Fault Tolerance on Large Scale Systems using Adaptive Process Replication

Cijo George and Sathish Vadhiyar

Abstract—Exascale systems of the future are predicted to have mean time between failures (MTBF) of less than one hour. At such low MTBFs, employing periodic checkpointing alone will result in low efficiency because of the high number of application failures resulting in large amount of lost work due to rollbacks. In such scenarios, it is highly necessary to have proactive fault tolerance mechanisms that can help avoid significant number of failures. In this work, we have developed a mechanism for proactive fault tolerance using partial replication of a set of application processes. Our fault tolerance framework adaptively changes the set of replicated processes periodically based on failure predictions to avoid failures. We have developed an MPI prototype implementation, PAREP-MPI that allows changing the replica set. We have shown that our strategy involving adaptive process replication significantly outperforms existing mechanisms providing up to 20 percent improvement in application efficiency even for exascale systems.

Index Terms—Fault tolerance, process replication, exascale systems

1 Introduction

With the development of high performance systems with massive number of processors [1] and long running scalable scientific applications that can use large number of processors for executions, the MTBF of the processors used for a single application execution has tremendously decreased [2]. The current petascale systems are reported to have MTBFs of less than 10 hours to a few days and future exascale systems are anticipated to have MTBFs of less than an hour [3], [4]. However, long running scientific simulations including climate modeling and molecular dynamics can have execution times of the order of weeks to even months, typically segmented into multiple job executions of few days execution time each and with checkpointing at the end of each execution to continue computations. This segmentation is done to enable time sharing of the computational resources by multiple different jobs on the HPC systems. It is highly imperative to develop efficient fault tolerance strategies to sustain executions of these long-running real scientific applications on future large scale systems like exascale systems.

Periodic checkpointing [5], [6] is the most popular and long-established strategy for fault tolerance in parallel systems and applications. In periodic checkpointing systems, the application is periodically made to store its state in anticipation of failures. Periodic checkpointing allows the application to rollback to a recently checkpointed state on the occurrence of failures. With lower platform MTBFs on systems with large number of processors, the number of failures increases resulting in a significant increase in the amount of work lost due to rollbacks. Reduced checkpoint intervals on systems with low platform MTBF will also lead to more number of checkpoints contributing to the overall reduction in application efficiency. Recent studies [7], [8], [9] have shown that periodic checkpointing with the commonly available PFS-based, fully coordinated, application-agnostic checkpointing approaches results in application efficiencies of only 20-30 percent on peta and exa scale systems! Thus, traditional periodic checkpointing is not a viable fault tolerance option for large scale systems. Hence, it is highly necessary to employ proactive fault tolerance mechanisms that can help avoid significant number of failures.

Process replication is being increasingly considered for fault tolerance for current and future large scale systems [10], [11]. In process replication, the state of a process is replicated in another process called *replica* or *shadow* process, such that even if one of them fails, the application can continue execution without interruption. Process replication is a highly favorable option for fault tolerance on large scale systems with large number of failures, since unlike periodic checkpointing, node failures will result in application failures only if a replica is not present or if the replica also fails simultaneously, thus resulting in reduced number of failures. Replication also leads to increased intervals between checkpointing, thereby reducing checkpointing overheads. All these advantages of replication result in significantly higher application efficiency than periodic checkpointing for peta and exascale systems.

In spite of these benefits, dual redundancy, in which processes in all sockets (or nodes) in a system are replicated[1], can achieve only less than 50 percent application efficiency, since only half the total number of nodes is used for actual application execution. This can result in huge amount of resource wastage on large scale systems. To verify this, we performed simulation experiments involving execution on

- *C. George is with NetApp Advanced Technology Group, Bangalore, India. E-mail: cijo@netapp.com.*
- *S. Vadhiyar is with the Supercomputer Education and Research Centre of Indian Institute of Science, Bangalore, India. E-mail: vss@serc.iisc.in.*

Manuscript received 26 Jan. 2013; revised 28 Aug. 2014; accepted 10 Sept. 2014. Date of publication 25 Sept. 2014; date of current version 10 July 2015. Recommended for acceptance by M. Parashar.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2360536

1. We use process replication and node replication interchangeably.

200,000 nodes/processors with dual redundancy for one week with a node MTBF of 25 years and Weibull distribution of failures. We found that the total number of node failures before an application failure, due to both a process and its replica failing, is only between 50 to 400. Thus only a maximum of 400 of the 200,000 nodes have to be replicated. However, in anticipation of this very small number of failures, the dual redundancy scheme replicates 100,000 nodes, thereby utilizing only 100,000 nodes for application execution! While about 199,600 nodes or more than 99 percent of the total number of nodes could have been used for application execution, the dual redundancy scheme utilizes only 50 percent, resulting in large resource wastage. The primary reason for this "safe" approach of dual redundancy is due to the lack of knowledge of the specific 400 nodes that may fail during execution.

