705, New York, NY, USA, 2003. ACM Press.

[2] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *International Symposium on Microarchitecture*, pages 196–207, 1999.

[3] 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, June 7-12, 2007. ACM Press, New York, NY, USA.

[4] G. Bronevetsky and A. Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical Report TR-JLPC-09-01, Lawrence Livermore National Laboratory, Livermore, CA, USA, Aug. 2009.

[5] J. T. Daly. ADTSC nuclear weapons highlights: Facilitating high-throughput ASC calculations. Technical Report LALP-07-041, Los Alamos National Laboratory, Los Alamos, NM, USA, June 2007.

[6] 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*, pages 19–22, May 2008.

[7] 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.

[8] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Microelectronics Division, 1997.

[9] 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, Feb. 15-17, 2011. ACTA Press, Calgary, AB, Canada.

[10] 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*, pages 189–194, Innsbruck, Austria, Feb. 16-18, 2009. ACTA Press, Calgary, AB, Canada.

[11] J. Fabula, J. Moore, and A. Ware. Understanding neutron single-event phenomena in FPGAs. *Military Embedded Systems*, 3(2), 2007.

[12] K. B. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretty, T. Kordenbrock, and R. Brightwell. Increasing fault resiliency in a message-passing environment. TR SAND2009-6753, Sandia National Lab, Oct. 2009.

[13] A. Golander, S. Weiss, and R. Ronen. DDMR: Dynamic and scalable dual modular redundancy with short validation intervals. *IEEE Computer Architecture Letters*, 7(2):65–68, 2008.

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

[15] 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*, volume 46, pages 494–499, Denver, CO, USA, June 25-29, 2006. Institute of Physics Publishing, Bristol, UK.

[16] T. Heijmen, P. Roche, G. Gasiot, K. R. Forbes, and D. Giot. A comprehensive study on the soft-error rate of flip-flops from 90-nm production libraries. *IEEE Transactions on Device and Materials Reliability (TDMR)*, 7(1):84–96, 2007.

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

[18] L. L. N. Laboratory. Personal communications. 2007.

[19] O. R. N. Laboratory. Personal communications. 2010.

[20] 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*, volume 5759, pages 124–133, Espoo, Finland, Sept. 7-10, 2009. Springer Verlag, Berlin, Germany.

[21] A. Mahmood and E. J. McKluskey. Concurrent error detection using watchdog processors - A survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[22] 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)*, 5(3):329–335, 2005.

[23] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.

[24] 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*, pages 99–110, Anchorage, AK, USA, May 25-29, 2002. IEEE Computer Society.

[25] 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.

[26] 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.

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

[28] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *International Symposium on Microarchitecture*, pages 214–224, 2001.

[29] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *International Symposium on Computer Architecture*, pages 25–36, 2000.

[30] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *International Symposium on Fault Tolerant Computing*, pages 84–91, 1999.

[31] N. Saxena and E. McCluskey. Dependable adaptive computing systems – the roar project. In *Intl. Conf. on Systems, Man, and Cybernetics*, pages 2172–2177, Oct. 1998.

[32] 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*, pages 193–204, 2009.

[33] 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)*, 6(2):135–148, 2009.

[34] D. P. Siemwiorek. Architecture of fault-tolerant computers: An historical perspective. *Proceedings of the IEEE*, 79(12):1710–1734, 1991.

[35] J. R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, 1976.

[36] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *International Symposium on Computer Architecture*, pages 87–98, 2002.

[37] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307, 1996.

[38] Y. C. B. Yeh. Design considerations in boeing 777 fly-by-wire computers. In *IEEE International High-Assurance Systems Engineering Symposium*, page 64, 1998.