DINO: Divergent Node Cloning for Sustained Redundancy in HPC ⋆

Arash Rezaei [a] Frank Mueller [a,*] Paul Hargrove [b] Eric Roman [b]

[a] *North Carolina State University, Raleigh, NC 27695-7534*

[b] *Lawrence Berkeley National Laboratory, Berkeley, CA*

Abstract

Complexity and scale of next generation HPC systems pose significant challenges in fault resilience methods such that contemporary checkpoint/restart (C/R) methods that address fail-stop behavior may be insufficient. Redundant computing has been proposed as an alternative at extreme scale. Triple redundancy has an advantage over C/R in that it can also detect silent data corruption (SDC) and then correct results via voting. However, current redundant computing approaches do not repair failed or corrupted replicas. Consequently, SDCs can no longer be detected after a replica failure since the system has been degraded to dual redundancy without voting capability. Hence, a job may have to be aborted if voting uncovers mismatching results between the remaining two replicas. And while replicas are logically equivalent, they may have divergent runtime states during job execution, which presents a challenge to simply creating new replicas dynamically.

This problem is addressed by, DIvergent NOde cloning (DINO), a redundant execution environment that quickly recovers from hard failures. DINO consists of a novel node cloning service integrated into the MPI runtime system that solves the problem of consolidating divergent states among replicas on-the-fly. With DINO, after degradation to dual redundancy, a good replica can be quickly cloned so that triple redundancy is restored. We present experimental results over 9 NAS Parallel Benchmarks (NPB), Sweep3D and LULESH. Results confirm the applicability of the approach and the correctness of the recovery process and indicate that DINO can recover from failures nearly instantly. The cloning overhead depends on the process image size that needs to be transferred between source and destination of the clone operation and varies between 5.60 to 90.48 seconds. Simulation results with our model show that dual redundancy with DINO recovery always outperforms 2x and surpasses 3x redundancy on up to 1 million nodes. To the best of our knowledge, the design and implementation for repairing failed replicas in redundant MPI computing is unprecedented.

Key words: Fault Tolerance, High Performance Computing, Node Cloning, Redundant Computing
PACS: 07.05.Bx

1 Introduction

Several studies have emphasized the significance of challenges in reliability for next generation supercomputers [2,5,7]. In projections, system reliability drastically decreases at exascale and system mean time to failure (MTTF) would be in the order of few hours without major hardware and software advances. Node failures are commonly due to software or hardware faults. Software faults can be due to bugs (some of which may only materialize at scale), complex software component interactions and race conditions that surface only for rare parallel execution interleavings of tasks [6]. Hardware faults may result from aging, loss of power, and operation beyond temperature thresholds. One resilience method is redundant computing [4,12,11,13]. In redundant computing, multiple components (2 ore more) are allocated to perform the same task. A recent study [19] shows that the skepticism toward redundancy with respect to its cost might not hold anymore. Their results on the cloud platform show that redundancy could be a viable and even cost-effective approach for HPC. They combine checkpointing with redundancy in variable-cost spot market allocations on Amazon EC2. They achieve up to 7 times cheaper execution compared to the on-demand default market. Redundancy provides tolerance not only against hard faults but also soft faults, such as silent data corruptions (SDCs), which do not stop application execution as they are undetectable. SDCs may manifest at application completion by producing wrong results or, prior to that, wrong interim results. A study at CERN raised concerns over the significance of SDC in memory, disk and RAID [22]. Their results indicate that SDC rates are orders of magnitude larger than manufacture specifications. Schroeder et al.'s study [27] of the DRAM errors on a large scale over the course of 2.5 years concludes that more than 8% of DIMMs are affected by errors per year. A study by Microsoft over 1 million consumer PCs also confirms that CPU faults are frequent [21].

State-of-the-art approaches for redundant computing do not provide a sustained redundancy level during job execution when processes experience failures. After a replica process fails, either the application deadlocks (RedMPI [13]) or other replicas ensure that the application can progress in execution [4]. Note that after a replica

⋆ An earlier version of this paper appeared at Cluster'15 [24]. This journal version extends the earlier paper by proposing a new model for job execution time under redundancy and an extensive experimental evaluation. This work was supported in part by grants from Lawrence Berkeley National Laboratory and NSF grants 1058779 and 0958311. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Computer Science program under contract number DE-AC02-05CH11231. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

* Corresponding author.
Email address: mueller@cs.ncsu.edu (Frank Mueller).

failure, even if the job can continue its execution, the SDC detection module cannot guarantee application correctness (e.g., an undetected SDC might occur). Checkpoint/Restart (CR) is another popular method for tolerating hard errors, but cannot handle soft errors. In CR, in every checkpoint interval, a snapshot of all processes is created and saved to storage. The snapshot could be a light-weight application-level approach or at the process-level. If a hard error occurs, all processes re-load the last snapshot into memory and the application continues execution.

Using the same simulation parameters as Ferreira et al. [12], we plotted in Fig. 1 the elapsed times (left x-axis) of three jobs of 24/168/720 hours over different redundancy levels (y-axis) ranging from no redundancy (0%) over half of the nodes replicated (50%) to dual redundancy for all nodes (100%). The results (dashed lines) show that for a 720-hour job applications runtime without replication is more than six times higher than under dual redundancy. Due to this, job capacity (2nd x-axis, dashed lines) increases up to a factor of 4.5 under dual redundancy, i.e., 4.5 times more dual redundant jobs of the same size can finish execution using the same resources in the time it would have taken to finish a non-redundant job due to checkpoint and (mostly) restart overheads. This shows that replication has the potential to outperform CR around the exascale range.

Elliott et al. [10] showed in a more refined model that CR will eventually take longer than redundancy due to recomputation, restart and I/O cost. At scale, this makes capacity computing (maximizing the throughput of smaller jobs) more efficient than capability computing (using all nodes of an exascale machine). E.g., at 80,000 CPU sockets, dual redundancy will finish twice the number of jobs that can

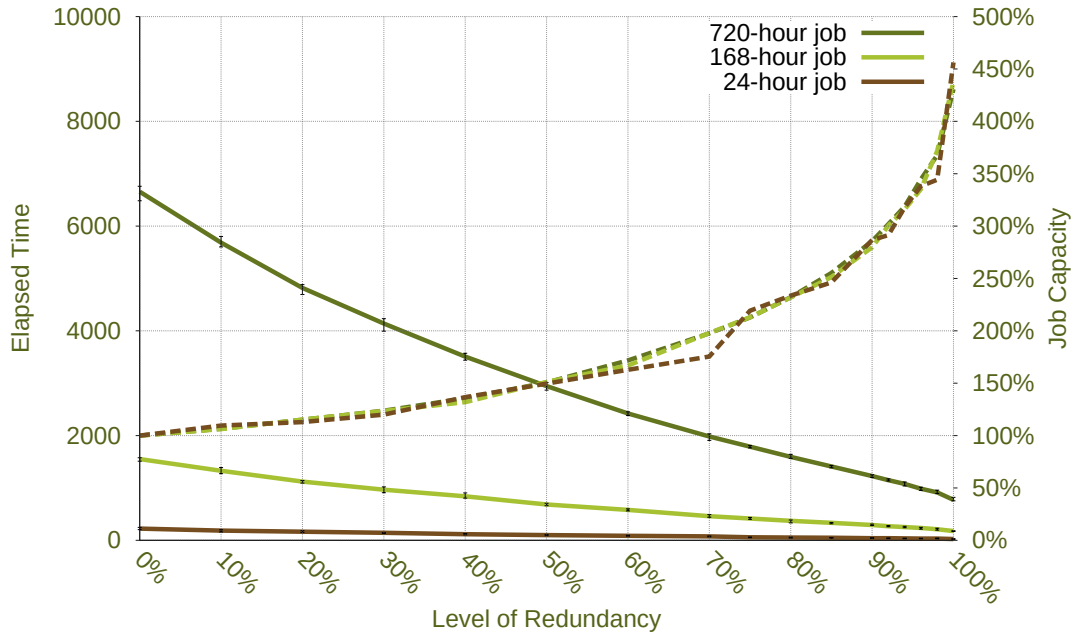


Fig. 1. Modeled application runtime and job capacity per redundancy level for 100,000 nodes

be handled without redundancy. This includes a redundancy overhead of 0-30% longer time (due to additional messages) with hashing protocols [13], which has no impact on bandwidth for Dragonfly networks since original and replica sphere exchange full messages independently. As hash messages are small, they add latency but do not impact bandwidth. Since twice the jobs finished under dual redundancy, this amounts to the *same* energy.

The objective of this work it to address process failures in parallel applications that follow tightly-coupled programming model. Such applications are executed on HPC platforms and use MPI-style communication [15]. Term *rank* is used to refer to an MPI task/process. Each MPI process is associated with a unique integer value identifying the rank. Assume that a job with n ranks requires t hours to complete in the absence of failures (plain execution time). A system with r levels of redundancy consist of $r \times n$ ranks, where n logical MPI tasks are seen by the user. Redundant replicas remain transparent and are launched automatically by the runtime. Replicas of the same task are identical in terms of functionality and they perform the same operation at the application level. We assume that a small pool of spare nodes exists in the cluster environment. Initially, these nodes are not executing any job but remain in powered state to be ready for recovery cases. Our work focuses on the recovery phase and assumes that orthogonal solutions in failure detection [34,18] exist, i.e., we assume that a fault detector is provided by the system.