In this work, we have developed a mechanism for proactive fault tolerance using partial replication of a set of application processes. Our framework starts an application with a fixed small number of processes replicated. At regular intervals of time, our framework adaptively changes the set of replicated processes based on failure predictions such that all failure prone nodes have healthy replica, thus attempting to avoid failures. Our adaptive strategy leverages on the advantages of process replication while keeping the number of replica nodes to a minimum, thereby minimizing resource wastage.

Our fault tolerance framework relies on a cost efficient mechanism to adaptively change the set of replicated processes without having to checkpoint/restart the application. We have developed an MPI prototype implementation, PaRep-MPI, that makes this possible by providing the ability to transparently copy a process state from a process in a local node to a process in a remote node and modify the remote process to act as the replica of the local process. Simulations using synthetic failure traces for exponential and Weibull distributions have shown that our fault tolerance framework involving adaptive partial process replication significantly outperforms existing mechanisms for large and very large scale systems providing up to 20 percent improvement in application efficiency even for exascale systems.

2 Related Work

2.1 Checkpointing and Process Migration

Most of the fault tolerance mechanisms in literature are based on checkpointing [5], [6]. Proactive process migration [12] is a failure avoidance mechanism in which processes in failure prone nodes are migrated to healthy spare nodes periodically to avoid application failures. There are also frameworks that combine different fault tolerance mechanisms and dynamically select the best strategy at regular intervals of time [7], [9].

Some efforts use system-level checkpointing to support fault tolerance for HPC applications. For example, the Berkeley Lab Checkpoint/Restart (BLCR) [13] is a system-level checkpoint/restart mechanism for Linux clusters. BLCR is intended for fault tolerance based on events generated by system sensors such as rising temperature, and for batch scheduling. BLCR is implemented as a kernel module, and MPI implementations can provide fault tolerance by integrating with BLCR via a callback interface to cooperate among processes for taking and restoring from a checkpoint. In this paper, we target application-level checkpointing and replication. Some efforts have focused on mitigating the cost of checkpointing. The Scalable Checkpoint Restart (SCR) library [14] employs multi-level checkpointing where the state of the entire parallel system is stored at the highest level using slow, but reliable parallel file systems, while node and cross-node checkpoints are taken at the lowest level using fast, and less reliable in-system storage. Recent work [15] has used MRNet tree-based overlay network library [16] for asynchronous transfers of checkpoints to the parallel file system to increase the computations-checkpointing overlap.

Bosilca et al. [6] had presented a unified model for evaluating different rollback recovery protocols including coordinated checkpointing, uncoordinated checkpointing using message logging, and hierarchical checkpointing. They have used this model to evaluate different protocols and conclude that with appropriate provision in I/O resources and technology, periodic checkpointing can outperform dual redundancy.

2.2 Failure Predictions

To help a runtime system make proactive fault tolerance decisions like process migration, techniques have been developed to predict failures. While some prediction mechanisms are based on failure history of the systems [17], one of the approaches that is relevant to our work on prediction-based replication is the use of symptom or health monitoring of the system to predict failures [18], [19]. The system attributes that are monitored for failure prediction include disk temperature, memory leaks, used swap space, work accomplished since last restart, CPU utilization, idle time, network IO event logs, and hardware sensor measurements such as environment temperature and power supply voltage.

In a recent work, Gainaru et al. [20] have developed a hybrid prediction technique embedded into their ELSA (Event Log Signal Analyzer) toolkit. Their work monitors various events generated by the system including memory failure, node crashes, cache errors and node card failures and uses a hybrid technique that combines signal analysis and data mining for prediction. The signal analysis component characterizes system's normal behavior and performs anomaly detection based on the events, while the data mining component performs advanced correlations between the events and the failures. Results with Bluegene/L system show that their hybrid prediction approach provides about 46 percent recall and 92 percent precision.

2.3 Replication

Process replication [10], [11], [21], [22], [23] is another fault tolerance mechanism, which is being considered for future large scale systems. In process replication, application processes are replicated so that even if a process fails, the application is not interrupted, due to the presence of a replica. VolpexMPI [21] is an MPI implementation that replicates certain processes to provide fault tolerance and proactive migration for MPI applications executing on volatile systems, where idle PCs can be used for "guest" MPI processes. The library follows sender-based message-logging with

replication for fault tolerance, while our work considers checkpointing with replication. VolpexMPI is designed for applications with minimal communication requirements on small-scale systems (e.g., institutional LANs), while our study is for general HPC scientific applications on peta and exa scale systems.

