Sustained Resilience via Live Process Cloning

Arash Rezaei
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: arezaei2@ncsu.edu

Frank Mueller
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: mueller@cs.ncsu.edu

***Abstract*—More flexible fault tolerance approaches with lower overhead are a must for the next generation of supercomputers that rely on massive numbers of computational elements.**

This work proposes a reactive method for fault resilience in high-performance computing (HPC) systems based on forward execution instead of rollback to checkpoints. We study the feasibility of combining redundancy with live process cloning to create highly reliable HPC systems. The main motivation is to avoid costly checkpoint restart approaches. We present live process cloning as a mechanism to create a copy of a running process on-the-fly. We show that the reliability of a dual redundant system with live process cloning is as good as a triple redundant system even for very large systems. We also investigate the effect of node failure and the changes in Mean time to Interrupt (MTTI) of the application. This provides a better understanding of the available time to recover from a failure by cloning a healthy replica.

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

I. Introduction

With the ever growing size of computing systems, reliability problems and their effects on the system performance and correctness are becoming a widespread concern. Resilience describes the ability of a system consisting of many components to provide sustained reliability, i.e., to provide its functionality even when a subset of its components fail over a period of time.

This work targets tightly-coupled parallel applications/jobs executing on HPC platforms over N compute nodes using MPI-style message passing [1]. Should a single node fail, the entire job is affected and typically needs to be started from beginning.

Node failures could happen due to hardware or software faults. Hardware faults could be due to aging, power loss, operation beyond certain temperature thresholds or bit flips due to radiation from space or small fabrication sizes. Software faults can be due to bugs that seldom materialize, complex software component interactions and race conditions that require unlikely but possible parallel execution orders of tasks. For example, a recent study [2] has tracked 736 faults in Linux 2.6.33 over time.

Software aging is another phenomenon that affects the system reliability. Software starts from a stable state. After some time, data corruption, memory leakage or fragmentation of storage cause a transition to an unstable state. This situation may lead to node failure. Recent work [3] gives an analysis of software aging in the Linux kernel.

In this context, an increasing number of techniques for predicting, isolating and managing faults are being proposed. This is of particular interest to future exascale systems due to their complexity with millions of cores. Failure management for such complex systems is a must.

One method to tolerate faults in HPC is the traditional Checkpoint/Restart (C/R) approach. Applications are periodically checkpointed, and their state is written to a parallel file system (PFS). Upon failure, the HPC application is rolled back to the last checkpoint, retrieved from the PFS by all tasks, and execution resumes from this point. However, the cost of checkpointing and especially writing checkpoints to the PFS is high. Furthermore, checkpoint intervals become shorter due to system scale since the system mean-time-to-failure (MTTF) decreases as the number of nodes increases. This potentially causes C/R overheads to dominate wallclock time instead of useful application execution for future exascale systems [4], [5], [6]. In long-running applications, software aging may also pose a problem as the state of the software system may degrade over time.

Another resilience method is redundancy, which 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 with increasing number of nodes, much in contrast to C/R [5].

This work is based on a fail-stop failure model, i.e., a computing node will stop functioning after the occurrence of a failure. Thus, fault propagation, Byzantine faults [7], and silent data corruption (SDC) are out of scope. (It shall be noted that detection of bit flips under dual redundancy [8] is orthogonal and meshes well with our cloning work.)

Contributions: The objective of our work is to study the feasibility of a system based on a combination of redundant execution and live process cloning. We will show that such a system not only eliminates the need (and overheads) for using C/R schemes, but that a given resilience level can be retained throughout job execution even as nodes fail.

The core idea is to handle node failures utilizing live process cloning to duplicate a healthy replica onto a spare node in order to retain a given redundancy level. 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 for the process.

Let us assume a system with r levels of redundancy. Our system then consists of $r \times N$ active computing nodes where N logical processing nodes are seen by the user while redundant shadow nodes remain transparent. We also assume the availability of a small pool of spare nodes. Spares are in a powered state but initially do not run 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 do allow multiple nodes to fail at a time so long as they are within different redundant *spheres*.[1] (A complete sphere failure with the primary node and its corresponding shadow nodes no longer allows one to engage in cloning, i.e., the entire system has failed.)

(1) We further show that a dual redundant system with live cloning will provide the equivalent resilience of a triple redundant system, yet at lower cost in terms of resources (required nodes) and communication overhead.

(2) We analyze the effect of node failure on a dual redundant system and the changes in MTTI of the application. This provides insight into the length of a required time window to clone a healthy replica before resuming in the regular, more reliable dual-redundant state.