We introduce node cloning as a means to sustain a given redundancy level. (We use the terms node / MPI task cloning synonymously.) The core idea is to recover from hard errors with the assistance of healthy replicas. A healthy replica is cloned onto a spare node to take over the role of the failed process in "mid-flight". To address shortcomings in current redundant systems, we provide the following contributions:

- We devise a generic high performance node cloning service under divergent node execution (DINO) for recovery. DINO clones a process onto a spare node in a live fashion. We integrate DINO into the MPI runtime under redundancy as a reactive method that is triggered by the MPI runtime to forward recover from hard errors, e.g., node crash or hardware failure.
- We propose a novel Quiesce algorithm to overcome divergence in execution without excessive message logging. Execution of replicas is not in a lock-step fashion, i.e., can diverge. Our approach establishes consistency through a novel, scalable multicast variant of the traditional (non-scalable) bookmark protocol [26] and resolves inconsistencies through exploiting the symmetry property of redundant computing.
- We evaluate DINO's performance for MPI benchmarks. The time to regain dual redundancy after a hard error varies from 5.60 seconds to 90.48 seconds depending on process image size and cross-node transfer bandwidth, which is short enough to make our approach practical.
- We provide a model with low error (at most 6%) for estimating job execution time under redundancy and validate our model on the Stampede supercom-

puter [30]. We extrapolate results under this model to extreme scale with a node MTTF of 50 years [14] and show that dual redundancy+cloning outperforms triple redundancy within the exascale node count range, yet at 33% lower power requirement.
- We show that 25 spare nodes suffice for a 256K node system (when nodes can be repaired) independent of communication overhead of the applications.

The paper is structured as follows: Challenges are discussed in Section 2. Section 3 presents a brief background on RedMPI. Section 4 introduces the design of DINO. Our Quiesce algorithm is presented in Section 5. Section 6 details the implementation of DINO. Analysis of job completion times are presented in Section 7. The experimental evaluation is provided in Section 8. Simulation results are presented in Section 9. Section 10 compares our work to related work. Section 11 summarizes the paper.

2 Challenges

High performance node cloning service. With increasing memory capacity and the applications' tendency to load larger data into memory, a cloning service that limits the interference with the process execution is more preferable. A generic service that clones a process (including the runtime/kernel state, memory content, and CPU state) onto a spare node is the first challenge. Off-the-shelf checkpointing libraries like BLCR [9] or MTCP [25] provide the needed functionality. However, the downtime to take the snapshot, save it to storage and retrieve it from storage on the other node followed by restoring the snapshot is quite high. We compare "CR-based" process cloning with our "live" process cloning in Section 8.

Communication consistency and divergent states. The execution of replica ranks does not occur in a lock-step fashion, i.e., they tend to diverge not within computational regions but also by advancing or falling behind relative to one another in terms of communication events. This provides a performance boost to redundant computing as it allows the replicas to execute freely to some extent. However, this divergence introduces complications for failure recovery. Lack of a proper approach that guarantees the consistency, results in deadlock. This would generally require message logging over large periods of time. Instead, we devised a novel algorithm to establish communication consistency that tolerates divergence due to asymmetric message progression and region-constrained divergence in execution.

Extending a job to a spare node. The MPI runtime leverages helper daemons on the compute nodes to facilitate job execution. The helper daemons launch the MPI ranks, monitor their liveness and play a role in application termination. The spare nodes are not part of a specific job, and consequently, no daemon is running on them. A spare node might be assigned to any job that has a failed replica as a

part of the recovery phase in DINO. Thus, extending the job includes modifying the internal data structures like routing information, task map and daemon map information besides spawning a new daemon.

Integration into the runtime system. Cloning a process without carefully resuming its communication state results in inconsistency and job failure. After the process is cloned onto a spare node, it should exchange its communication endpoint information with the rest of the processes in order to make normal application progress.

3 Background (RedMPI)

We use RedMPI [13] for redundant and transparent MPI execution. RedMPI detects SDCs through analyzing the content of messages. Data corruption may occur in CPU, memory or cache, either due to multi-bit flits under ECC or even single-bit flips without ECC. Such corruption either does not affect the output or eventually materializes in the transmitted message (or in disk/std I/O, which is addressed elsewhere [3]).

It supports linear collectives where all the collective operations are mapped to point-to-point communication. These collectives do not provide high performance, especially not at large scale. Another mode, named *optimized*, directly calls the collective module of the MPI library and provides native performance (parallelized collectives). RedMPI benefits from the interpositioning layer of MPI known as the MPI profiling layer. [1] The current implementation of RedMPI is not capable of detecting hard errors. As a result, an MPI process failure leads to a deadlock where the processes wait indefinitely for progress in communication with the failed process (or until the time-out from the communication layer terminates the application with an error).

In the following, we describe how basic MPI functions are implemented in RedMPI under dual redundancy. `MPI_Send` and `MPI_Recv` are blocking calls, and `MPI_Isend` and `MPI_Irecv` are non-blocking. `MPI_Send` is implemented with two non-blocking send calls to rank x and its corresponding replica x' followed by a call to `MPI_Waitall` (see Table 1). `PMPI_Isend` provides the actual send functionality. Similarly, `MPI_Recv` is implemented with two `PMPI_Irecv` calls, then a `MPI_Waitall` call. RedMPI extends the `MPI_Request` data structure to contain the extra requests that it creates. For the sake of brevity, this is omitted. `MPI_Isend` and `MPI_Irecv` are similar to `MPI_Send` and `MPI_Recv`

[1] PMPI is the MPI standard profiling interface. Each MPI function can be called with a MPI_ or PMPI_ prefix. This feature of MPI allows one to interpose functions with MPI_ prefix while using functions with PMPI_ prefix to implement the required functionality.

but there is no MPI_Waitall call. The MPI_Wait is implemented with a call to the actual function (PMPI_Wait), it then verifies the integrity of the messages in case of a receive request. The implementation of MPI_Waitall performs a PMPI_Waitall over all requests followed by the integrity check for received messages.

Table 1
Implementation of 6 basic MPI functions in RedMPI

MPI_Send(x,msg){
PMPI_Isend(x,msg);
PMPI_Isend(x',msg);
MPI_Waitall();
}
	MPI_Recv(x,msg){
	MSG msg';
	PMPI_Irecv(x,msg);
	PMPI_Irecv(x',msg');
	MPI_Waitall();
	}

MPI_Isend(x,msg){
PMPI_Isend(x,msg);
PMPI_Isend(x',msg);
}
	MPI_Irecv(x,msg){
	MSG msg';
	PMPI_Irecv(x,msg);
	PMPI_Irecv(x',msg');
	}

MPI_Wait(req){
PMPI_Wait(req);
if(req.type=="Recv")
verify_integrity(req);
}
	MPI_Waitall(req){
	PMPI_Waitall(req);
	if(req.type=="Recv")
	verify_integrity(req);
	}

There is no lock-step execution among the replicas and they only communicate directly to resolve certain calls to avoid non-determinism. Wildcard values in source or tag (MPI_ANY_TAG, MPI_ANY_SOURCE) and MPI_Wtime are examples of the latter case (not shown here), which are supported (see [13]). In redundant computing, send (or receive) calls are always posted in pairs. In other words, the number of posted send (or receive) requests to (or from) any two replicas are always equal. We call this the *symmetry* property and exploit it in the Quiesce algorithm (Section 5).

4 Design of DINO

DINO has a generic process cloning service at its core. Node cloning creates a copy of a given running process onto a *spare* node. The cloning mechanism itself is MPI agnostic and is applied to processes encapsulating MPI tasks in this work. DINO considers the effect of cloning on the MPI runtime system, as detailed later. Fig. 2 shows how the system retains dual redundancy in case of a failure. A and A' are logically equivalent and both perform the same computation. They run on nodes 0 and 1, respectively, and comprise *sphere 1* of redundant nodes. Ranks B, B' on nodes $2, 3$ are also replicas and shape sphere 2. If node 2 (B) fails, its replica (B') on node 3 (*source* node) is cloned onto node 4 (a *spare* node) on-the-fly. The newly created rank B'' takes over the role of failed rank B and the application recovers from the loss of redundancy. At the end of node cloning, B' and B'' are in the same state from the viewpoint of the application, but not necessarily from another rank's point of view due to stale B references. The Quiesce algorithm resolves such inconsistencies.

