DINO: Divergent Node Cloning for Sustained Redundancy in HPC

Arash Rezaei, Frank Mueller
Department of Computer Science, North Carolina State University, Raleigh, NC.

***Abstract*—Complexity and scale of next generation HPC systems poses significant challenges in fault resilience methods such that contemporary checkpoint/restart methods may be insufficient. Redundant computing has been proposed as an alternative at extreme scale. However, redundant approaches do not repair failed replicas, and a given job can only continue execution as long as there exists at least one healthy replica per task. Replicas are logically equivalent, yet may have divergent runtime states during job execution, which complicates on-the-fly repairs.**

In this paper, we present a redundant execution environment that quickly repairs failures via DIvergent NOde cloning (DINO). DINO contributes a novel node cloning service integrated into the MPI runtime system that solves the problem of consolidating divergent states among replicas on-the-fly. We provide execution time analysis. Experimental results over multiple benchmarks indicates that DINO can recover from failures nearly instantaneously, thus retaining the redundancy level throughout job execution. The resulting resilience of a dual redundant system with cloning nearly matches that of triple redundancy, yet at a third lower cost in terms of resources.

***Keywords*-Fault Resilience; HPC; Node Cloning**

I. Introduction

Reliability has been highlighted as a problem for next generation supercomputers [1], [2], [3]. 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 hardware or software faults. Hardware faults may result from aging [4], loss of power, and operation beyond temperature thresholds. Software faults can be due to bugs (some of which may only materialize at scale), complex software component interactions and race conditions that require rare parallel execution interleavings of tasks. (A recent study [5] has tracked 736 faults in Linux 2.6.33 over time). The de-facto method for fault tolerance in HPC is Checkpoint/Restart (CR). Applications are periodically checkpointed, and their state is written to persistent storage. Upon failure, the application is rolled back to the last checkpoint, its state is retrieved from storage in all tasks, and execution resumes from this point. However, the cost of checkpointing and especially writing checkpoints to a parallel file system (PFS) is high. For future extreme scale systems, this potentially causes CR overheads to dominate wallclock time instead of spending most time on core application execution [6], [7], [8].

An alternate resilience method is redundant computing [7], [9], [10]. It aims at improving reliability and availability of systems by allocating two or more components to perform the same work. Although redundancy adds to the cost and complexity of systems, it scales with system size. Resilience actually increases with redundancy at large scale, much in contrast to CR [7]. However, current redundant approaches do not provide a sustained redundancy level during job execution. When a process of a job fails, replicas ensure that the application can progress in execution. This requires that there by at least one healthy replica, i.e., should all replicas of an MPI task fail, then the entire job fails. Current redundant systems rely on higher degrees of redundancy (more replicas) to provide fault resilience at large scale and over extended periods of time for long-running jobs.

Considering the cost of higher degrees of redundant computing, both in terms of computing resources and power, we introduce node cloning as a means to sustain a given redundancy level. The core idea is to "repair" node failures with the assistance of healthy replicas. A healthy replica is cloned onto a spare node to recreate the failed process in "mid-flight". As a process is cloned onto a spare node with a clean operating system state, it also implicitly becomes rejuvenated in terms of the execution environment (operating system state and shared libraries) for the process. We integrate this service within the MPI runtime and provide experimental results for our system. We show that such a system not only eliminates the need (and overheads) of CR schemes, but also provides a sustained resilience level throughout job execution.

Contributions: To address shortcomings in current redundant systems, we provide the following contributions in the DINO system:

- We devise a generic node cloning service and integrate it into the MPI runtime under redundancy. It represents a reactive method that eliminates the overheads related to traditional CR schemes. There is also no need for PFS storage to keep large checkpoint files, i.e., PFS acquisition costs and system failure rates (due to absence of PFS failures) would be much lower.
- We contribute a novel method to overcome divergence in execution with minimal message logging. The execution of replicas does not occur a lock-step fashion. They can easily diverge, thereby introducing complexity to the recovery mechanism. We devised a novel algorithm to establish communication consistency to facilitate

recovery.

- We discuss DINO's performance on several MPI benchmarks. The time to recover from failures is short enough to make our approach practical. Time to regain dual redundancy after a node failure varies from 5.60 seconds (LU with 32 processes) to 90.48 seconds (FT with 16 processes), depending on process image size and cross-node transfer bandwidth.
- We analyze job completion times and extrapolates results to extreme scale. With a node MTTF of 50 years [11]. Compared to CR, dual redundancy with cloning neither introduces checkpointing overhead (due to storage, network, filesystems etc.) nor requires re-computing after restarts. At extreme scale, when CR overhead prevents jobs to make progress, our cloning approach completes the job in much shorter time and complements dual redundancy without cloning. Compared to triple redundancy, it also utilizes 33% less computing resources and power, yet provides similar job completion time up to 1 million nodes. We also show that 25 spare nodes suffice for a 256k node system when nodes can be repaired, independent of the computation to communication ratio of applications.

The paper is structured as follows: Section II lists the assumptions we make. Challenges are discussed in Section III. Section IV introduces the design and architecture of DINO. Analysis of job completion time are presented in Section V. The experimental evaluation is provided in Section VI. Section VII presents the related work. Section VIII summarizes the paper.

II. Assumptions

This work targets tightly-coupled parallel applications/jobs executing on HPC platforms using MPI-style message passing [12]. Should a single node fail, the entire job is affected and typically needs to be started from the beginning. We assume a fail-stop failure model, i.e., a computing node will stop functioning after the occurrence of a fault.