(3) We also show that the number of spare nodes required is quite small under cloning.

The approach has the follows benefits:

- It eliminates the overhead related to traditional C/R schemes. There is also no need for PFS storage to keep large checkpoint files, i.e., the PFS acquisition costs would be much lower.
- It provides high reliability for long-running jobs even on very large systems.
- It represents a reactive method that provides high reliability and eliminates the need for potentially inaccurate failure prediction, e.g., due to live migration [9], [10], [11].

Section II summarizes past work on attaining fault tolerance in HPC. An overview of live process cloning is provided in III. Section IV develops a model for reliability analysis under cloning based on Stochastic Activity Networks (SANs), which are concisely introduced, as well as an analysis of node failure. Section V summarizes our findings.

[1]A sphere encompasses all shadow nodes of a cloned node, i.e., nodes executing the same code with the same input.

II. Background and Related Work

This section investigates different approaches that have been pursued to achieve higher degree of reliability in HPC.

A. Checkpoint-based Methods

These methods consider the system as a collection of processes that are communicating over a network. C/R methods periodically take snapshots of the processes related to a job, create a consistent global state and save it to a persistent storage system. In case of a failure, rollback recovery reloads the last checkpoint, and the job is restarted. Common approaches mainly use a single process C/R system as the core mechanism and then extend that to a system-wide C/R implementation inside or on the top of an MPI library.

CoCheck [12] is one of the earliest systems that implements C/R on top of Condor [13] for Parallel Virtual Machine (PVM) and MPI. Starfish [14] provides automatic recovery in MPI-2 programs running on a network of workstations. It uses strict atomic group communication protocols to handle state changes. LAM/MPI [15] and subsequently Open MPI provide C/R on the top of Berkeley Lab Checkpoint/Restart (BLCR) [16], which is a system-level and transparent C/R implementation. BLCR can be used as a standalone system for a single node or as a part of a larger system that runs parallel jobs.

One of the challenges related to C/R is to choose the length of a checkpoint interval. A method to calculate the optimal checkpoint interval has been presented in [17], which minimizes the application runtime considering overhead spent writing the checkpoint files, time required to restart the application in case of failure and the time lost as a result of failure. Unfortunately, C/R based approaches do not scale well in terms of performance with increasing job size due to I/O contention. The work in [18] tries to improve the performance of checkpointing by aggregating checkpoint writes into a buffer pool and interleaves the application progress with file writes.

There are two fundamental types of checkpoint protocols: coordinated and uncoordinated. While coordinated checkpointing involves collaboration between processes to create a system-wide consistent state, uncoordinated checkpointing does not impose any restriction on the checkpoint time. Uncoordinated checkpointing allows better I/O scheduling but is subject to the *domino effect* property [19] discussed next.

In case of failure recovery, the inter-process dependencies imposed by message passing may force some of the processes that did not fail to roll back. Such dependencies can create a chain going all the way back to job start time since causal dependencies are transitive. This phenomenon is called *domino effect*. Methods based on coordinated checkpointing are not affected by the domino effect since a globally consistent state is established at the cost of higher overhead [20]. Log-based methods combine checkpointing with logging of all non-deterministic events, which means all the messages in the worst case. This requires large amounts of overhead and storage to maintain these logs. Considering the size and the induced overhead of these methods, exascale systems [21] may require uncoordinated checkpointing.

Communication Induced Checkpointing [22] provides uncoordinated checkpointing without the domino effect. However, their work shows that the approach provides good performance only under a low communication load and it does not scale well with the larger number of processes. Recent work [23] addresses the same problem for send deterministic MPI applications. A given MPI application is said to be send deterministic, if, for a set of input parameters, the sequence of sent messages, for any process, is the same in any correct execution [24]. Focusing on applications with this property, they provide a protocol that needs to log a subset of the application messages. They also provided an implementation in MPICH2 with experimental results showing low overhead.

Recent work [25] uses a combination of non-blocking and multi-level checkpointing. The main idea involves running agents on extra nodes to asynchronously transfer checkpoint files to the PFS. Using dedicated nodes to create checkpoints gives compute nodes the opportunity to continue their execution.

Generally, the inherent problem with any C/R schemes is the downtime to take the checkpoints and the restart overhead at high failure rates. What makes it worse is the very short MTTF in next-generation supercomputers. As a result, checkpoint intervals become shorter and the system may even spend more time on checkpointing than on actual application execution.

There are also log-based methods with different policies including optimistic logging [26], casual logging (Manetho [27]), and pessimistic logging (MPI/FT [28] and MPICH-V [29]).