The process B'' is created on node 4 as follows. While B' performs its normal execution, its memory is "live copied" page by page to B''. This happens in an iterative manner (detailed in Section 6). When we reach a state where few changes in dirty pages (detailed in the implementation) remain to be sent, the communication channels are drained, i.e., any buffered messages are removed. This is necessary to keep the system of all communication processes in a consistent state. After this, the exe-

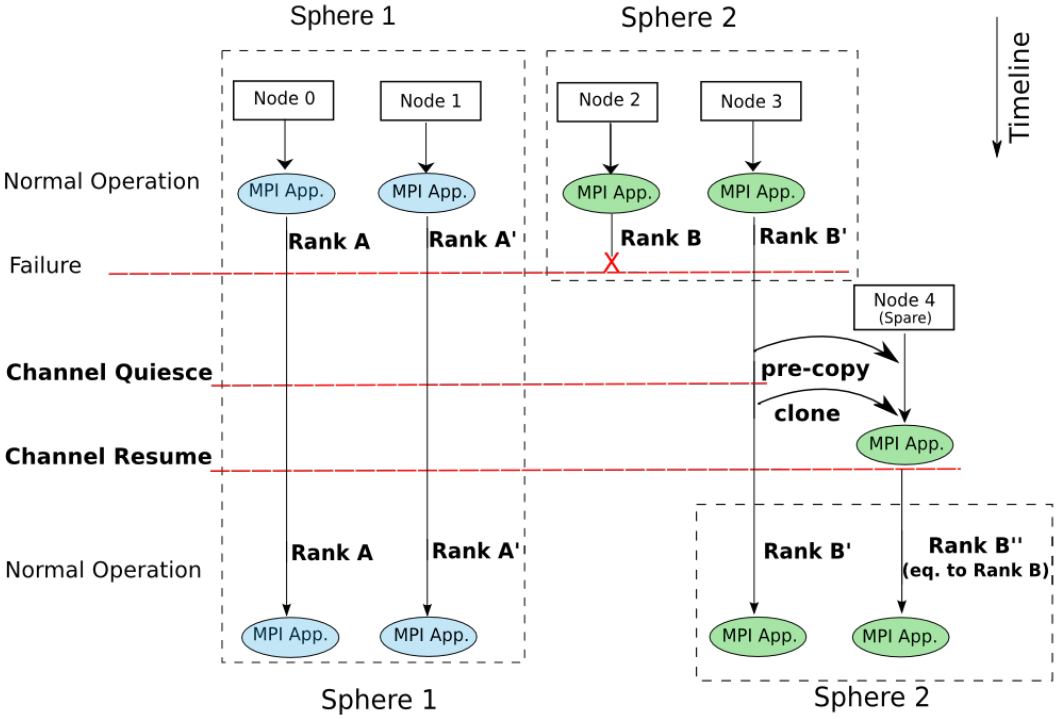


Fig. 2. Node cloning: Application w/ 2 ranks under dual redundancy

cution of rank B' is briefly paused so that the last dirty pages, linkage information, and credentials are sent to node 4. Rank B'' receives and restores this information and then is ready to take over the role of failed rank B. Then, communication channels are resumed and execution continues normally. Between channel draining and channel resumption, no communication may proceed. This is also necessary for system consistency with respect to message passing.

The time interval between error detection and the end of DINO recovery is a "vulnerability window" where undetected SDCs may occur. The vulnerability window depends on the process image size and is evaluated experimentally in Section 8 and projected for large scale in Section 9.

5 Quiesce Algorithm

The purpose of the Quiesce algorithm is to resolve the communication inconsistencies inside DINO at the library level and provide transparent and consistent recovery to the application layer. The inconsistencies are rooted in the state divergence of replicas. In Section 3, we described basic MPI functions and their implementation inside RedMPI. Blocking operations impose limited divergence. But non-blocking operations can easily create scenarios where the state of replicas differs largely as there is no enforced state synchronization among replicas. Only application *barriers* are true synchronization points and RedMPI does not introduce additional barriers. However, the application-specific inter process dependencies caused by *wait* operations limit the divergence among replicas. Thus, divergence of replicas is application-dependent. If an application only uses Send/Recv (blocking calls), the divergence is bounded by 3 MPI calls (see Fig. 3-A). But if it has "n" Isend/Irecv calls followed by a Waitall, then the divergence bound is $2n + 1$ (see Fig. 3-B). Note that the bound is an indirect result of communication dependencies.

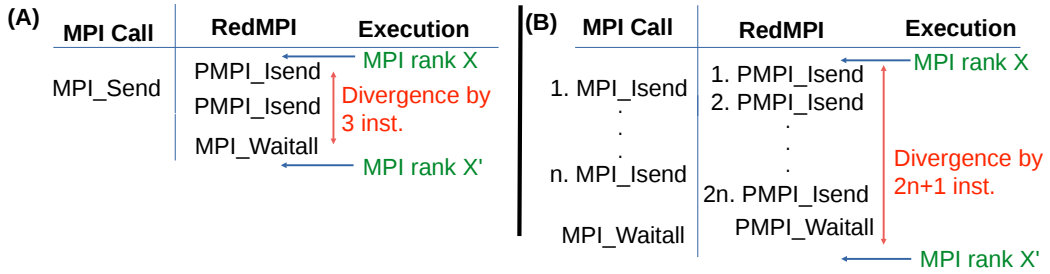


Fig. 3. Divergence of replicas: (A) one MPI_Send (B) "n" MPI_Isends

Algorithm 1 shows the steps for Quiesce. Let us assume that rank B has failed and rank B' is cloned to create B'', which takes over the work of B. Ranks B and B'' are in the same state, but any other ranks may still assume B'' to have the state of B.

Algorithm 1 Quiesce Algorithm

```
1: /* 1. Exchange communication state with B' */
2: if rank == B' then
3:    bookmarks *array;
4:    for (i = 0; i < nprocs; i++) do
5:       if i ≠ B' & i ≠ B then
6:          /*send bookmark status, then receive into
7:          appropriate location in bookmarks array */
8:          send_bookmarks(i);
9:          recv_bookmarks(i, array[i]);
10:      end if
11:   end for
12: else
13:   bookmark bkmrk;
14:   /*Receive remote bookmark into bkmrk then send*/
15:   recv_bookmarks(B', bkmrk);
16:   send_bookmarks(B');
17: end if
18: /* 2. Calculate in-flight msg(s) and drain them */
19:   Cal_and_Drain();
20: if rank ≠ B' then
21:   /* 3. Resolve possible Send inconsistencies */
22:   if Sent[B] == Sent[B'] then
23:      noop;
24:   else if Sent[B] < Sent[B'] then
25:      Skip the diff to B'';
26:   else if Sent[B] > Sent[B'] then
27:      Repeat the diff to B'';
28:   end if
29:   /* 4. Resolve possible Recv inconsistencies */
30:   if Received[B] == Received[B'] then
31:      noop;
32:   else if Received[B] < Received[B'] then
33:      Skip the diff from B'';
34:   else if Received[B] > Received[B'] then
35:      Repeat the diff from B'';
36:   end if
37: end if
```

Stage 1 and 2 of the Algorithm 1 clear the outgoing channels of B' and of any other ranks that have initiated a send to B'. We modified the bookmark exchange protocol [26] to equalize these differences. The original bookmark protocol creates a consistent global snapshot of the MPI job, which requires an all-to-all communication and is not scalable due to its high overhead. In our modified bookmark protocol, each process informs B' of the number of messages it has sent to B' and receives how many messages are sent by B'. Then, the following question can be answered: Have I received all the messages that B' put on the wire? If not,

some messages (buffered or in transit) remain in the MPI communication channel and should be drained. Rank B' performs a similar task to drain all messages that other ranks put on the wire to reach B'. In stage 2, receive requests are posted to drain the outstanding messages identified in stage 1. They are saved in temporary buffers. When the application execution later (during normal execution) reaches a Recv call, these drain lists are first consulted to service the request. At the end of the drain phase, no more outstanding send requests to/from B' exist in the system.

Due to the symmetry property of redundant computing, every rank receives the same number of messages from members of a given sphere (e.g., B and B'). The same rule applies to the number of messages sent to a given sphere. This property is the basis for resolving the inconsistencies in stages 3 and 4. The goal of these two stages is to prepare all ranks for communicating with B'' by resolving any state inconsistency between their view of B and B''. Every rank keeps a vector of the number of messages that are sent to other ranks ($Sent[]$) and received from other ranks ($Received[]$) along with the message signatures. The sent/received messages and their signatures are buffered inside the runtime of the off-the-shelf MPI library until their receipt can be inferred and they are matched by the same signature on other side of the communication (to also handle out-of-order messages). We use this matching information along with the counters to drain in-flight messages by removing them from the internal buffer. In-flight messages are those that are on the wire but have not yet been fully transmitted during the quiesce phase.

In stage 3, each rank X (other than B') resolves its possible communication inconsistency due to sends to B. Three cases are distinguished: (1) Bookmarks match ($Sent[B] == Sent[B']$): Then B, B', and B'' are in the same state from the point of view of X, and no action is needed. (2) B lagged behind B' ($Sent[B] < Sent[B']$): Then sends from X to B are in transit/will be issued (Fig. 4 part 3.2). Since B has been removed and B'' is ahead (has already seen these messages), they are silently suppressed (skipped). (3) B was ahead of B' ($Sent[B] > Sent[B']$): Then there exist messages in transit/to be sent from X to B' (Fig. 4 part 3.3). Since B'' is in the same state as B', these messages need to be sent to B'' as well.

In stage 4, the same procedure is performed on the Received counters. The 3 cases are symmetric to the prior stage: (1) Bookmarks match ($Received[B] == Received[B']$): Then B, B', and B'' are in the same state from the point of view of X, and no further action is needed. (2) B' was ahead of B ($Received[B] < Received[B']$): Then X is expecting messages from B (Fig. 4 part 4.2). Since B does not exist anymore and B'' will not send them (as it is ahead), these receives silently complete (skipped). Instead, X will provide the corresponding message from B' to the user level. (3) B' lagged behind B ($Received[B] > Received[B']$): Then X is expecting messages from B' (Fig. 4 part 4.3). Since B'' and B' are in the same state, both will send those messages, even though X has already received a copy from B. Thus, messages from B'' are silently absorbed (up to the equalization point).