Suppose there is a job that requires t hours to run on n nodes to complete without any failures. This is called plain execution time. We consider systems with r levels of redundancy (at the process/MPI task level). Our system then consists of $r \times n$ active computing nodes, where n logical processing nodes (MPI tasks) are seen by the user while redundant shadow nodes remain transparent. We also assume the availability of a small pool of spare nodes. Spare nodes are in a powered state but initially do not execute any jobs. We further assume absence of a single common-mode fault in the system. Common-mode faults (e.g., power failure of an entire HPC system) cause all operational nodes to fail simultaneously.

We call a "sphere" the set of all nodes executing the same code with the same input. All nodes ($r \times n$) comprise a total of n spheres, where each sphere contains nodes of the same replica (same MPI task). A complete sphere failure with the primary node and its corresponding shadow nodes is considered a job failure under redundant execution. DINO additionally triggers cloning operations as soon as a node failure is detected. The intuition here is to quickly regain the original redundancy level as there is no checkpointing in the system. In contrast, if CR is used exclusively (or in conjunction with redundancy but without cloning), a sphere failure triggers the latest snapshot to be restored such that the job's execution is resumed from that state.

III. Challenges

There are a number of challenges involved in providing node cloning when MPI jobs are running. These challenges:

A. *High performance node cloning service*

A generic service that clones the processes, including the runtime/kernel state and memory content onto a spare node is the first challenge. This service requires that processes are already isolated from rest of the job, i.e. ongoing communication is finalized and no new communication will start (a.k.a, quiesce). The performance of this service affects the overall job completion time. Off-the-shelf checkpointing libraries like BLCR [13] or MTCP [14] could provide the needed functionality. However, the downtime to take the snapshot, save it to storage and retrieve it from storage followed by restoring the snapshot is quite high. Thus, a high performance approach is required.

B. *Divergent states*

One of the main technical challenges is how to overcome divergence in execution and to do so with only minimal message logging. The execution of replica nodes does not occur 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 would generally require message 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.

C. *Integration to the runtime system*

An important aspect of a process is the relation of its state to the rest of the processes. Cloning a process without careful resuming of its communication state results in inconsistency and job failure.

IV. DINO

DINO benefits from a node cloning service for a given process at its core. Node cloning creates a copy of a running process (a *source* node) onto a *spare* node. While the cloning mechanism is MPI agnostic, it is applied to processes encapsulating MPI tasks in this work. (DINO also considers the effect of cloning on the MPI runtime system, as detail

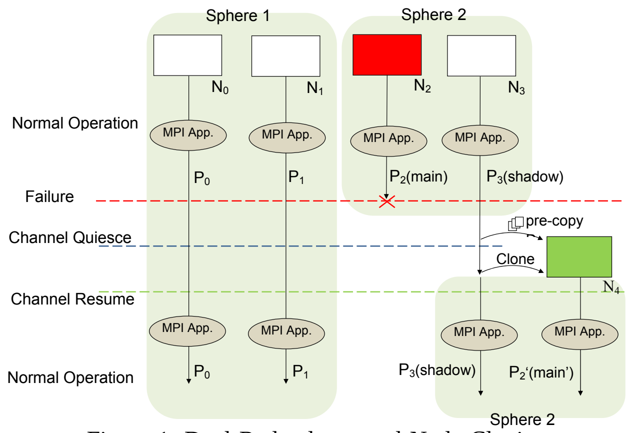


Figure 1: Dual Redundancy and Node Cloning

later.) Fig. 1 shows how the system retains dual redundancy in case of a failure. Suppose two dual redundant processes, P_2(*main*) and P_3(*shadow*), are logically equivalent (i.e., both perform the same computation) and run on N_2 and N_3, respectively. If node N_2 fails after some time, its shadow, located on N_3 (*source*), is cloned onto N_4 (*spare* node) on-the-fly.

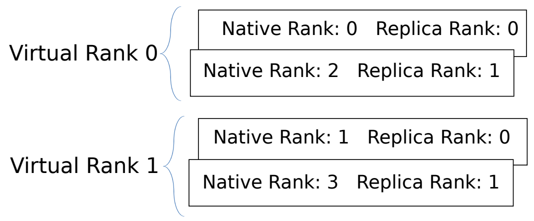
A process is created on N_4 with the same number of threads. While P_3 is performing its normal execution, its memory is "live copied" page by page to the newly created process. This happens in an iterative manner. When we reach a state where few changes in dirty pages (detailed in the implementation) remain to be sent, the communication channels are drained. This is necessary to keep the system of all communication processes in a consistent state. After this, P_3's execution is briefly paused so that the last dirty pages, linkage information, credentials, etc. may be sent to N_4, i.e., a process P_2' has been created on N_4. 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.

A. Equalization Algorithm

In the following, we explain problems rooted in communication inconsistency with an example and provide the details of how we address them. Fig. 2 shows a simple MPI example under dual redundancy in terms of MPI rank layout. Dual redundancy doubles the number of process and Fig. 2(b) indicates the corresponding terminology for virtual, native and replica ranks, where a virtual rank with its two boxes represents a sphere. The native rank is the rank assigned by `mpirun` within the range $[0, 3]$. The rank API call, `MPI_Comm_Rank`, returns the virtual rank. The MPI processes of each replica sphere, so-called replica ranks, are numbered $[0, 1]$. Thus, rank 0 and 2 are *main* and *shadow* pairs, and rank 1 and 3 are *main* and *shadow* pairs.

```
if $rank = 0$ then
    $Computation();$
    $Isend(1);$
    $Isend(1);$
    $Waitall();$
    $Computation();$
else if $rank = 1$ then
    $Computation();$
    $Recv(0);$
    $Recv(0);$
    $Computation();$
end if
```