B. Redundancy

rMPI [30] is a library that 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 [31] supports partial and full replication and uses PMPI, the MPI performance tool interface, to intercept MPI calls.

Work in [6] determines the best configuration of a combined approach including redundancy and C/R. 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.

Work in [32] investigates the feasibility of process replication for exascale computing. A combination of modeling, empirical and simulation experiments are presented in this work. The authors show that replication outperforms traditional C/R approaches over a wide range of the exascale system design space.

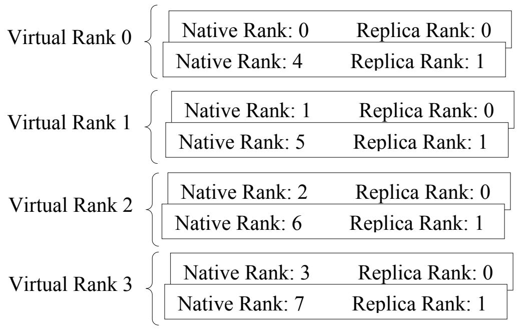

Figure 1: Dual Redundancy for a job with 4 tasks

RedMPI [33] allows the execution of MPI applications in a redundant fashion. Each process is replicated r times for a redundancy degree of r. Point-to-point messages and collective operations are performed within each replica set (original nodes and shadow nodes for $r = 2$, or a second shadow node set for $r = 3$). Figure 1 depicts a dual redundant job and indicates the corresponding terminology for virtual, native and replica ranks, where a virtual rank with its two boxes represents a sphere. An application with originally N processes now creates twice the number of MPI processes ($2N$). The native rank is the rank assigned by `mpirun` within the range $[0...2N - 1]$. The rank API call, `MPI_Comm_Rank`, returns the virtual rank. The MPI processes of each replica sphere, so-called replica ranks, are numbered $[0...r - 1]$.

RedMPI allows SDC detection and correction through comparing the exchanged messages. It can be configured to utilize one of the following protocols: *Basic*, *Message-plus-hash* and *All-to-all*. In the *Basic* protocol, no SDC detection/correction is provided; in *Message-plus-hash*, a message and a hash is sent; and in *All-to-all*, all communication doubles up for $r = 2$, i.e., a send becomes two sends and a receive becomes two receives. *Message-plus-hash* has superior performance over *All-to-all* since the overhead of communication is significantly reduced.

C. Migration

A process-level proactive live migration approach is presented in [34]. 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 the top of Xen virtualization.

[35] 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.

D. *Rejuvenation*

Three rejuvenation techniques for HPC have been introduced in [36]. They also provide an overhead estimation of using rejuvenation right after taking checkpoints. Based on their simulation results, they conclude that rejuvenation does not increase the reliability of HPC systems at all times. Another intuitive result is that the computing time lost in case of C/R with rejuvenation is larger than that when only C/R is used. Their work did not allow for redundancy and process cloning at all. In contrast, we propose to use live process cloning to eliminate the overhead of C/R schemes.

III. Live Process Cloning

Live cloning creates a copy of a running MPI process (a *source* node) on a *destination* node on-the-fly, i.e., while the source continues to run. Figure 2 shows how the system retains dual redundancy in case of a failure. There are at least three nodes directly involved in live process cloning. Suppose two processes, P_1 and P_2, are logically equivalent (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 (*destination*) on-the-fly.

A process is created on N_4 with the same number of threads. While P_2 is performing its normal execution, the memory pages are sent over to the newly created process. This happens in an iterative manner. When we reach a state where few changes in dirty pages remain to be sent, the communication channels are drained. This is necessary to keep the system in a consistent state. After this, P_2 is paused for a very short time and the last dirty pages, linkage information, credentials, etc. are sent over. This way, a clone process is created on N_4. Then, the 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 messaging.

IV. System Model

A. *Stochastic Activity Networks*

A Petri net is a directed bipartite graph consisting of two sets of nodes, namely *places and transitions*. The need for associating exponentially distributed firing times to transitions led to the introduction of stochastic Petri nets (SPN). A generalized form of SPNs called GSPN divides the transitions into timed (exponentially distributed firing times) and immediate transitions (zero firing times). Stochastic Activity Networks (SANs) are an extension of stochastic Petri nets with instantaneous and timed activities, reward variables and supporting composed models. These properties have made SANs a powerful tool for modeling complex systems specially for reliability and availability analysis.