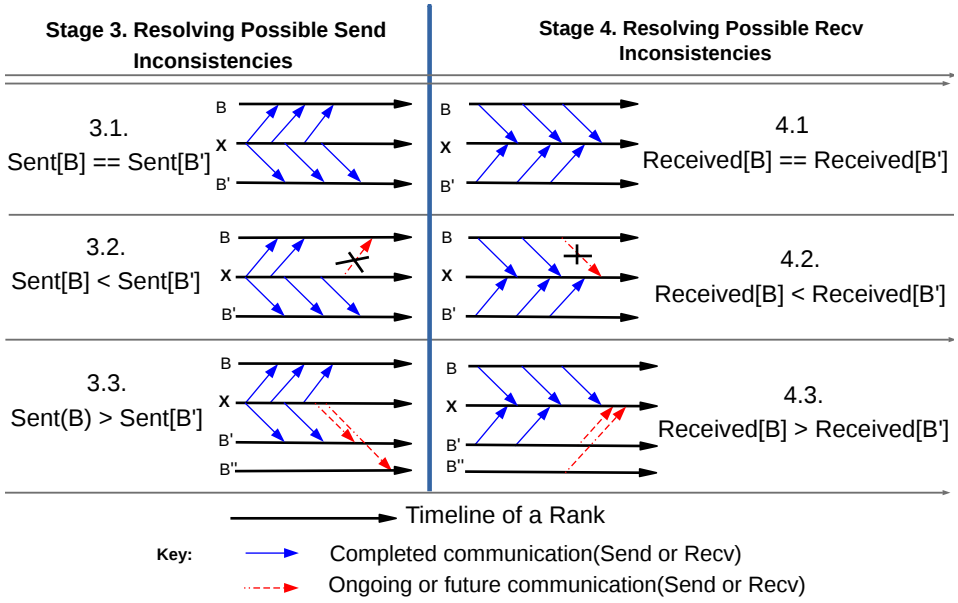


Fig. 4. A view of Steps 3 and 4 of the Quiesce algorithm

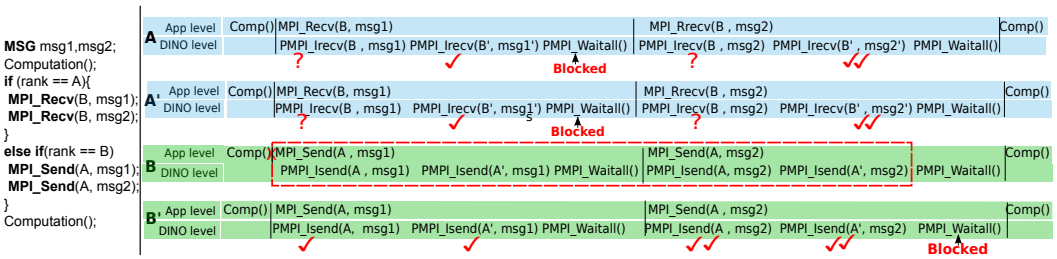


Fig. 5. Program fragment with ranks A and B (left); its application-/DINO-level execution for dual redundancy under 1 failure (right)

Fig. 5 (left side) depicts a program fragment with two ranks, A and B. Two computation sections are separated by two message exchanges from rank B to A (for a predefined type MSG). A failure scenario and our recovery approach with dual redundancy are as follows. Four ranks are created by the MPI runtime, namely A, A', B, B'. `MPI_Recv` and `MPI_Send` calls are redirected to the RedMPI library and implemented as described in Section 3. Fig. 5 (right side) describes the execution per rank at the levels of the application and DINO. Suppose rank B fails at mark "X" in Fig. 5. Then ranks A and A' receive the first message from B' and are blocked to receive a message from B. Rank B', after sending all of its messages, continues execution and reaches the waitall. At this point, only communications with single check-marks have finished, and all processes are blocked so that recovery is required. During recovery, B' is cloned to a spare node to create B'' (equivalent to B). Process B'' starts executing the application from where B' was blocked (the *waitall*). Therefore, neither B nor B'' ever execute the `PMPI_Isend` calls shown in the dotted area. Consequently, ranks A and A' do not receive these messages from B (or B''). In stage 1 of the Quiesce algorithm, all outstanding sends to B' are drained. Receive requests are posted by A and A', and these messages are

logged in temporary buffers (indicated by double check-marks in Fig. 5). Receives from B, highlighted by question marks, remain a unanswered. Quiesce then cancels the first receive (from B) in both A and A' and skips the next receive from B. Subsequently, ranks have consistent states and rank B'' may join the communication of the entire job.

This algorithm does not support wild-cards and assumes collective operations are implemented over point-to-point messages. Other implementations of collectives require a more advanced coordination among ranks during Quiesce. This includes distinguishing the missing messages due to failure along with the topology to correctly determine the destination of messages and to issue cancel/skip operations. Quiesce assumes that the applications issue `MPI_{Wait/Waitall}` calls at least once after a sequence of n non-blocking operations.

6 Implementation

Architecture with Open MPI. Fig. 6 shows the system architecture where novel DINO components are depicted with shaded boxes. RedMPI provides a transparent interpositioning layer for MPI calls between the application and the MPI runtime system (Open MPI). Open MPI has 3 layers: the Open MPI (OMPI) layer, the Open Run-Time Environment (ORTE) and the Open Portability Access Layer (OPAL). OMPI provides the top-level MPI API, ORTE is the interface to the runtime system, and OPAL provides a utility layer and interfaces to the operating system. The `mpirun` process interacts with the cloning APIs to launch tools on source/spare nodes. The node cloning service provides generic process-level cloning functionality via our extensions to BLCR [9]. This service includes three new tools and functionalities named `restore`, `pre-copy` and `clone`. DINO recovery starts with `mpirun` initiating the `restore` tool on the spare node (Step 1). Then `mpirun` runs `pre-copy` on the source node where the healthy replica exists (Step 2). A communication channel between these two processes is created to transfer the process image. Then `mpirun` invokes the daemons on every node for the Quiesce phase. The daemons send a Quiesce signal to the corresponding ranks, and all ranks enter the Quiesce phase (Step 3). This phase includes the Quiesce algorithm that is described in the previous section and ends by pausing the communication channels. The `mpirun` process runs the `clone` tool on the source node to copy the last portion of process information (Step 4). As the last step (Step 5), `mpirun` communicates with the daemons again to update internal data structures of all processes and resumes the communication among ranks.

We next discuss the steps taken during DINO recovery:

1. Launch restore tool. The procedure starts with executing `restore` on the spare node. It listens on a predefined port for incoming information and memory

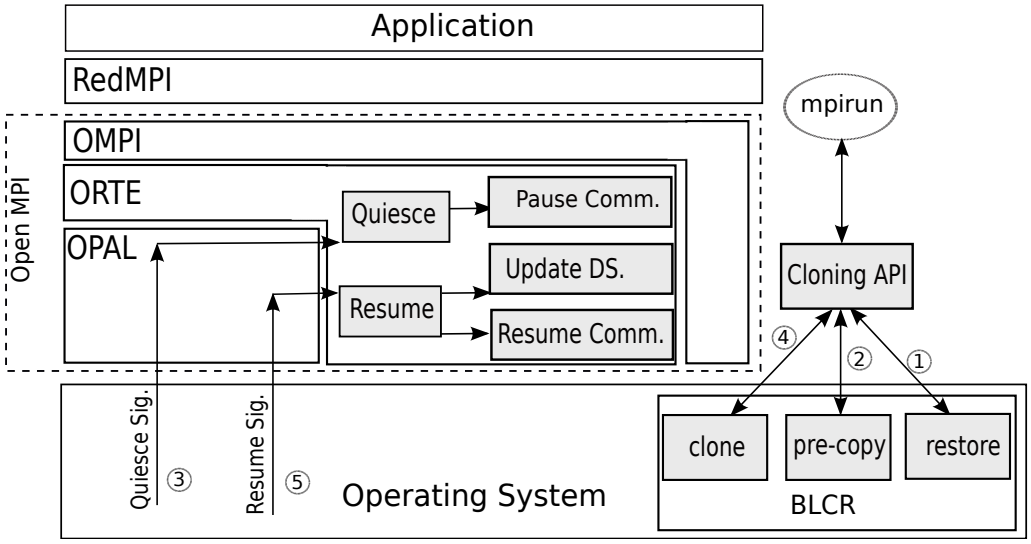


Fig. 6. DINO Architecture

pages later sent by the `pre-copy` and `clone` tools and builds the process state, both at kernel and application level, on the spare node.

2. Pre-copy. This phase transfers a snapshot of the memory pages in the process address space communicated to the spare node while normal execution of the process continues on the source node. We use TCP sockets to create a communication channel between *source* and *spare* nodes. The pre-copy approach borrows concepts from [33] (under Linux 2.4), but adapted to Linux to 2.6 (see related work for a comparison). Vital meta data, including the number of threads, is transferred. [2] The spare node receives the memory map from the pre-copy thread. All non-zero pages are transferred and respective page dirty bits are cleared in the first iteration. In subsequent iterations, only dirty pages are transferred after consulting the dirty bit. We apply a patch to the Linux kernel to shadow the dirty bit inside page table entry (PTE) and keep track of the transferred memory pages. The pre-copy phase terminates when the number of transferred pages reaches a threshold (1MB in our current setting).

3. Channel Quiesce. The purpose of this phase is to create a settle point with the shadow process. This includes draining all in-flight MPI messages. The runtime system also needs to stop posting new send/receive requests. We build this phase on top of the functionality for message draining provided by the CR module of Open MPI [16]. The equalization stage described in Section 5 is implemented in this step.

4. Clone. This phase stops the process for a short time to transfer a consistent image

[2] We assume that applications maintain a constant sized thread pool after initialization, e.g., OpenMP implementations. Cloning applies to the execution phase after such thread pool creation.

of its recent changes to the `restore` tool. The memory map and updated memory pages are transferred and stored at the corresponding location in the address space in B''. Then, credentials are transferred and permissions are set. Restoration of CPU-specific registers is performed in the next phase. The signal stack is sent next and the sets of blocked and pending signals are installed. Inside the kernel, we use a barrier at this point to ensure that all threads have received their register values before any file recovery commences. In short, different pieces of information are transferred to fully create the state of the process.