(a) A simple program


Virtual Rank 0
Native Rank: 0 Replica Rank: 0
Native Rank: 2 Replica Rank: 1
Virtual Rank 1
Native Rank: 1 Replica Rank: 0
Native Rank: 3 Replica Rank: 1

(b) Redundancy layout

Figure 2: A simple program and its redundancy layout

$Recv(x)$ is a blocking and $Isend(x)$ is a non-blocking call. An $Isend$ call is mapped to $Isend2(x, y)$ and denotes a send to rank x and its corresponding replica. A $Recv(x)$ is mapped to a $Recv2(x, y)$ call, as well. Note that send (or receive) requests are always posted in pairs. In other words, the number of posted send (or receive) requests to any two replicas are equal. We call this the *symmetry* property and exploit it in the equalization algorithm. $Isend2$ and $Recv2$ are as follows, where $waitall$ in the latter blocks until the preceding non-blocking calls finish.

```
$Isend2\ (x, y)\{$          ||    $Recv2\ (x, y)\{$
    $Isend(x);$             ||        $Irecv(x);$
    $Isend(y);$             ||        $Irecv(y);$
$\}$                        ||        $Waitall();$
                            ||    $\}$
```

Fig. 3 describes the execution of 4 ranks along with a failure scenario. Suppose at the mark "X" in Fig. 3, rank 0 fails. Then ranks 1 and 3 get the first message from 2 and are blocked to receive a message from 0. Rank 2, after sending all of its messages, continues execution and reaches the waitall. At this point, all processes are blocked and a recovery strategy is required.

Note that we clone rank 2 to a spare node to create $0'$. Rank $0'$ starts executing the application from the *waitall*. Therefore, neither 0 nor $0'$ ever execute the Send2 calls (dotted area in Fig. 3). Consequently, ranks 1 and 3 will never receive those two messages from $0'$. In the first stage of the equalization algorithm, all outstanding sends where

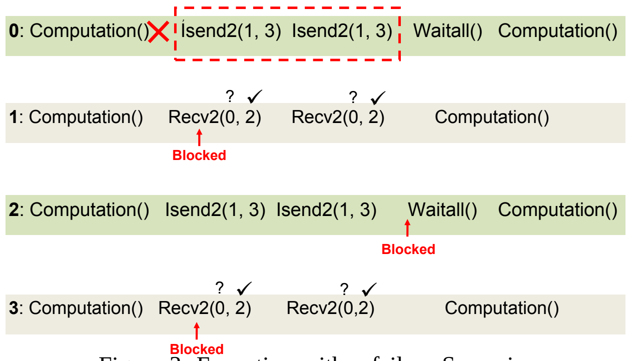


Figure 3: Execution with a failure Scenario

rank 2 is the receiver are drained. Recv requests are posted by 1 and 3 and the message is saved in a temporary buffer (indicated by check-marks) whose size is bounded by the number of asynchronous events being awaited next. Receives from 0, highlighted by question marks, remain a challenge. Note that dual (identical) messages are received from rank 2. Ranks 1 and 3 simply cancel the outstanding (ongoing) receive request from 0 and skip incrementing a message counter on the next receive from 0. This provides message equalization across the clones in this example.

Algorithm 1 shows the steps for Quiesce. At first, all outstanding send requests involving *shadow* are drained. A modified version of the bookmark exchange protocol [15] is utilized. The original bookmark protocol creates a consistent global snapshot of the MPI job. In our modified bookmark protocol, each process communicates with *shadow* the number of messages it has sent to *shadow* and receives how many messages are sent by *shadow* (and vice versa). Then, the following question can be answered: Have I received all the messages that *shadow* has put on the wire? If not, then message(s) remain on the MPI communication channel (e.g. buffered or in transmission) and should be drained. Processes exchange message counts, and Recv requests are posted to drain and save them in temporary buffers. Later, during normal execution when a Recv is posted, these drain lists are first consulted to service the request. At the end of the drain phase, no more outstanding send requests to/from *shadow* nodes exists in the system.

In general, three cases could occur with regard to the number of received messages to a rank R with respect to communication with the failed rank *main* and its *shadow* after the drain phase:
(1) Received (*main*) $=$ Received (*shadow*)
(2) Received (*main*) $<$ Received (*shadow*)
(3) Received (*main*) $>$ Received (*shadow*)
Case 1 creates no problem as it reflects message symmetry within a sphere. Case 2 is discussed above and requires a combination of canceling and skipping future message events until counters are equalized. In case 3, *main'* will send messages that have already been received by R before it failed. Thus, R must re-issue receives from *main'* up to an equalized count but silently absorbs the messages, i.e., by using a dummy receive buffer. Silent absorption is crucial since computation in R may have advanced to the point where data destinations of message receives could have been modified by calculations. The above is reflected in stage 3 of Algorithm 1. Similarly, three cases can be considered for the number of sent messages after the drain phase:
(1) Sent (*main*) $=$ Sent (*shadow*)
(2) Sent (*main*) $<$ Sent (*shadow*)
(3) Sent (*main*) $>$ Sent (*shadow*)