SAN models consist of *places, activities, input gates, output gates, arcs and tokens*. Each place may have a number of tokens initially, which is shown inside the place as a number. The activities may have different distributions, out of which *exponential* is of interest, here. There are also input and output gates, which may be attached to an activity and one or more places. Input gates have a predicate that, when evaluating to true, trigger activities and a rule for state transitioning when the related activity completes. Output gates define the marking changes that occur when an activity completes and attach to the places that changes in markings.

Input arcs connect places to activities while output arcs connect activities to places. Assume an activity with an input arc from P_1 and an output arc to P_2. This activity is enabled if the number of tokens in each input place is at least equal to the aggregate of the input arcs and the predicate of all input gates are true. When this activity completes, it removes one token from P_1 and deposits one token to P_2.

B. *Redundancy Modeling*

Let us assume that a system consists of N computing nodes with redundancy degree r, i.e., there are $N \times r$ total nodes in the system. Individual nodes could fail with an exponential failure rate. The system is said to have failed when there is at least one sphere in which all the r nodes have failed. So to model this behavior, we need to keep track of the number of spheres with x failed nodes, where $x \in [1, r]$.

Figure 3 shows the SAN model for a redundancy degree of r. In the beginning, there are $N \times r$ tokens in P_{up}. The figure shows the working state of all computing nodes. There is an activity T_{fail} that models the failure of nodes with an exponential rate. As time passes, nodes start to fail. When the activity T_{fail} completes, a token will be removed from P_{up} and, with probability c_1, will be deposited to the

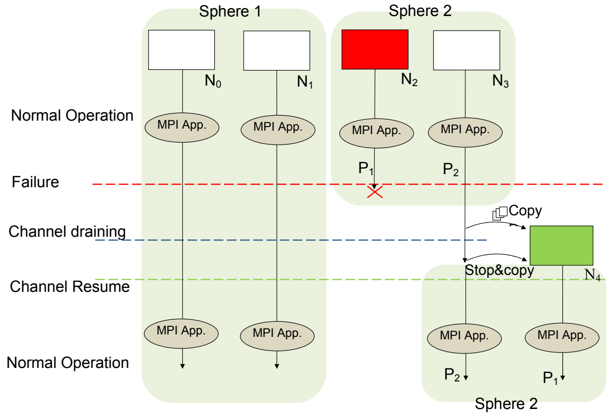

Figure 2: Dual Redundancy and Live Cloning

$P_{1st-fail}$, with probability c_2, it will be put in $P_{second-fail}$, etc. C_i is the probability of a node failure in a sphere with already $i-1$ failed nodes. In our model, we need to keep track of number of spheres with x failed nodes where $x \in [0, r]$. P_{x-fail} represents the number of spheres with x failed nodes. As soon as a sphere with r failed nodes exists, the whole system has failed. We know from probability theory that $\sum_{i=1}^{r} c_i = 1$. Computing c_1 directly is difficult, but its complement is easy to express. The probability, c_i, that the failed node belongs to a specific category for $i \in [2, r]$ is:

$$\begin{aligned} c_2 &= \frac{(r-1) \times Mark(P_{1-fail})}{Mark(P_{up})}. \\ c_3 &= \frac{(r-2) \times Mark(P_{2-fail})}{Mark(P_{up})}. \\ &\ldots \\ c_r &= \frac{(1) \times Mark(P_{(r-1)\_fail})}{Mark(P_{up})}. \end{aligned} \tag{1}$$

We use the notation $Mark(P)$ to denote the number of tokens in place P. Now, we can determine c_1 as $c_1 = 1 - \sum_{i=2}^{r} c_i$. A function is associated with each output gate. This function is executed after the activity completes and the related case is selected. For $i \in [1, r-1]$, the function for OG_i is shown in Table 1. For the last case, we have a sphere with r failed nodes, so we generate one token in P_{r-fail}. As a result, the whole system has failed, so we remove the tokens from all the states and generate one token in $P_{sysfail}$.

Figure 4 depicts the model for dual redundancy with cloning. In the case of live process cloning, a new activity is added to perform the clone functionality when a node fails. The system starts with all nodes ($N \times 2$) in working state (P_{up}) and 0 tokens in other places (P_{1-fail}, P_{2-fail}, $P_{sys-fail}$). After the first node failure (T_{fail} is enabled), a token is removed from (P_{up}). Where this token is transferred to depends on the current values of c_1 and c_2 (i.e., $c_2 = 0$ and $c_1 = 1$). In this case, it will be transferred to P_{1-fail}.