5. Resume. In this phase, processes re-establish their communication channels with the recovered sphere. All processes receive updated job mapping information, reinitialize their Infiniband driver and publish their endpoint information.

MPI System Runtime. The off-the-shelf Open MPI runtime does not allow to dynamically add nodes (e.g., patch in spare nodes to a running MPI job) and, subsequently, to add daemons to a given job. We implemented this missing functionality, including manipulation of job data structures, creation of a daemon, redirection of I/O and exchange of contact information with the `mpirun` process. Finally, there are communication calls issued by RedMPI that violate the symmetry property. These control messages are required to ensure correct MPI semantics under redundancy, e.g., for MPI calls like `MPI_Comm_split` and wildcard receives. We distinguish them and only consider application-initiated calls when identifying messages to *skip* and *repeat* in Algorithm 1.

OS Runtime and constraints. Many modern Linux distributions support *prelinking*, which enables applications with large numbers of shared libraries to load faster. Prelinking is a method of assigning fixed addresses to and wrapping shared libraries around executables at load time. However, these addresses of shared libraries differ across nodes due to randomization. We assume prelinking to be disabled on compute nodes to facilitate cloning onto spare nodes. Files opened in write mode on a globally shared directory (e.g., NFS) can cause problems (due to access by both replicas), a problem considered in orthogonal work [3].

7 Job Completion Time Analysis

The objective of this section is to provide a qualitative job completion analysis for redundant computing to assess the effect of our resilience technique. We make the following assumptions in the mathematical analysis. (1) Node failures follow a Poisson process. Subsequently, the time between two failures follows an exponential distribution. (2) Failures occur independently. A failure does not affect or increase the failure probability of other ranks. (4) Failures do not strike the two nodes involved in a cloning operation (the source and the spare node) while DINO recovery is in progress. The short time required for cloning (seconds) relative to

time between failures (hours) allows us to make this assumption. Let us denote:
• t: failure-free execution time of the application (without redundancy)
• r: level of Redundancy
• α: fraction of total application time spent on communication (without redundancy)
• β: fraction of total application time spent on serial communication (with redundancy)
• θ: MTTF of a node ($\lambda = 1/\theta$: failure rate of a node)
• t_{clone}: time spent on a DINO recovery

Estimating redundancy overhead requires measuring the application's communication overhead. We use mpiP [20] to collect such information. The mpiP tool computes "AppTime" and "MPITime". "AppTime" is the wall-clock time from the end of `MPI_Init` until the beginning of `MPI_Finalize`. "MPITime" is the wall-clock time for all MPI calls within "AppTime". It also reports the aggregate time spent on every MPI call. "SendTime", "IsendTime", "RecvTime", "IrecvTime" and "WaitTime" are the aggregate times spent on `MPI_Send`, `MPI_Isend`, `MPI_Recv`, `MPI_Irecv` and `MPI_{Wait,Waitall}`, respectively.

We use mpiP with the 1x (no redundancy) execution of the application, collect the information on communication overhead and then try to estimate the execution time with rx ("r" level of redundancy). We define α as the fraction of time spent on communication under 1x. The α ratio is application specific and also depends on the input data size. α is defined as: $\alpha = MPITime/AppTime$.

Next we consider a redundant computing environment and reason about different types of communication. Point-to-point MPI calls are always processed serially as r messages are required to be transmitted. E.g., `MPI_Send` takes 2 times longer under 2x. However, this depends on the message size and the transfer protocol deployed dynamically by the MPI runtime (r times overhead may not be the best estimate under some scenarios). Collective operations, in contrast, are performed in parallel as we use RedMPI in its `optimized` collective mode. Under this mode, collectives are called directly from RedMPI and are internally parallelized. As a result, only the point-to-point (P2P) segment of the application is performed serially. We define F_{Send} and F_{Recv} fractions as follows:

$$F_{Send} = \frac{SendTime + IsendTime + C_1 \cdot WaitTime}{MPITime} \tag{1}$$

$$F_{Recv} = \frac{RecvTime + IrecvTime + C_2 \cdot WaitTime}{MPITime} \tag{2}$$

Without non-blocking receives, we fold "WaitTime" into the F_{Send} fraction ($C_1 = 1, C_2 = 0$). Similarly, without non-blocking sends, we fold wait time in F_{Recv} fraction ($C_1 = 0, C_2 = 1$). With both `MPI_Isend` and `MPI_Irecv`, we divide the wait time equally between the F_{Send} and F_{Recv} fractions. ($C_1 = 0.5, C_2 = 0.5$). Finally, without non-blocking sends and receives, we assign $C_1 = 0, C_2 = 0$.

We define β as portion of the communication that is performed serially with redundancy: β is estimating the message transfer overhead. The F_{Send} and F_{Recv} fractions have an inherent wait (e.g., until the other side's execution reaches the communication point). A good estimation is the minimum of the two values as it captures their overlap:

$$\beta = min(F_{Send}, F_{Recv}) \cdot \alpha \tag{3}$$

Under redundancy, the serial segment of the communication ($\beta \cdot t$) is multiplied by the redundancy factor (r) while the rest due to computation and parallel communication ($(1 - \beta) \cdot t$) remains the same. In other words, the extra point-to-point communication due to redundancy is performed serially, but the replicas are performing their computation and collective segments in parallel. As a result, the execution time with redundancy is estimated as:

$$T_{red} = \beta \cdot t \cdot r + (1 - \beta)t \tag{4}$$

Table 2 shows the model validation experiment performed on the Stampede supercomputer. Stampede nodes are Dell C8220z running CentOS 6.3 with the 2.6.32 Linux kernel. Each node contains two Xeon Intel 8-Core 2.7GHz E5-processors (16 total cores) with 32GB of memory (2GB/core). We choose 5 benchmarks each in 4 configurations: 3 of NPB programs (CG, LU, MG), Sweep3D (S3D), and LULESH. Sweep3D represents an ASC application that solves a neutron transport problem. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements [17]. We used 1024 and 2048 ranks under dual redundancy for CG, LU, MG and Sweep3D. As LULESH requires cubic values for the number of ranks, we used 1000 and 1728 ranks. We are interested in studying the overhead of redundancy in large number of ranks, so we launch one rank per core (16 ranks per node). Each benchmark is tested with two input sizes, Classes D and E, for NAS benchmarks. Two configurations of 100×40×400 and 320×40×400 are used for Sweep3D and size 25 and 27 are used for LULESH. Table 2 depicts execution times without redundancy (1x), α, *Send*, and *Recv* fractions from mpiP profiling. Eq. 3 is used to compute the β column. Experimental execution time under redundancy (2x Real), and modeled execution time under redundancy using Eq. 4 (2x Model) are the next two columns.

CG, LU and MG do not have any `MPI_Isend` call, so we assign $C_1 = 0, C_2 = 1$ in Equations 1 and 2. Sweep3D has no non-blocking calls ($C_1 = 0, C_2 = 0$) and LULESH has both non-blocking send and receive calls ($C_1 = 0.5, C_2 = 0.5$). The benchmarks cover 3 of 4 possible cases. The fourth case ($C_1 = 1, C_2 = 0$) is rather uncommon due to penalties associated with it.

Fig. 7 compares the real overhead and modeled overhead for the experiments in Table 2. The overhead percentage of redundancy is derived from the experiment via $\frac{2xReal-1x}{1x}$ (Real Overhead) and by modeling using $\frac{2xModeled-1x}{1x}$ (Modeled Overhead). The x axis shows 4 configurations of each benchmark numbered from 1 to

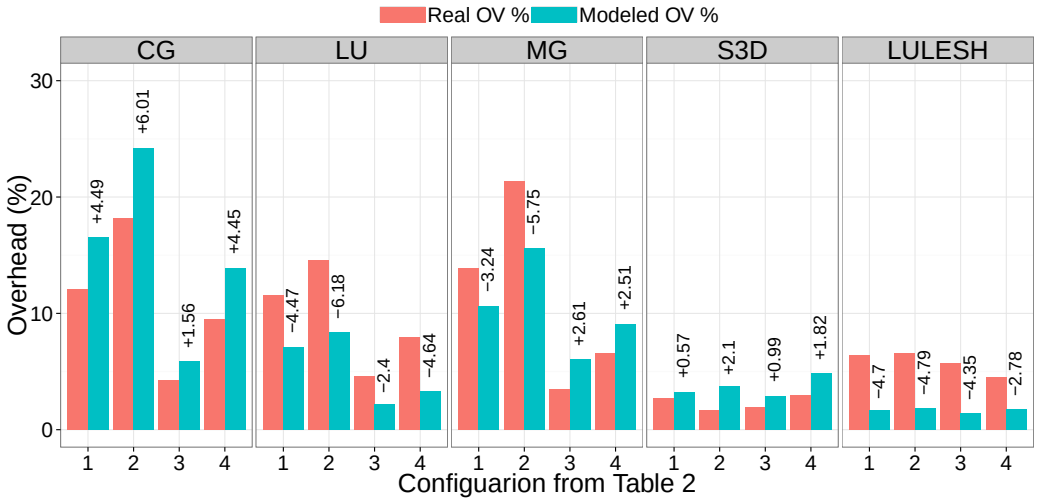


Fig. 7. Experimental vs. modeled overhead for redundancy — The model error is depicted on top of the each bar of modeled overhead

4 in the same order shown in Table 2. The model error is also shown on the top of the modeled overhead. The results show overestimation in CG between 1.56% and 6.01%. Our model underestimates the LU benchmark in all 4 cases between 2.4% to 6.18%. Configurations 1 and 2 of MG are underestimated (3.24% and 5.75% error) while the other two (3, 4) are overestimated (2.61% and 2.51% error). Sweep3D, which has no non-blocking calls, has been estimated more accurately (with error between 0.57% to 2.1%). Finally, overhead for LULESH has been underestimated (between 2.78% and 4.79%).