Ferreira et al. [10] have shown that a process replication strategy with dual hardware redundancy, in which all processes in a system are replicated, can significantly increase the mean time to interrupt (MTTI) of an application. Their work has also developed *r*MPI, a user-level MPI library that enables replication for MPI applications. They have shown using simulations with an exponential failure distribution that replication outperforms traditional periodic checkpointing for socket counts greater than $20{,}000$ and is a viable fault tolerance technique for socket counts and I/O bandwidths anticipated for future exascale systems. However, with Weibull failure distribution both periodic checkpointing and dual redundancy were shown to give low application efficiency. Our adaptive replication strategy provides higher application efficiency than dual redundancy even for Weibull distributions of failures.

Elliott et al. [11] have developed a fault tolerance strategy that combines periodic checkpointing with partial redundancy. The study shows the effects of varying degree of redundancy on application execution time for different kinds of systems. Their work has built RedMPI, a user-level MPI library similar to *r*MPI that enables redundant computing for MPI applications, with support for both partial and dual redundancy. They have shown that partial redundancy can be beneficial even for medium scale systems with $4{,}000$ to $25{,}000$ processors with a degree of redundancy between 1.5 and 2, where a degree of 2 corresponds to dual redundancy. While this work deals with finding an optimal degree of replication that gives maximum performance, the nodes that are replicated are arbitrarily chosen. Our adaptive replication strategy dynamically changes the set of replicated processes based on failure predictions to avoid the nodes that are predicted to fail.

In a recent work [23], RedMPI was extended to use dual and triple redundancy to detect and correct silent data corruption (SDC) errors. The error detection happens at the MPI communication layer where the receiver replicas compare and verify received messages for data corruption. A receiver replica receives a full message from a sender replica and a hash of the message from another sender replica to verify correctness. Stearly et al. [24] have also recently developed a model for studying job interrupt times on systems of arbitrary replication degree and node failure distribution.

There have been efforts using simulations to evaluate different fault tolerance mechanisms and policies. The work by Tikotekar et al. [25] built a simulator framework to compare reactive fault tolerance using checkpointing, proactive process migration, and a strategy using both checkpointing and process migration. They have performed simulations using LLNL logs for up to 512 nodes. They conclude that a fault tolerance policy that combines proactive migration with reactive checkpointing is promising for certain systems. Our work evaluates the benefits of partial replication with checkpointing using failure predictions. We conduct studies for large-scale systems including peta and exa scale systems.

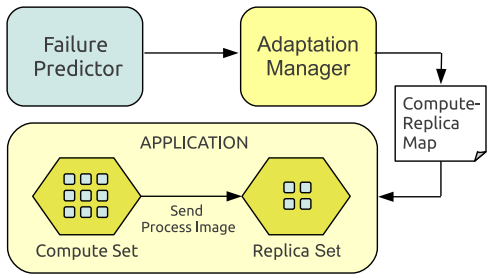


Fig. 1. Adaptive process replication framework.

3 Adaptive Process Replication Framework

We assume the presence of a *failure predictor* that can estimate the node failures in a system for a given time window. Specifically, the predictor has to periodically estimate the list of nodes that are expected to fail in the next time interval. *Precision* of such a predictor is defined as the ratio of the number of correct predictions to the total number of predictions made. *Recall* of the predictor is defined as the ratio of the number of correct predictions to the total number of failures. Higher the values of precision and recall, the better the predictor. A node is considered as *failure prone* if it is predicted to fail, and considered as *healthy* if it is not predicted to fail, in the next interval.

An application, which is allotted a fixed N_{tot} number of nodes, starts its execution on a fixed N_{cmp} number of compute nodes, using the remaining N_{rep} $(= N_{tot} - N_{cmp})$ number of nodes as *replica* nodes. Each application process either belongs to the *compute set* or to the *replica set*. We assume partial replication ($N_{rep} < N_{cmp}$), i.e., not all application processes will have an associated replica process. A *compute-replica map* is used by the processes to find the mapping between compute nodes and replica nodes. This can be a file or a global data structure accessible by all the processors in the system.

The overall working of our adaptive replication framework is illustrated in Fig. 1. The goal of the framework is to ensure that at any given point of time, all failure prone nodes in the system have healthy replica nodes associated with them. At regular intervals of time, an *adaptation manager* determines the compute-replica map and selects a set of nodes for allocation of healthy replica for the next interval, based on the node failure predictions given by the failure predictor. A node that is predicted to fail is selected for allocation of a replica if it is a compute node and satisfies one of the following conditions.