Table I: Gate Functions for Redundancy Model

Gate name	Function
$OG_i \quad i \in [1, r-2]$	$P_{i-fail} = P_{i-fail} - 1;$ $P_{(i+1)-fail} = P_{(i+1)-fail} + 1;$
OG_{r-1}	$P_{i-fail} = 0; \; i \in [1, r-1]$ $P_{up} = 0;$ $P_{r-fail} = 1;$ $P_{sys-fail} = 1;$

This state indicates that we have one sphere with one failed node (one token in P_{1-fail}). A failed node could be recovered by cloning processes from its healthy replica to a backup nodev(T_{clone}). Let us suppose another node fails. c_1 and c_2 are recalculated based on the current marking of the model. Using formula 1, we get $c_2 = \frac{1}{(N \times 2-1)}$ and $c_1 = 1 - \frac{1}{(N \times 2-1)}$. This second failed node belongs to the sphere with the first failed node with probability c_2, and it belongs to the other spheres with probability c_1. In the first case, a token is transferred to P_{2-fail} and $P_{sys-fail}$ via gate function OG_1 and results in a system failure. In the second case, a token is transferred to P_{1-fail} (now becoming $Mark(P_{1-fail}) = 2$). This means that there are 2 spheres, each with one failed node. Models for dual and triple redundancy without cloning can be derived from Figure 3.

Figure 5 shows the results for reliability modeling of a system with $N = 200K$ nodes, each with a $MTTF$ of 5 years. The time to clone all processes on one node is assumed to be two minutes. Reliability of dual redundancy without cloning decreases to 75% after 400 hours for a long running job. The two curves, dual redundancy with cloning and triple redundancy without redundancy, overlap and provide high reliability (very close to 100%) during the entire period. This figure shows that $r = 2$ with cloning has a reliability as good as $r = 3$ without cloning. This is an interesting result showing that live process cloning can save system cost while providing the same reliability level of a system with larger redundancy r.

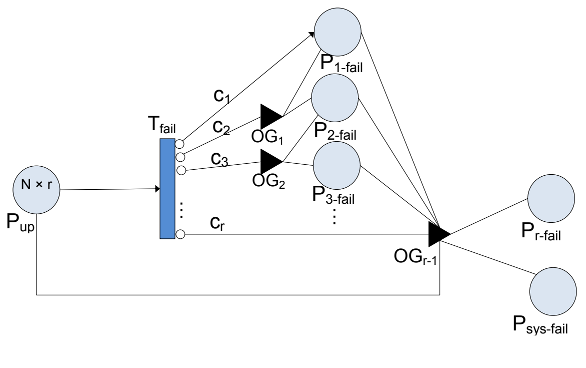

Figure 3: SAN Model for a Redundancy Degree of r

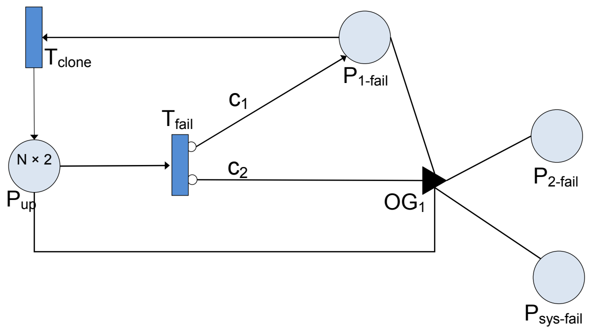

Figure 4: Redundancy Models for r=2 with Clone

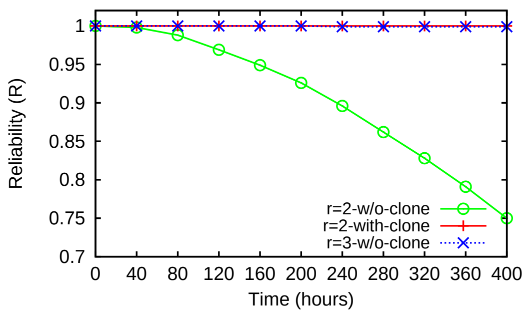

Figure 5: Reliability for $N = 200K$

C. *Node Failure Analysis*

Assume that every node has an identical exponential failure rate (λ) and that the reliability of the node is $R(t) = e^{-\lambda t}$. Consider all nodes ($N \times r$) in N spheres, where each sphere contains nodes of the same replica, reliability and MTTI of a sphere is:

$$R_{sphere}(t) = 1 - (1 - R(t))^r$$

$$\begin{aligned} MTTI_{sphere} &= (\int_0^\infty R_{sphere}(t)\, dt) \\ &= (\int_0^\infty 1 - (1 - R(t))^r\, dt) \end{aligned}$$

System reliability and MTTI [37] can be calculated as:

$$R_{sys}(t) = (R_{sphere}(t))^N$$

$$MTTI = \sum_{i=1}^{N} \sum_{j=1}^{i.r} \left(\frac{\binom{N}{i}\binom{i.r}{j}(-1)^{i+j}}{\lambda j} \right) \tag{2}$$

We want to investigate the time interval to a system failure for dual redundancy. We also want to determine the effect of multiple concurrent node failure and maximum simultaneous failures tolerable by our system. Please note that our policy is to clone from a replica to a spare node as soon as a node failure is detected. Thus, as long as there are one or more failed nodes in the system, cloning is in progress. From now on we focus on the case of dual redundancy $r = 2$.