Algorithm 1 Equalize Algorithm
```
/* 1. Exchange communication state with shadow */
if $rank = shadow$ then
    bookmarks $*array$;
    for $(i = 0; i < nprocs; i++)$ do
        if $i = shadow \mid\mid i = Main$ then
            continue;
        end if
        /*send bookmark status, then receive into
        appropriate location in bookmarks array */
        $send\_bookmarks(i)$;
        $recv\_bookmarks(i, array[i])$;
    end for
else
    bookmark $bkmrk$;
    /*Receive remote bookmark into bkmrk then send*/
    $recv\_bookmarks(shadow, bkmrk)$;
    $send\_bookmarks(shadow)$;
end if

/* 2. Calculate in-flight msg(s) and drain them */
    Cal_and_Drain();

/* 3. Recvs: Calculate skip or repeat */
if $rank \neq shadow$ then
    $X \leftarrow Received(shadow) - Received(Main)$;
    if $X > 0$ then
        $skip\_recv \leftarrow X$
    else if $X < 0$ then
        $repeat\_recv \leftarrow |X|$
    end if
end if

/* 4. Sends: Calculate cancel */
if $rank \neq shadow$ then
    $X \leftarrow Sent(shadow) - Sent(Main)$;
    if $X > 0$ then
        $cancel\_send \leftarrow X$
    end if
end if
```

Case 1 is already symmetric, no action is required. Case 2 is fixed by canceling the outstanding sends of R to main. As the number of posted sends to main and shadow are equal (symmetry property), the difference is due to ongoing send requests, which can be canceled. This is reflected in stage 4 of the algorithm. Case 3 is impossible due to the symmetry property, which ensures that *posted* sends are represented by symmetric sent-counts due to message draining before cloning.

This algorithm does not support wild-cards and assumes collective operations implemented as point-to-point.

B. Architecture with Open MPI

Fig. 4 shows our system architecture where novel DINO components are depicted as shaded boxes. RedMPI provides a transparent interpositioning layer for MPI calls between 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 command `mpirun` interacts with the cloning APIs to launch tools on source/spare nodes. The node cloning system provides generic process-level cloning functionality via extensions to BLCR [13]. The quiesce and resume phases are triggered through signals relayed to each process. The quiesce phase includes the equalization algorithm and pauses the communication. Subsequently, the resume phase updates the internal data structures and resumes the communications.

C. Implementation

The node cloning service consists of three extension tools to BLCR named `restore`, `pre-copy` and `clone`. In the following, we explain the steps of cloning concisely.

Pre-copy. This phase transfers a snapshot of the memory pages in the process address space, which are communicated

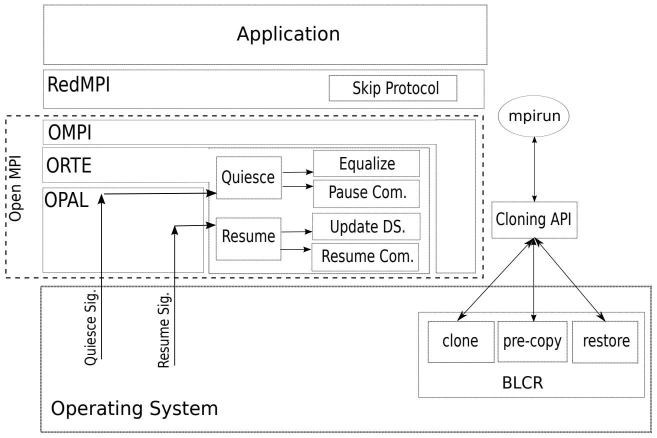


Figure 4: DINO Architecture

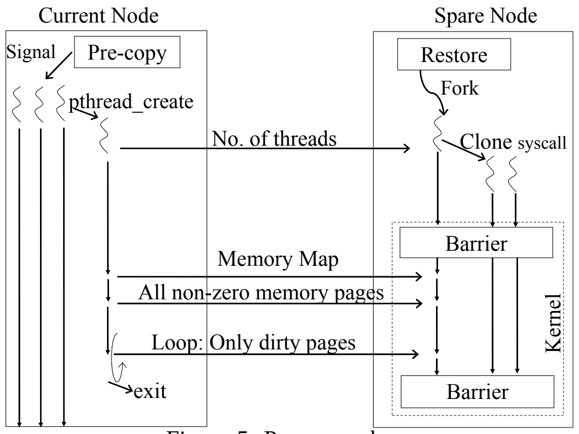


Figure 5: Pre-copy phase

to the spare node while normal execution of the process continues on the source node (see Fig. 5). We use TCP sockets to create a communication channel between *local* and *remote* nodes. The pre-copy approach is similar to [16]. The `pre-copy` tool signals the process. The signal handler then creates a thread and opens a communication channel with the `restore` tool on the spare node. Vital meta data, including number of threads, is transferred. The receiver side creates the required threads and enters a barrier. Then, threads enter the kernel and one thread walks through the page table and unmaps all virtual memory regions. The barrier ensures that all threads exit the user mode safely before the unmapping operation. 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 by a dirty bit. We apply a patch to the Linux kernel to keep track of the modified memory pages via main memory unit (MMU) dirty bits. In this approach, we shadow the dirty bit within the reserved bits of a page table entry (PTE). When the aggregate size of dirty memory is lower than a threshold or the difference of aggregate dirty memory in consecutive iterations is less than a threshold, the pre-copy thread exits (e.g., the threshold may be 1MB).

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/recv requests. We build this phase on top of the functionality for message draining provided by the CR module of Open MPI [17]. The equalization algorithm described in Section IV-A is implemented here. Through careful synchronization, we ensure that the quiesce phase never interrupts an MPI call or violates the symmetry property.

Clone. This phase stops the process for a short time to transfer a consistent image 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 *P'*. 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. Then, POSIX interval timers are transferred and saved, but they are not activated yet. We next transfer file information and restore open files. Files opened in write mode on a globally shared directory (e.g., NFS) can cause problems (due to access by both *Shadow* and *main′*), a problem considered in orthogonal work [18]. As the very last step, the POSIX interval timers are resumed.

Channel Resumption. In this phase, processes re-establish their communications channels with *Shadow* and *main′*. All processes receive updated job mapping information, reinitialize their Infiniband driver and publish their endpoint information.