Discussion. Overall, the presented results show a low model error of at most 6.18%. It also shows that capturing the overhead of non-blocking calls is more challenging as Sweep3D has no non-blocking calls and results in the least model error. In the current model, we give equal weight to *wait* time in Equations 1 and 2 when both non-blocking sends and receives exist. A more advanced profiling technique could separate the *wait* time spent on *Send* from *Recv* and might provide a more accurate *Send* and *Recv* fraction. One could extract such information from the `MPI_Request` data structure and extend mpiP to retrieve fine-grained *wait* information. One other model enhancement is to extract the overhead of each P2P call individually (O_i). Then, the overall P2P overhead can be computed as $\beta = \sum_{i=1}^{n} O_i$.

Job Completion time with DINO failure recovery. The t_{clone} parameter depends on the application as it is directly related to the process image size. Moreover, increasing the input size of the application leads to larger data in memory and eventually larger t_{clone}. The total job completion time is the sum of the time to perform actual computation (t) and the time to recover from failures.

$$T_{total} = T_{red} + T_{recovery} \tag{5}$$

Let n_f be the number of failures that occur till the application completes. On aver-

Table 2
Experiment vs. modeled execution time under redundancy

	# of ranks (problem size)	1x (sec)	α	F_{Send}	F_{Recv}	β	2x Real (sec)	2x Model (sec)
CG	1024 (Class D)	22.75	.5351	.6755	.3092	.1654	25.5	26.52
	2048 (Class D)	13.85	.6507	.6027	.3725	.2423	16.38	17.21
	1024 (Class E)	232.62	.2985	.8012	.196	.0585	242.62	246.23
	2048 (Class E)	109.04	.4657	.6967	.2987	.1391	119.36	124.21
LU	1024 (Class D)	30.28	.3298	.2159	.7553	.0712	33.79	32.43
	2048 (Class D)	26.55	.4621	.1857	.7702	.0858	30.42	28.83
	1024 (Class E)	376.80	.1131	.1979	.7942	.0223	394.28	385.24
	2048 (Class E)	206.08	.1738	.1887	.7992	.0327	222.39	212.84
MG	1024 (Class D)	2.28	.3357	.4332	.317	.1064	2.59	2.52
	2048 (Class D)	1.45	.5116	.3055	.3594	.1562	1.76	1.67
	1024 (Class E)	17.18	.1458	.4903	.4162	.0606	17.77	18.22
	2048 (Class E)	9.06	.2342	.3884	.462	.0909	9.65	9.88
S3D	1024 ($100 \times 40 \times 400$)	28.06	.286	.1139	.7345	.0325	28.81	28.97
	2048 ($100 \times 40 \times 400$)	35.39	.3592	.1054	.6893	.0378	35.99	36.73
	1024 ($320 \times 40 \times 400$)	88.04	.2821	.1036	.7217	.0292	89.74	90.61
	2048 ($320 \times 40 \times 400$)	117.81	.3742	.1289	.6804	.0482	121.39	123.49
LULESH	1000 (Size=25)	270.82	.2855	.679	.599	.171	288.20	275.45
	1728 (Size=25)	328.34	.2885	.72	.631	.182	350.07	334.32
	1000 (Size=27)	351.39	.244	.669	.576	.140	371.63	356.33
	1728 (Size=27)	448.60	.2533	.719	.689	.174	468.89	456.43

age, a node failure occurs every $\frac{\theta}{n \cdot r}$. Therefore, n_f is calculated as $n_f = T_{total} \cdot \frac{n \cdot r}{\theta}$ or $n_f = T_{total} \cdot \lambda \cdot n \cdot r$. Then, $T_{recovery} = n_f \cdot t_{clone}$. In summary,

$$T_{recovery} = T_{total} \cdot \lambda \cdot n \cdot r \cdot t_{clone} \tag{6}$$

Plugging Eq. 6 into Eq. 5, the job completion time under redundancy and DINO failure recovery is estimated as:

$$T_{total} = \frac{T_{red}}{1 - \lambda \cdot n \cdot r \cdot t_{clone}} \tag{7}$$

We use Eq. 7 in Section 9 for large scale simulations.

8 Experimental Results

The node cloning experiments require insertion of our kernel module into the Linux kernel. This permission is not granted on large-scale supercomputers maintained by NSF or DOE. Thus, we conducted the experiments on a 108-node cluster with QDR Infiniband. Each node is equipped with two AMD Opteron 6128 processors (16 cores total) and 32GB RAM running CentOS 5.5, Linux kernel 2.6.32 and Open MPI 1.6.1. The experiments are demonstrating failure recovery rather than exploring compute capability for extreme scale due to the limitations of our hardware platform. Hence, we exploit one process per node in all experiments. Experiments were repeated five times and average values of metrics are reported.

Live Node Cloning Service. In this section, we evaluate the node cloning performance at process-level. We compare "CR-based" node cloning (checkpoint, transfer, restart) with our live cloning approach. We created a microbenchmark consisting of `malloc` calls (`sbrk` system calls) to control the size of the process image. It has an `OpenMP` loop with 16 threads long enough to compare the performance of our node cloning mechanism with Checkpoint/Restart(CR). In CR, we checkpoint the process locally, transfer the image to another node and then restart the snapshot (using off-the-shelf BLCR). We omit the file transfer overhead and consider the best case for CR (checkpoint and restart locally). Table 3 shows the results for different image sizes ranging from 1GB to 8GB. Our cloning approach takes less time than just checkpointing only without restart. It is more than two times faster than CR in all cases (2.24x for 1GB and 2.17x for 8GB). Cloning is performed via TCP over QDR Infiniband with an effective bandwidth of 300 MB/s. The speedup of our approach is due to creating the process while copying the process image. As a result, we parallelize the two steps that are serial in CR. Furthermore, we do not use the disk but instead transfer memory pages over the network.

Table 3
Microbenchmark performance (in sec.) of process cloning vs CR (Process with 16 threads)

Process Image Size	**Cloning**	**Checkpoint**	**Restart**	**Total CR**	**Speedup ($\frac{CR}{Cloning}$)**
1GB	**12.42**	14.98	12.85	**27.83**	2.24
2GB	**26.11**	29.79	25.63	**55.42**	2.12
4GB	**49.58**	59.79	50.39	**110.36**	2.22
8GB	**100.94**	119.10	100.20	**219.3**	2.17

Overhead of Failure Recovery. In this section, we analyze the performance of DINO. We consider 9 MPI benchmarks: (BT, CG, FT, IS, LU, MG, SP) from the NAS Parallel Benchmarks (NPB) plus Sweep3D (S3D) and LULESH. We use input

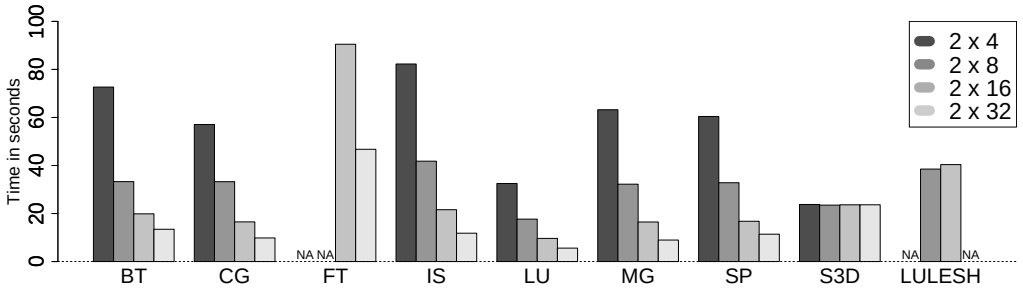
class D for NPB, size $320 \times 100 \times 500$ for Sweep3D and size 250 for LULESH. We present results for 4, 8, 16 and 32 processes under dual redundancy (CG, IS, LU, MG). We use 4, 9, 16, 25 processes for BT and SP (square numbers are required by the benchmark). FT with 4 and 8 processes could not be executed due to memory limitations. LULESH only runs with cubic numbers of processes, so we run it with 9 and 27 processes. Due to lack of support from the Infiniband driver to cancel outstanding requests without invalidating the whole work queue and lack of safe re-initialization, current experiments are performed with marker messages. Every process receives a message indicating the fault injection and acts accordingly. One rank mimics the failure by performing a `SIGSTOP`. Then the Cloning APIs are used to start the clone procedure and the discussed steps in Section 4 are performed: Pre-copy, Quiesce, Clone, Resume.

Fig. 8(a) and 8(b) depict the overhead and transferred memory size, respectively. NPB are strong scaling applications and the problem size is constant in a given class. Therefore, the transferred memory size and consequently time decreases when the number of processes increases. In contrast, Sweep3D and LULESH are weak scaling and the problem size remains constant for each process, solving a larger overall problem when the number of processes increases. Thus, weak scaling benchmarks show negligible difference in overhead and transferred process image size over different number of processes.