1) First Node Failure: Suppose at time t the first node failure occurs. Thus, there are $N - 1$ spheres with $r = 2$ nodes and 1 sphere with $r = 1$. Following shows the corresponding system reliability:

$$\begin{aligned} R_{sys}(t) &= (R_{sphere}(t))^{N-1} \times (R(t)) \\ &= (1 - (1 - e^{-\lambda t})^2)^{N-1} \times (e^{-\lambda t}). \end{aligned} \tag{3}$$

The MTTI could be computed as following:

$$\begin{aligned} &= \int_0^\infty (R_{sys}(t)dt) \\ &= \int_0^\infty ((1 - (1 - e^{-\lambda t})^2)^{N-1}(e^{-\lambda t})dt) \\ &= \int_0^\infty (\sum_{i=0}^{N-1} \binom{N-1}{i} (-1)^i (1 - e^{-\lambda t})^{2i} (e^{-\lambda t})dt) \\ &= \sum_{i=0}^{N-1} \binom{N-1}{i} (-1)^i \left(\int_0^\infty (1 - e^{-\lambda t})^{2i} e^{-\lambda t} dt \right) \\ &= \sum_{i=0}^{N-1} \binom{N-1}{i} (-1)^i \left(\frac{1}{\lambda(2i+1)} \right) \end{aligned}$$

Thus:

$$MTTI = \sum_{i=0}^{N-1} \binom{N-1}{i} \left(\frac{(-1)^i}{\lambda(2i+1)} \right) \tag{4}$$

Let us assume that the system has an exponential failure distribution. We can then approximate the system failure rate by

$$R_{sys}(t) = (e^{-\lambda_{sphere} t})^{N-1} \times (e^{-\lambda t})$$

$$\lambda_{sys} = (N-1)\lambda_{sphere} + \lambda. \tag{5}$$

Comparing 5 to the old system failure rate ($N\lambda_{sphere}$), we see that the system failure rate is increased by $\lambda - \lambda_{sphere}$. Considering the very short time to clone(a few minutes) the change in the failure rate is still negligible.

2) Multiple Concurrent Node Failures: Suppose at time t, x simultaneous yet independent node failures happen ($1 < x < N$). Assume the system is still functional. Thus, failures happened in different spheres. Then there are $N - x$ spheres with $r = 2$ nodes and x spheres with $r = 1$. Thus the system reliability:

$$\begin{aligned} R_{sys}(t) &= (R_{sphere}(t))^{N-x} \times (R(t))^x \\ &= (1 - (1 - e^{-\lambda t})^2)^{N-x} \times (e^{-\lambda t})^x \end{aligned}$$

The MTTI of the whole application could be calculated as:

$$= \int_0^\infty (R_{sys}(t)dt)$$

$$= \int_0^\infty ((1-(1-e^{-\lambda t})^2)^{N-x}(e^{-x\lambda t})dt)$$

$$= \int_0^\infty (\sum_{i=0}^{N-x} \binom{N-x}{i} (-1)^i (1-e^{-\lambda t})^{2i} (e^{-x\lambda t})dt)$$

$$= \int_0^\infty \sum_{i=0}^{N-x} \binom{N-x}{i} (-1)^i \left(\sum_{j=0}^{2i} \binom{2i}{j} (-1)^j e^{-\lambda j t} \right) e^{-x\lambda t} dt$$

$$= \sum_{i=0}^{N-x} \binom{N-x}{i} (-1)^i \left(\sum_{j=0}^{2i} \binom{2i}{j} (-1)^j \int_0^\infty e^{-\lambda(j+x)t} dt \right)$$

$$= \sum_{i=0}^{N-x} \binom{N-x}{i} (-1)^i \left(\sum_{j=0}^{2i} \frac{\binom{2i}{j} (-1)^j}{\lambda(j+x)} \right)$$

$$= \sum_{i=0}^{N-x} \sum_{j=0}^{2i} \left(\frac{\binom{N-x}{i} \binom{2i}{j} (-1)^{i+j}}{\lambda(j+x)} \right)$$

$$MTTI = \sum_{i=0}^{N-x} \sum_{j=0}^{2i} \left(\frac{\binom{N-x}{i} \binom{2i}{j} (-1)^{i+j}}{\lambda(j+x)} \right) \tag{6}$$