D. Runtime Requirements

MPI System Runtime. In Open MPI, daemon processes, created at job launch, run on each node. Daemons assist as part of the out-of-band (OOB) channel, redirect output, and play a role in the initialization and finalization of the job. 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 this 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. Another issue is posed by the fact that native ranks of *Shadow* and *main′* at the end of cloning have identical state. We overwrite the native rank of *main′* in the MPI layer with the corresponding native rank within its sphere (cf. Fig. 2(b) for the rank layout). Finally, there are communication calls issued by the RedMPI layer 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 computing *skip* and *repeat* numbers of algorithm 1.

OS Runtime. 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.

V. 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) There is a pool of spare nodes available to be used in case of failures. Failures can occur at any time, even during cloning. (3) In case of an unsuccessful cloning (e.g., a failure in the spare node), we retry the operation. We assume that the healthy replica itself never fails during cloning. (The short time required for cloning of seconds relative to job executions of hours/days allows us to make this assumption.) We consider exponential failure distribution for the system since it has been shown that although Weibull distribution may capture failure distributions more accurately, exponential distribution provides sufficient accuracy in HPC [19].

We use the following notation:

- t: Failure-free execution time of the application
- r: Degree/level of Redundancy
- α: Communication/computation ratio of the application
- θ: MTTF of one compute node
- λ: Failure rate of one compute node
- Θ: MTTF of the whole system
- Λ: Failure rate of the whole system

The total application execution time with redundancy can be calculated as:

$$t_{red} = (1-\alpha)t + \alpha t r \tag{1}$$

Redundancy and Cloning. We add one more parameter for our calculation in this section:

- c: Time for failure-free execution of a cloning operation

The cloning operation can fail. The probability of system survival until time t is $e^{-\Lambda t}$. Thus, the probability of failure in $(0, c)$ is $1 - e^{-\Lambda c}$. The expected cloning time can be calculated as a sum of two cases. The derivation of the expected cloning time is presented below.

$$\begin{aligned} t_{clone} =& Pr(failure\, before\, c) \\ & \times (expected\, time\, of\, failure\, in (0,c)) \\ & + Pr(failure\, after\, c) \times (c) \end{aligned}$$

$$\begin{aligned} t_{clone} &= (1-e^{-\Lambda c}) \int_0^c (t\Lambda e^{-\Lambda t} dt) + (e^{-\Lambda c})(c) \\ &= (1-e^{-\Lambda c})(-e^{-\Lambda c}(c+\frac{1}{\Lambda}) + \frac{1}{\Lambda}) + ce^{-\Lambda c} \end{aligned}$$

The total time is the sum of the time to perform actual computation (t) and the time to recover from failures.

$$T_{total} = t_{red} + t_{recovery}$$

Let n_f be the number of failures that occur till the application completes. On average, 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, $t_{recovery} = n_f \cdot t_{clone}$. Thus, the job completion time under redundancy and cloning is calculated as:

$$\begin{aligned} T_{total} &= t_{red} + T_{total} \cdot \lambda \cdot n \cdot r \cdot t_{clone} \\ &= \frac{t_{red}}{1 - \lambda \cdot n \cdot r \cdot t_{clone}} \end{aligned} \tag{2}$$

Checkpoint and Restart. We add the following two parameters for our analysis in this section:

- C: Time to take a checkpoint of the job
- R: Time for restarting from a checkpoint

We use Daly's formula [20] to compute the optimal checkpoint interval as

$$\delta_{opt} = \sqrt{2C\Theta}\left[1 + \frac{1}{3}\left(\frac{C}{2\Theta}\right)^{1/2} + \frac{1}{9}\left(\frac{C}{2\Theta}\right)\right] - C. \tag{3}$$

The Job completion time with CR is computed using [8]

$$T_{total} = \frac{t + \frac{tC}{\delta}}{1 - \Lambda t_{RR}}, \tag{4}$$

where the time spent on restart and rework is

$$\begin{aligned} t_{RR} = &\left(1 - e^{-\frac{(R+t_{lw})}{\Theta}}\right)\left[-e^{-\frac{(R+t_{lw})}{\Theta}}(R + t_{lw} + \Theta) + \Theta\right] \\ &+ e^{-\frac{(R+t_{lw})}{\Theta}} \cdot (R + t_{lw}) \end{aligned}$$

and the lost work time is

$$t_{lw} = \frac{1}{\left(1 - e^{\frac{-(\delta+C)}{\Theta}}\right)}\left[\Theta - \Theta e^{-\frac{\delta}{\Theta}} - \delta e^{-\frac{(\delta+C)}{\Theta}}\right].$$

VI. Experimental Results

Experiments were conducted on a 108-node cluster with QDR Infiniband. Each node was 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 limits 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.

A. Node Cloning Service

We created a microbenchmark consisting of `malloc` 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 CR. In CR, we checkpoint the process locally, transfer the image to another node and then restart the snapshot. We omit the file transfer overhead and consider the best case for CR (checkpoint and restart locally). Fig. 6 shows the results for different image sizes ranging from 1GB to 8GB. Our cloning approach takes less time than just checkpointing only without restart. Our cloning is more than two times faster that 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.