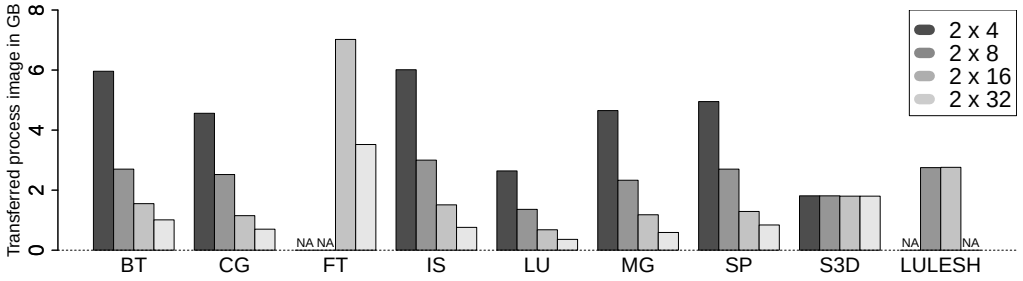
FT has the largest process image. The size of memory for FT with 16 processes is 7GB and takes 90.48 sec to transfer, while it takes 46.75 sec with 32 processes to recover from a failure when transferring 3.52GB of memory. LU has the smallest process image among NPB, its memory size ranges from 2.64GB to 0.36GB with transfer times of 32.51 sec to 5.60 sec for 4 to 32 processes, respectively. For Sweep3D, the overhead is almost constant at 23.5 sec over different numbers of processes when transferring a 1.8GB image. The same applies to LULESH with a constant process image size of 2.75GB and an overhead of 38.51 sec. [3] The relative standard deviation in these experiments is less 7% in all cases.

We also measure the time spent in each phase: pre-copy, quiesce, clone and resume for 32 processes except for LULESH, where 27 processes are used (see Fig. 8(c)). The pre-copy phase is shown on the left axis, and the rest of phases are shown on the right axis which is an order of magnitude smaller. The majority of time is spent in the pre-copy phase and the remaining three phases take about 1 second combined. These three phases take almost similar time across all the benchmarks with only small variations. In our experiments, we observed that shortly after the fault injection, a deadlock occurs. Other processes wait for the completion of a communication with the failed process and this will eventually create a chain of dependen-

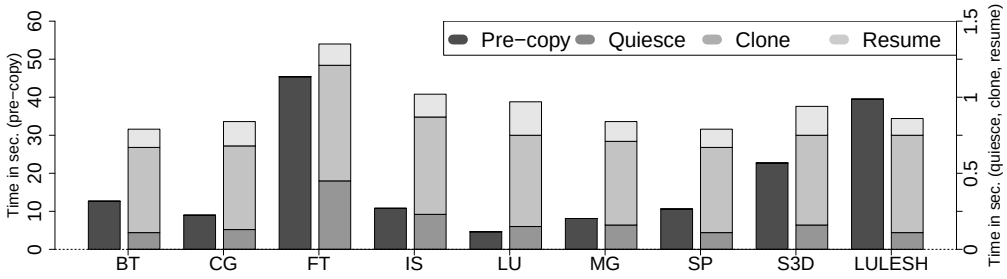
[3] In the current implementation, the memory is copied one page at a time. This lowers the performance of the cloning operation. A larger buffer size might increase the performance as the effective bandwidth could be increased.



(a) Overhead (vulnerability window)



(b) Transferred process image



(c) Overhead: Step-wise with 2×32 ranks

Fig. 8. DINO recovery from 1 fault injection for different MPI benchmarks (1 rank per physical compute node)

cies among all processes. The Quiesce phase of the DINO recovery resolved this problem successfully. Due to early deadlock occurrence, the overlap of the normal operation with the pre-copy phase was negligible for the chosen benchmarks.

9 Simulation Results

Size of Spare Node Pool. This section analyzes the effect of cloning on the average number of required spare nodes to still complete a job. Assume each node has a $MTTF$ of 50 years. On average, every $MTTF/(r \times n)$, a node fails in the system. Further, assume $r = 2$ and $T = 200$ hours. The time to complete the job with

$\beta = 0.2$, 0.4 and 0.6 can be calculated using Eq. 7. We consider large values for β to study the node pool size in more extreme cases where redundancy has larger overhead. Table 4 indicates the number of spare nodes required for successful job completion for different values of n ranging from 18 ($n = 16K$, $\beta = 0.2$) to 465 nodes ($n = 256K$, $\beta = 0.6$). In this case, we did not consider any repair for the system (MTTR =∞).

If we consider a Mean Time to Repair (MTTR) of 20 hours, the required number of spare nodes is shown in the last column of Table 4. We observe that the average number of spare nodes is ranging from 2 to 25, which is only a small fraction of total number of nodes. Assuming that nodes are repairable, the average number of required spare nodes turns out to be independent of the β value. For example, consider $n = 64K$. When $\beta = 0.2$, the job takes 252.28 hours, and we have $\lfloor 252.28/20 \rfloor = 12$ repair intervals. Similarly, for $\beta = 0.4$, there are $\lfloor 283.45/20 \rfloor = 14$ repair intervals, and for $\beta = 0.6$, there are $\lfloor 336.38/20 \rfloor = 16$ repair intervals. If we divide the number of required spare nodes by the number of repair intervals, we obtain a bound on the number of required spare nodes. This value is $\lceil 74/12 \rceil$, $\lceil 86/14 \rceil$ and $\lceil 99/16 \rceil$ for $\beta = 0.2$, 0.4 and 0.6, respectively. In all three cases, 7 spare nodes are required. Similar conditions hold for the rest, meaning that results are independent of β.

Table 4
Avg. number of required spare nodes

	MTTR = ∞			**MTTR = 20h**
Size (n)	β = 0.2	β = 0.4	β = 0.6	β = 0.2, 0.4, 0.6
16000	18	21	24	2
32000	36	42	48	3
64000	74	86	99	7
128000	156	182	208	13
256000	349	407	465	25

Job Completion Time. Next, we study the behavior of different methods at extreme scale. Plain job execution time (t) is 256 hours and β varies from 0.1 to 0.4. We use Eq. 7 to extrapolate the job completion time with DINO recovery. Under redundancy, we only re-execute the job upon job failure (when both replicas fail) and use the following equation [8]: $T_{total} = (D + \frac{1}{\Lambda})(e^{\Lambda T_{red}} - 1)$ where D is time to re-launch the job (assumed to be 2 minutes), and Λ is the system MTTF ($\Lambda = -\ln(R_{sys})/T_{red}$). R_{sys} is the overall system reliability and can be computed as: $R_{sys} = (1 - (T_{red}/\theta)^r)^{n \times r}$ [10].

Three solutions are studied: dual redundancy (2x), 2x with DINO recovery (2xD), and triple redundancy (3x). The cloning overhead is 2 minutes since only a single node pair is involved in cloning. We choose a node MTTF of $\theta = 50$ years. Fig. 9