- The node does not have a replica allotted to it.
- The node has a replica allotted to it, but the replica is also failure prone in the next interval.

For each node in the list of selected nodes for healthy replica node allocation, the adaptation manager then takes the following steps.

- Select a random healthy replica node, which is not currently associated with a failure prone compute node.
- Modify the compute-replica map, such that the selected replica node is now mapped to the given failure prone compute node.

Each of the application processes checks the compute-replica map regularly. If the map is modified, the processes take the following steps.

- If the process belongs to the compute set and finds that it has been allotted a new replica node, it sends its process image to the new replica node.
- If the process belongs to the replica set and finds that it has been allotted a new compute node, it receives the process image of the corresponding compute node and initiates a process replacement. Once the process replacement is completed, it acts as the replica of the newly allotted compute node and continues execution.

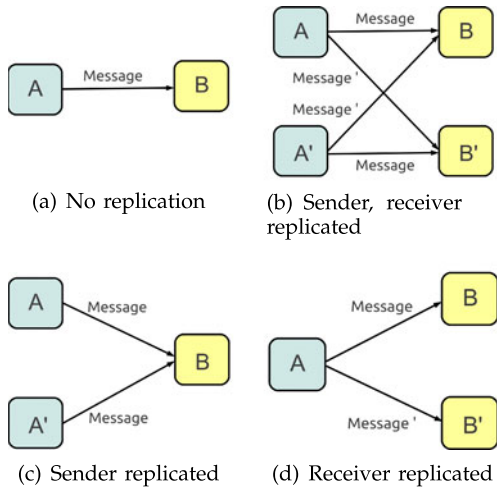
Similar to MR-MPI [22], our framework ensures that the computational process and its replica are located on different nodes to prevent application failure due to a node failure. Our framework can avoid all the node failures which are correctly predicted if the number of healthy nodes in the replica set at any given point of time is at least equal to the number of nodes predicted to fail in a given interval. While our framework can avoid a significant number of predicted node failures by using replication, it tolerates unpredicted node failures in the system by using *periodic checkpointing*. The checkpointing interval is calculated based on the interval between unpredicted failures, and using this interval in Daly's higher order checkpoint/restart model [26]. To determine the time interval between unpredicted failures, recall of the failure prediction is used. The lower the recall of the predictor, the higher will be the number of unpredicted failures. Given the recall, the time interval between such unpredicted failures can be estimated as $\frac{platform\ MTBF}{1-recall}$.

4 PaRep-MPI: A Prototype Implementation for Adaptive Process Replication

4.1 Basic Design

PaRep-MPI is our MPI implementation that supports partial process replication with adaptive changing of the set of replicated processes for MPI applications. It acts as a profiling layer between the application and the MPI library and allows users to specify a replication degree less than or equal to 2, where a degree of 2 corresponds to dual redundancy. While MPI implementations like VolpexMPI [21] implements MPI functions from scratch using a specialized socket library, our PaRep-MPI library adopts the approach of *r*MPI [10] and RedMPI [23] libraries by using MPI profiling layer (PMPI) to intercept MPI function calls and wrap the MPI calls with additional functionalities. While this profiling layer approach results in lesser optimization opportunities when compared to from-the-scratch approach, it has the advantage of providing portability over different MPI implementations with different communication protocols and parallel architectures.

Processes in MPI_COMM_WORLD are divided into two groups and corresponding communicators, one for the processes in the compute set and the other for the processes in the replica set. All subsequent MPI library calls made by the application are intercepted by PaRep-MPI's profiling layer, and replica processes are handled transparent to the application. For example, redundancy is handled by the PaRep-MPI layer for a $Send()$ call from the application by posting a redundant $Send()$ call to the replica process thereby sending the message to both the compute and the replica processes. Our current implementation supports the MPI functions $Comm\_rank\ ()$, $Comm\_size\ ()$, $Send\ ()$, $Isend\ ()$, $Recv\ ()$, $Irecv\ ()$, $Reduce\ ()$, $Allreduce\ ()$, $Alltoall\ ()$, $Alltoallv\ ()$ and $Bcast\ ()$.



(a) No replication (b) Sender, receiver replicated

(c) Sender replicated (d) Receiver replicated

Fig. 2. Communication replication in PaRep-MPI.

4.2 Communication Replication

In PaRep-MPI, the sender process and its replica (if one exists) sends the message to the receiver process and also its replica (if one exists). This translates to the following communication patterns for four replication scenarios.