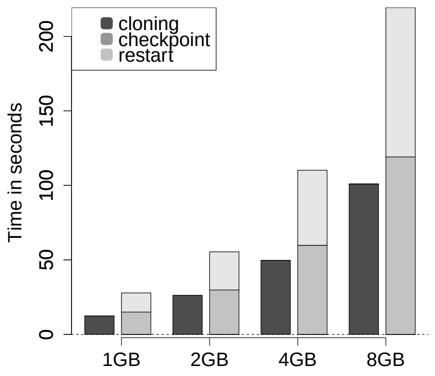


Figure 6: Microbenchmark (single process with 16 threads) – Node cloning vs CR

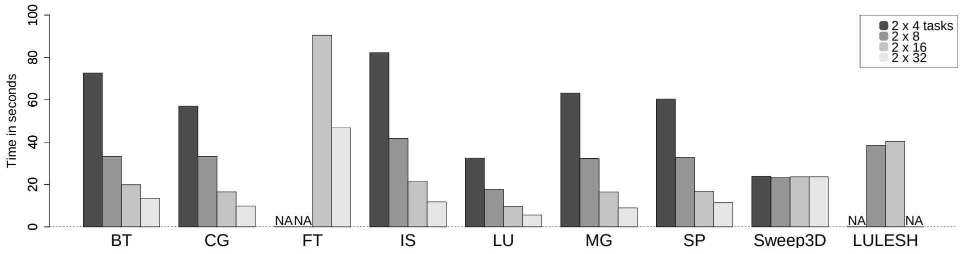
B. Overhead of Failure Recovery

In this section, we analyze the performance of DINO. We consider 9 MPI benchmarks: (BT, CG, FT, IS, LU, MG, SP) from NAS Parallel Benchmarks (NPB) plus Sweep3D 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 [21]. 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. FT with 4 and 8 processes could not be executed due to memory limits. LULESH only runs with cubic numbers of processes, so we run it with 9 and 27 processes.

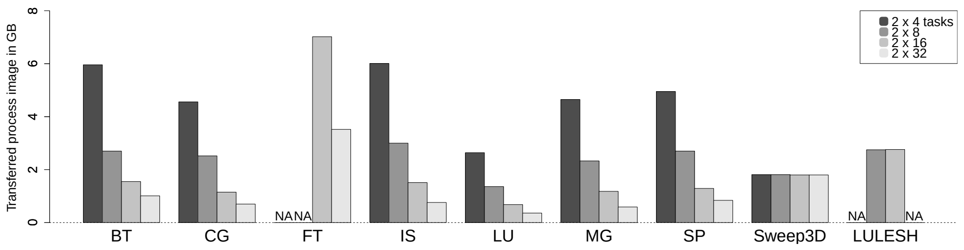
Due to lack of support from the Infiniband driver to cancel outstanding requests without invalidating the whole work queue and lack of safe re-initialization, current experiments are performed with marker messages. Every process receives a message indicating the fault injection, and acts accordingly. In other words, we inject faults during computation and measure the cloning time and size of the transferred process image.

Fig. 7(a) and 7(b) depict the overhead and transferred data, respectively. NPB are strong scaling applications and the problem size is constant in a given class. Therefore, the transferred data 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. As a result, weak scaling benchmarks show negligible difference in overhead and transferred process image over different number of processes.

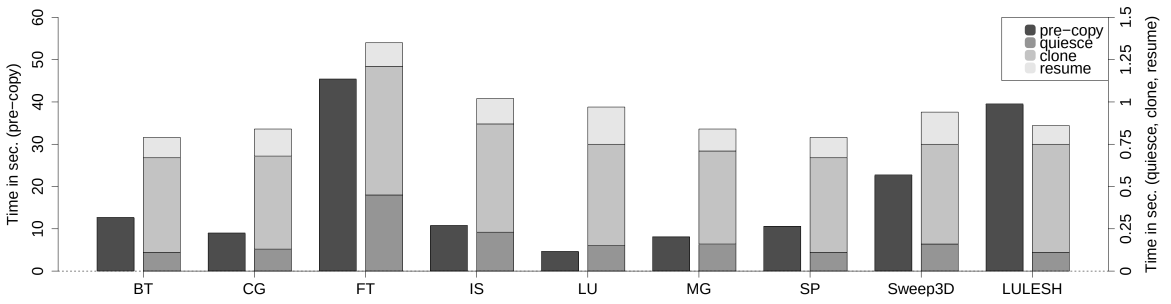
FT has the largest process image. The size of transferred data for FT with 16 processes is 7GB and takes 90.48



(a) Overhead



(b) Transferred process image



(c) Overhead: Step-wise with 2 $\times$ 32 processes

Figure 7: MPI Performance Evaluation – Recovery from 1 fault injection

sec, while it takes 46.75 sec with 32 processes to recover from a failure when transferring 3.52GB of data. LU has the smallest process image among NPB, data 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. The relative standard deviation in these experiments was 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. 7(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 sec combined. These three phases take almost similar time across all the benchmarks with only small variations.

C. *Comparison of different methods*

Next, we compare the job completion time in three scenarios with one fault injection. We compare 1x with CR, 2x with cloning and 3x with voting. We use LULESH with size 250 and run it for 100 iterations with 27 processes. We selected this setting to solve a larger problem size with longer execution time. The plain application finishes in approximately 64 minutes. Under 1x with CR, we choose the checkpoint interval as 15 minutes. Therefore, a total of four checkpoints is taken during the execution of the application. In this scenario, we inject a fault at half the checkpoint interval (7.5 minutes). Then, we restart from the latest checkpoint to recover from failure. Table I depicts the timing of each step. Out of the total time, 4.5 minutes is spent on checkpointing, 0.85 minutes on restarting, and 7.5 minutes on rework. The job completion time is 77.29 minutes.

For 2x and 3x, we inject the fault at a place which is logically equivalent to where we injected in the 1x case. Under 2x, we inject one fault resulting in a complete failure of a replica and use cloning to regain the dual redundancy

Table I: Completion Time (min.) of LULESH with injecting 1 fault (plain time is 64 min.)