Again, let us assume an exponential failure distribution for the system. We can then approximate the failure rate by

$$R_{sys}(t) = (e^{-\lambda_{sphere} t})^{N-x} \times (e^{-\lambda t})^x$$

$$\lambda_{sys} = (N-x)\lambda_{sphere} + x\lambda.$$

In this situation, the system failure rate is increased to:

$$\lambda_{sys} = \lambda_{old} + x(\lambda - \lambda_{sphere})$$

We can recover from x simultaneous failure via cloning if

$$x \times T_c < \frac{MTTF}{(2 \times N) - x} \tag{7}$$

where T_c is the time to clone and the right-hand side is the average time between two consecutive node failures. Cloning to recreate the processes that are co-located on a single node could be carried out in parallel. We assume that x node failures overlap by just one cycle. Thus, $x \times T_c$ is a good approximation of the time to clone x nodes. Lets assume $TTC(x) = x \times T_c$ as the time to clone for x concurrent failures.

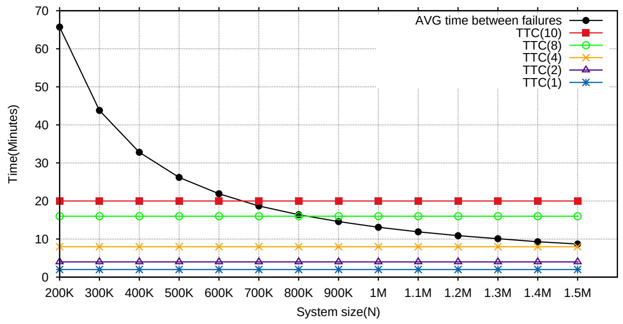

Figure 6: Average Time between Failures vs. $TTC(x)$

Figure 6 shows the average number of required spare nodes with non-repairable nodes. As long as the time to clone for x concurrent failures ($TTC(x)$) is less than the average time between failures, we can guarantee that cloning completes and we can recover from failures. Thus, we are interested in the area where $TTC(x)$ is less than the average time between failures. Let us consider $TTC(2)$. The graph indicates that one can recover from failures for a system exceeding 1.5M nodes. Ten concurrent failures ($TTC(10)$) can be tolerated for systems up to $N = 600k$ nodes, four concurrent failures are tolerated up to $N = 1M$ nodes, and four concurrent files up to $N = 1.5M$ nodes.

D. *Average Number of Required Spare Nodes*

This section analyzes the effect of cloning on the average number of required spare nodes to still complete a job when using live process cloning. Let α be the communication-to-computation ratio of an application. Equation 8 shows the execution time with redundancy [6]:

$$T_{Red} = (1-\alpha)T + \alpha Tr \tag{8}$$

Suppose there is a job that requires T hours to run on N machines to complete without any failures. This is called plain execution time. Assume each node has a $MTTF$ of 50 years. On average, every $MTTF/(r \times N)$ a node fails in the system. Further, assume $r = 2$, $T = 200$ hours, and live cloning is used in case of a failure. The time to complete the job with $\alpha = 0.2$, 0.4 and 0.6 using Equation 8 is 240, 280 and 320 hours, respectively. Figure 7 shows the number of spare nodes needed for successful job completion for different values of N ranging from 110 ($N = 100k$, $\alpha = 0.2$) to 1462 ($N = 1M$, $\alpha = 0.6$). In this case, the MTTR of a node is not taken into account.

If the mean time to repair of a computing node is assumed to be 20 hours, the average number of spare nodes for an application with any α is shown in Figure 8. As we can see, the average number of spare nodes is ranging from 10 to 92, which is only 0.01% of total number of nodes (N). Assuming that nodes are repairable, the average number of required spare nodes turns out to be independent of the α

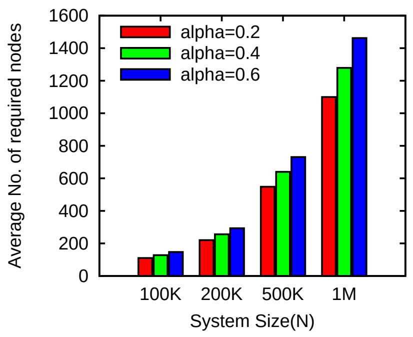

Figure 7: Avg. # Required Spare Nodes (Without repair)