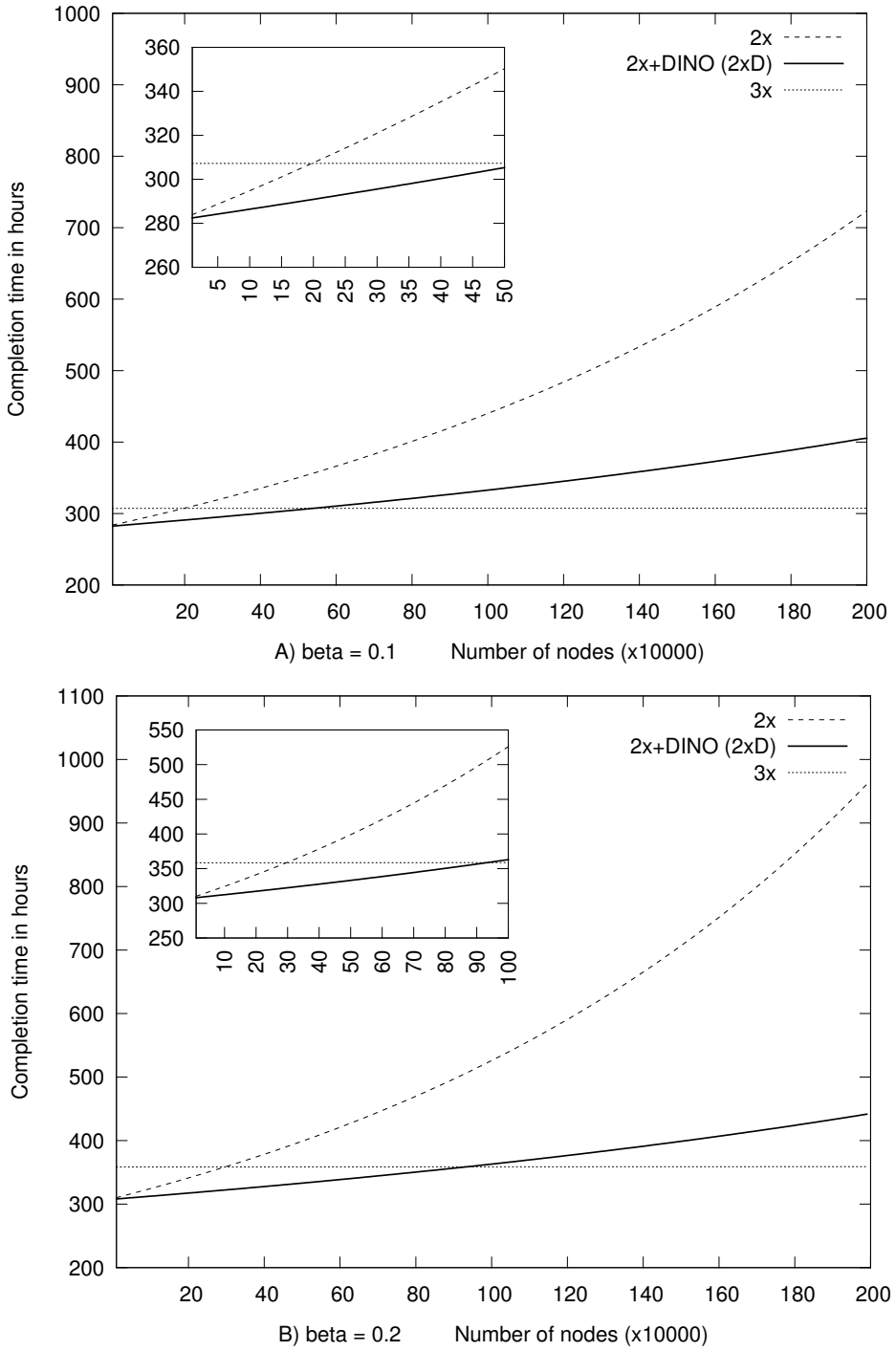


Fig. 9. Modeled job completion time

shows the job completion time for systems with different numbers of nodes ranging from 10K to 2M. 2x is the dashed line, 2xD is the solid line, and 3x is the dotted line. Based on the results from Table 2, we choose $\beta = 0.1$ and 0.2 in this experiment. Fig. 9(A+B) depict the results for $\beta = 0.1$ and 0.2. A zoomed view of the area of interest is shown on the top of each plot. This area is 10K to 500K in Fig. 9(A) and

10K to 1M in Fig. 9(B). 2xD outperforms 3x for up to 540K and 940K nodes for a β of 0.1 and 0.2, respectively. After this point, 3x provides shorter job execution times due to DINO recovery overhead for every failure, but this range is likely beyond the size of exascale systems. For $\beta = 0.1$, 2x is between 282 to 719 hours. This changes to 308.72 to 960.99 hours for $\beta = 0.2$.

Vulnerability Window. Our experimental evaluation in Section 8 showed that for typical HPC application, the DINO recovery time is in order of seconds to less than 2 minutes (depending of the application footprint per process). In this section, we reason about the probability of experiencing a soft error during the vulnerability window under dual redundancy. Suppose the vulnerability window is w, and the execution time per rank is T_p (total: $2 \times N \times T_p$). Assume that at some point of time, a process fails and there are currently $2 \times N - 1$ active process. The probability of a soft error hitting the application during w is $w/2 \times N \times T_p$. Assuming $w = 2$ minutes, $T_p = 100$ hours and $N = 1$ million nodes, this probability is very small ($1.65e^{-10}$). As a result, the effect of the vulnerability window on the correctness of application result is negligible.

10 Related Work

Our earlier work [23] provides stochastic modeling and simulation results for a system based on redundancy and process cloning. This paper presents the design and implementation of a node cloning service, the Quiesce algorithm, an experimental evaluation with fault injection and a model for job completion time under redundancy. rMPI [4] and MR-MPI [11] provide transparent redundant MPI computing. Elliott et al. [10] determine the best configuration of a combined approach including redundancy and CR. They propose a model to capture the effect of redundancy on the execution time and checkpoint interval. Ferreira et al. [12] investigate the feasibility of process replication for exascale computing. A combination of modeling, empirical and simulation experiments is presented in this work. PLR [28] provides transparent process redundancy and is capable of detecting soft errors. None of the above works deal with replica failures and only focus on providing replication capability. Our work addresses hard replica failures and solves the consistency problem in the MPI environment during failure recovery. Data parallel engines like Hadoop YARN [31] and Spark [35] find failed or slow (straggler) tasks and re-launch them on healthy compute nodes. This is fundamentally different from our approach in the sense that we clone processes on-the-fly without rolling back to prior states. In fact, Hadoop requires idem potency to start a redundant task based on HDFS for input/output consolidation while our clone mechanism avoids reexecution and does not reply on prior checkpoints. Instead, it just performs forward execution for the remaining portion of a task, albeit likely at a faster pace than the straggler. As an optimization in Spark Streaming, the work of a failed task is divided between multiple healthy nodes to allow for faster recovery [29], often with a delay to avoid

I/O contention [1]. Thus, failed tasks can be relaunched in parallel on all the other nodes in the cluster. This evenly distributes all the recomputations across many nodes and speeds up recovery from failures since one does not need to wait for stragglers but is subject to the same full reexecution as before, much in contrast to our work.

A process-level proactive live migration approach is presented in [33] with an integration within LAM/MPI. VMware Workstation [32] provides virtual machine migration and cloning. In cloning, a snapshot of the VM is created. The cloned VM is independent from the main VM. Clones are useful when one must deploy many identical virtual machines in a group. Cloning in VMs is mainly to avoid the time-consuming installation of operating systems and application software. Unlike DINO, prior work does not handle (a) new process/thread creation with the same progress but a *different* identity, (b) compensation for divergence in progress, and (c) techniques to limit divergent scopes.

11 Conclusion

We introduced DINO, a quick forward recovery method from failures in redundant computing. DINO contributes a novel live node cloning service with a *scalable* multicast variant of the bookmark algorithm and a corresponding Quiesce algorithm for consistency among *diverging* communicating tasks. With its integration into the MPI runtime system, DINO allows a redundant job to retain its redundancy level via cloning throughout job execution. Experimental results with multiple MPI benchmarks indicate low overhead for failure recovery. A model for job for redundant computing indicates that dual redundancy with cloning (DINO-style) outperforms triple redundancy up to 0.5-1M nodes depending on the communication to computation ratio, which is the range relevant to exascale computing. This means node failures can be tolerated by forward recovery with DINO with sustained resilience levels due to live cloning. In future work, SDC could also be detected (without correction) by DINO at 33% lower power and fewer nodes than triple redundancy. This is because dual redundancy requires twice the number of nodes (and power) over a non-redundant execution, and triple redundancy three times as much. With cloning, we can sustain dual redundancy after a node failure and ensure SDC detection capabilities with just a small number of spare nodes, which reduces power and number of nodes by 33% over triple redundancy.

References

[1] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *USENIX Conference on Networked*

Systems Design and Implementation, pages 185–198, 2013.

[2] Keren Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems, September 2008.

[3] Swen Böhm and Christian Engelmann. File I/O for MPI Applications in Redundant Execution Scenarios. In *Euromicro International Conference on Parallel, Distributed, and network-based Processing*, February 2012.

[4] Ron Brightwell, Kurt Kurt Ferreira, and Rolf Riesen. Transparent redundant computing with MPI. In *Euro-Par*, pages 208–218, 2010.

[5] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov 2009.

[6] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. Software Aging Analysis of the Linux Operating System. In *ISSRE*, pages 71–80, 2010.

[7] Jack Dongarra et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[8] Andrzej Duda. The effects of checkpointing on program execution time . *Information Processing Letters*, 16(5):221 – 229, 1983.

[9] Jason Duell. The design and implementation of Berkeley Labs Linux Checkpoint/Restart. Technical report, LBL, 2003.

[10] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining Partial Redundancy and Checkpointing for HPC. In *International Conference on Distributed Computing Systems*, Macau, China, June 18-21 2012.

[11] Christian Engelmann and Swen Böhm. Redundant Execution of HPC Applications with MR-MPI. In *International Conference on Parallel and Distributed Computing and Networks*, pages 31–38, February 15-17, 2011.

[12] Kurt Ferreira, Jon Stearley, James H. Laros, III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Supercomputing*, pages 44:1–44:12, 2011.

[13] D. Fiala, F. Mueller, C. Engelmann, K. Ferreira, and R. Brightwell. Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing. In *Supercomputing*, 2012.

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

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[16] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *IPDPS*, March 2007.

[17] Ian Karlin et al. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, LLNL, 2012.

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

[19] Aniruddha Marathe, Rachel Harris, David Lowenthal, Bronis R. de Supinski, Barry Rountree, and Martin Schulz. Exploiting redundancy for cost-effective, time-constrained execution of hpc applications on amazon ec2. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 279–290, New York, NY, USA, 2014. ACM.

[20] mpip.sourceforge.net.

[21] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: An empirical analysisof hardware failures on a million consumer pcs. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM.

[22] Bernd Panzer-Steindel. Data Integrity. Technical Report 1.3, CERN, 2007.

[23] A. Rezaei and F. Mueller. Sustained resilience via live process cloning. In *Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, pages 1498–1507, May 2013.

[24] A. Rezaei and F. Mueller. Tbd. In *IEEE Cluster*, September 2015.

[25] Michael Rieker and Jason Ansel. Transparent user-level checkpointing for the native posix thread library for linux. In *In Proc. of PDPTA-06*, pages 492–498, 2006.

[26] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The LAM/MPI Checkpoint/Restart framework: System-initiated checkpointing. In *LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[27] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. *SIGMETRICS Perform. Eval. Rev.*, 37(1):193–204, June 2009.

[28] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *IEEE Trans. Dependable Secur. Comput.*, 6(2):135–148, April 2009.

[29] http://spark.apache.org/streaming/.

[30] tacc.utexas.edu/stampede.

[31] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric

Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[32] VMware. Getting Started with VMware Workstation 10. Technical Report EN-001199-00, VMware Inc, 2013.

[33] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *Supercomputing*, pages 1–12, 2008.

[34] Keun Soo Yim, Z. Kalbarczyk, and R.K. Iyer. Pluggable Watchdog: Transparent Failure Detection for MPI Programs. In *International Parallel and Distributed Processing Symposium*, pages 489–500, May 2013.

[35] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.