Method	**App.**	**checkpoint**	**restart**	**rework**	**cloning**	**voting**	**Total (minutes)**
1x+CR	64.44	4.5	0.85	7.5	▨	▨	**77.29**
2x+Cloning	65.12	▨	▨	▨	1.04	▨	**66.16**
3x+Voting	65.58	▨	▨	▨	▨	0.18	**65.76**

level. Table I shows the time spent in the application is slightly more than the plain execution time. The extra time is overhead due to redundancy. Note that we are running these experiments at a small scale where the redundancy overhead is very small (unlike for petascale or beyond). The cloning time is 1.04 and total time is 66.16 minutes. This shows a speedup of 1.16x in job completion time compared to 1x. Under 3x, once a fault is injected, one process constantly sends incorrect messages. The processes receiving a message from the faulty process then need to engage in a voting phase to figure out the correct message. The application time is measured as 65.58, voting overhead as 0.18, and total time as 65.76 minutes. Overall, CR overheads are higher than those of redundant computing.

D. Simulation

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 $\alpha = 0.2$, 0.4 and 0.6 can be calculated using Eq. 2. Table II indicates the number of spare nodes needed for successful job completion for different values of n ranging from 18 ($n = 16K$, $\alpha = 0.2$) to 465 nodes ($n = 256K$, $\alpha = 0.6$). 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 needed number of spare nodes are shown in the last column of Table II. As we can see, 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 α value. For example, consider $n = 64K$. When $\alpha = 0.2$, the job takes 252.28 hours, and we have $\lfloor 252.28/20 \rfloor = 12$ repair intervals. Similarly, for $\alpha = 0.4$, there are $\lfloor 283.45/20 \rfloor = 14$ repair intervals, and for $\alpha = 0.6$, 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 $\alpha = 0.2$, 0.4 and 0.6, respectively. In all three cases, 7 spare nodes are required. Similar conditions holds for rest, meaning that results are independent of α.

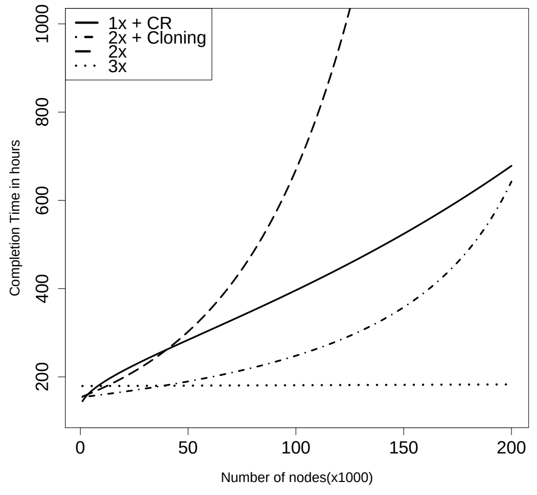
Job Completion Time. Next, we study the behavior of different methods at extreme scale. Plain job execution time (t) is 128 hours and the communication to computation ratio (α) is 0.2. We use Equations 2 and 4 to extrapolate the job completion time with cloning and CR, respectively. Under redundancy we re-execute the job in case of failure and use the following equation [22]:

Table II: Avg. # Required Spare Nodes

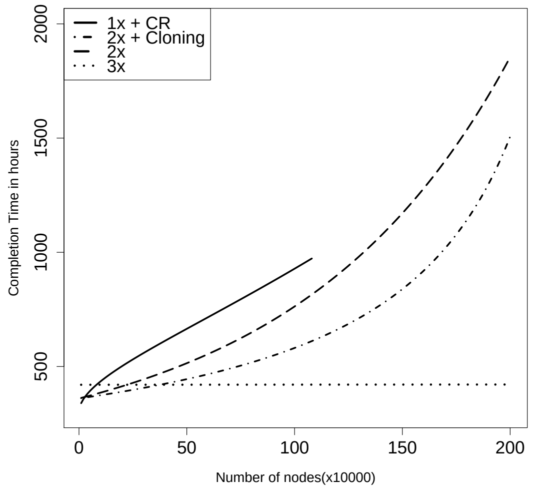
	MTTR = ∞			**MTTR = 20h**
Size (n)	α **= 0.2**	α **= 0.4**	α **= 0.6**	α **= 0.2, 0.4, 0.6**
16000	18	21	24	2
32000	36	42	48	3
64000	74	86	99	7
128000	156	182	208	13
256000	349	407	465	25

$$T_{total} = (D + \frac{1}{\Lambda})(e^{\Lambda t_{red}} - 1)$$

where D is time to re-launch the job, which is assumed to be as 5 minutes. The number of nodes ranges from 1K to 200K. We choose two node MTTF values: $\theta = 5$, 50 years. Four solutions are studied: 1x with CR, 2x with cloning, 2x and 3x. We use Daly's formula (Eq. 3) to compute the optimal checkpoint interval for a constant checkpoint and restart overhead of 10 minutes each. The cloning overhead is half of the checkpoint overhead (5 minutes) since only a single node pair involved in cloning requires high bandwidth communication while CR imposes PFS overheads across all nodes. (In fact, the difference may be differences may even be more in favor of cloning at extreme scale.) Fig. 8(a) shows that completion time for 1x with CR increases from 144.79 to 678.31 hours as the number of nodes increases from 1K to 200K. For 2x with cloning, the execution time first increases slightly and then with sharper slope for large system sizes. The steeper slope is due to the cloning overhead at each failure to regain the redundancy degree. In contrast, 2x without cloning is not useful at all considering the much lower cost of 2x with cloning. For 3x, the system MTTF is long enough that it can accommodate job execution (close to 180 hours) without sphere failures, but at the cost of 50% resources (computational and networking) as well as power. This shows that cloning enables job execution at a scale that would otherwise be impossible under dual redundancy. Results for $\theta = 50$ and $t = 128$ hours (omitted due to lack of space) show that 2x with cloning outperforms 1x with CR from about 25k nodes and still provides lower completion times than 3x beyond 200k nodes at lower resource and power costs, which is in the projected range of exascale. Results for $\theta = 50$ and $t = 300$ hours for up to 2 million nodes (see Fig. 8(b)) show that dual redundancy with cloning provides the shortest completion times from about 10k nodes to 35k nodes and still may remain more competitive than 3x up to 100k nodes with a completion time of about 600 hours compared to 450 hours for 3x since the latter has 50% higher