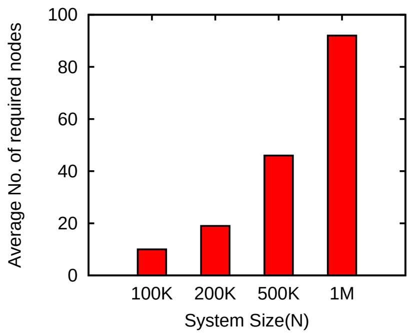

Figure 8: Avg. # of Required Spare Nodes (With repair)

value. For example, consider N=100k at MTTR=20h. When α = 0.2, the job takes 240 hours, and we have 240/20 = 12 repair intervals. Similarly, for α= 0.4, there are 280/20 = 14 repair interval, and for α=0.6, there are 320/20=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 110/12, 128/14 and 147/16 for α=0.2, 0.4 and 0.6, respectively. In all three cases, ten spare nodes are required. The same holds for 200K, 500K and 1M, meaning that results are independent of α.

V. Conclusion

In this paper, we studied the combination of redundancy and live process cloning to increase reliability of high-performance computing systems at large scales. We discussed different approaches that have been proposed to mitigate node failures in HPC. We presented a reliability model for redundant computing based on SAN. The modeling results show that dual redundancy with clone is as good as triple redundancy with regards to reliability. We analyzed the effect of multiple simultaneous node failures on the system MTTI. We also investigated the average number of required spare nodes and showed that it is only 0.01% of total number of nodes.

Acknowledgement

This work was funded in part by NSF grants 0958311, 0937908 and 1058779.

References

[1] 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.

[2] 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

[3] 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.

[4] 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.

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

[6] 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.

[7] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[8] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Supercomputing*, nov 2012.

[9] C. Clark, K. Fraser, S. Hand, J. Hansem, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *2nd Symposium on Networked Systems Design and Implementation*, May 2005.

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

[11] C. Wang and F. Mueller and C. Engelmann and S. Scott, "Proactive Process-Level Live Migration and Back Migration in HPC Environments," *Journal of Parallel Distributed Computing*, vol. 72, no. 2, pp. 254–267, Feb. 2012.

[12] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," in *Proc. of 10th International Parallel Processing Symposium (IPPS 96)*. IEEE CS Press, 1996, pp. 526–531.

[13] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," University of Wisconsin - Madison Computer Sciences Department, Tech. Rep. UW-CS-TR-1346, April 1997.

[14] A. Agbaria and R. Friedman, "Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations," in *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 167–176.

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

[17] 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, Feb. 2006.

[18] X. Ouyang, K. Gopalakrishnan, D. K. Panda, and et al., " Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture," 2009.

[19] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 437–449.

[20] G. Cao and M. Singhal, "On Coordinated Checkpointing in Distributed Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 12, pp. 1213–1225, Dec. 1998.

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

[22] L. Alvisi, S. Rao, S. A. Husain, A. de Mel, and E. Elnozahy, "An Analysis of Communication-Induced Checkpointing," in *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, ser. FTCS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 242–249.

[23] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 989–1000.

[24] F. Cappello, A. Guermouche, and M. Snir, "On Communication Determinism in Parallel HPC Applications," in *Proceedings of the 19th International Conference on Computer Communications and Networks, IEEE ICCCN 2010, Switzerland, August 2-5, 2010*. IEEE, 2010, pp. 1–8.

[25] K. Sato, A. Moody, K. Mohror, T. Gamlin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, "Design and Modeling of a Non-Blocking Checkpointing System," in *Proceedings of the 2012 IEEE conference on Supercomputing*, ser. SC '12, 2012.

[26] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," Department of Computer Science, Rice University, Houston, Texas, Technical report, 1987.

[27] E. Elnozahy and W. Zwaenepoel, "Replicated distributed processes in Manetho," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, jul 1992, pp. 18–27.

[28] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, and M. Apte, "MPI/FTTM: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, pp. 26–33, 2001.

[29] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: toward a scalable fault tolerant MPI for volatile nodes," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–18.

[30] 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.

[31] 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.

[32] 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. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063443

[33] 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.

[34] 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.

[35] 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.

[36] N. Naksinehaboon, N. Taerat, C. Leangsuksun, C. F. Chandler, and S. L. Scott, "Benefits of Software Rejuvenation on HPC Systems," in *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, ser. ISPA '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 499–506.

[37] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, "Combining Process Replication and Checkpointing for Resilience on Exascale Systems," INRIA, Rapport de recherche RR-7951, May 2012. [Online]. Available: http://hal.inria.fr/hal-00697180