(a) $\theta = 5$ years, t = 128 hours



(b) $\theta = 50$ years, t = 300 hours

Figure 8: Modeled job Completion Time

resource requirements than the former. It also shows that 2x with cloning is always better than 2x without cloning.

VII. Related Work

rMPI [23] provides transparent redundant computing. rMPI is implemented using the profiling layer of MPI. It does not support certain complex MPI communications and relies on the MPI library to implement collective operations. MR-MPI [9] supports partial and full replication and uses PMPI, the MPI performance tool interface, to intercept MPI calls. Work in [8] determines the best configuration of a combined approach including redundancy and CR. They propose a cost model to capture the effect of redundancy on the execution time and checkpoint interval. Their result shows the superiority of full redundancy over partial redundancy in terms of execution time and specifically dual redundancy. Our aim is to avoid using CR with redundant computing due to its higher cost, power consumption, increase in the number of components and its potentially high PFS failure rate.

Work in [24] investigates the feasibility of process replication for exascale computing. A combination of modeling, empirical and simulation experiments is presented in this work. The authors show that replication outperforms traditional CR approaches over a wide range of the exascale system design space. RedMPI [10] allows the execution of MPI applications in a redundant fashion and provides SDC detection/correction. Our work builds on RedMPI but investigates cloning with redundancy.

A process-level proactive live migration approach is presented in [25]. It includes live migration support realized within BLCR, combined with an integration within LAM/MPI. Their experimental results show low overhead. They also compare process-level live migration against operating system migration running on top of Xen virtualization. [26] proposes a framework and architecture for proactive fault tolerance in HPC, including health monitoring and feedback control-based preventive actuation. This work investigates the challenges in monitoring, aggregating data and analysis. VMware Workstation [27] 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. Our work goes one step further by cloning applications and their runtime state, which is dependent on other nodes. This is a much harder problem.

VIII. Conclusion

In this paper, we introduced DINO, an approach to quickly recover from failures in redundant computing. DINO contributes a novel node cloning service, which has been integrated into the MPI runtime system. The approach contributes a novel solution to consolidating divergent states among replicas on-the-fly. This eliminates the need for checkpointing and consequently the cost of maintaining dedicated storage and filesystem to host job snapshots. Via cloning, DINO allows an n-level redundant job to retaining its redundancy level throughout job execution. Experimental evaluation using multiple MPI benchmarks suggests low overhead of DINO for failure recovery. It further demonstrates that the resilience of a dual redundant system with cloning nearly matches that of triple redundancy for exascale system size ranges, yet at a third lower cost in terms of hardware resources and power.

References

[1] K. Bergman *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[2] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 374–388, Nov 2009.

[3] J. Dongarra *et al.*, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[4] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software Aging Analysis of the Linux Operating System," in *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ser. ISSRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 71–80.

[5] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten years later," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 305–318, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1961295.1950401

[6] 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.

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

[8] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining Partial Redundancy and Checkpointing for HPC," in *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS) 2012*, Macau, China, Jun. 18-21 2012.

[9] C. Engelmann and S. Böhm, "Redundant execution of HPC applications with MR-MPI," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*. Innsbruck, Austria: ACTA Press, Calgary, AB, Canada, Feb. 15-17, 2011, pp. 31–38.

[10] 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 *Proceedings of the 2012 IEEE conference on Supercomputing*, ser. SC '12, 2012.

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

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[13] J. Duell, "The design and implementation of Berkeley Labs Linux Checkpoint/Restart," Lawrence Berkeley National Laboratory, Technical Report, 2003.

[14] M. Rieker and J. Ansel, "Transparent user-level checkpointing for the native posix thread library for linux," in *In Proc. of PDPTA-06*, 2006, pp. 492–498.

[15] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine, "The LAM/MPI Checkpoint/Restart framework: System-initiated checkpointing," in *in Proceedings, LACSI Symposium, Sante Fe*, 2003, pp. 479–493.

[16] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "Proactive Process-Level Live Migration in HPC Environments," in *SC*, 2008.

[17] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 03 2007.

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

[19] K. Ferreira, "Keeping Checkpointing Viable for Exascale Systems," Ph.D. dissertation, Universoty of New Mexico, Albuquerque, 2012.

[20] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[21] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[22] A. Duda, "The effects of checkpointing on program execution time," *Information Processing Letters*, vol. 16, no. 5, pp. 221 – 229, 1983.

[23] R. Brightwell, K. Kurt Ferreira, and R. Riesen, "Transparent redundant computing with MPI," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 208–218.

[24] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 44:1–44:12.

[25] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 43:1–43:12.

[26] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive Fault Tolerance Using Preemptive Migration," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 252–257.

[27] "Getting started with vmware workstation 10," VMWare Inc, Tech. Rep. EN-001199-00, 2013.