1) If sender process A and receiver process B are not replicated, then A simply sends the message to B as shown in Fig. 2a.
2) If sender process A has a replica A' and receiver process B has a replica B', then both A and A' sends a copy of the message to both B and B', resulting in both B and B' having a redundant copy of the message. This is shown is Fig. 2b.
3) If sender process A has a replica A' and receiver process B is not replicated, then both A and A' sends the message to B, resulting in a redundant copy of the message at B. This is shown in Fig. 2c.
4) If sender process A is not replicated and receiver process B has a replica B', then A sends a copy of the message to both B and B' as shown in Fig. 2d.

Message redundancy at the receiver end that occurs in two of the above scenarios can be potentially used to detect and correct soft errors in the application by comparing redundant messages from replica processes. Fiala et al. have recently studied this and have extended RedMPI to detect and correct SDC [23]. In our current prototype implementation of PaRep-MPI, we assume that the message from a sender and its replica are the same and hence one of the messages is simply discarded and the other is used for computation.

4.3 Adaptive Replica Change

The fault tolerance framework explained in Section 3 relies on the existence of a reliable, cost efficient mechanism for adaptively changing the set of replicated processes. PaRep-MPI uses a strategy which involves replacing process image

to change processes in the replicated set. A process image is basically the information in the address space of the process that includes the state of the process and its data structures. Process image replacement is used in many process migration systems [12], [27] in which an entire process image of a process is migrated to another process or a machine and continued execution. This complete process image replacement is not applicable in PAREP-MPI since replica processes are MPI processes and are part of the MPI environment (communicator) of the parallel application. Processes in an MPI environment will have their own identities which are defined by the MPI communicator structures. A process replacement strategy to change the set of replicated processes should make sure that these identities are retained by the processes, while the state of a replica process and its data structures should change to reflect that of its newly allotted compute process. PAREP-MPI performs such partial process image replacement for MPI applications at the user-level during application execution, thereby enabling adaptive changing of the set of replicated processes.

As mentioned in Section 3, an adaptation manager regularly updates the compute-replica map for the next interval based on failure predictions. The implementation of adaptive replica change in PAREP-MPI is independent of the implementation of the adaptation manager and the failure predictor. One of the challenges in the implementation of adaptation manager is developing an efficient data structure for compute-replica map and ensuring its consistency across nodes when updated. In our prototype implementation, we use a text file on a shared file system for the purpose. This compute-replica map file contains n entries, one entry per line, where n is the total number of compute nodes in the system. Each entry consists of a compute node number and the corresponding replica node number. If a compute node does not have a replica, its replica node number will be -1. Checking for updates in this file will have negligible overhead since it only involves checking the file modification time. Even when the file is modified, each process has to read only a single line from the file. A compute or replica process will then read the line corresponding to its associate compute node number. This overhead is negligible by itself, and even more negligible in current systems owing to the advanced read caching policies implemented by most operating systems. In our experiments, we have observed that the overhead of these operations related to the mapfile is about 30 milliseconds.

On completion of every MPI call from the application, PAREP-MPI checks the compute-replica map file for modifications before the control is returned to the application routine. Alternatively, the processes can check for updates selectively based on a time threshold. If the file is modified, the application processes first synchronize to determine whether there are pending MPI requests in any of the processes. A pending MPI request refers to non-blocking communication operations that have not completed. In such a case, the processes will have to wait for the requests to be completed before proceeding further. This is to ensure that there are no in-flight messages when a replica change is triggered. PAREP-MPI keeps track of MPI request handles in an internal data structure in each of the processes and posts a $Wait\ ()$ call on existing handles to complete the requests. This synchronization also ensures that the first process that notices a change in the compute-replica map will wait until all the other processes see the updated map and synchronize. After this step, each of the processes whose mapping has changed initiates the send/receive of process image to/from its newly mapped process, and the receiving process performs process image replacement, as explained in Section 3.

5 Process Image Replacement in PAREP-MPI

The strategy used in PAREP-MPI for replacing the image of a process with that of another process involves three steps: replacing the initialized and uninitialized data (henceforth referred to as data segment), replacing the user data structures in the heap (henceforth referred to as heap data structures) and replacing the stack segment. It is also important that after the image replacement is complete, the new replica process continues from exactly the same instruction the corresponding compute process executes just after sending its process image. Hence the *stack context* should also be saved.

5.1 Data Segment

Replacing the data segment is trivial if the start and end addresses of this segment are known. In Unix/Linux, the boundary addresses of this segment can be obtained from $/proc/self/maps$ file while the application is executing. Replacing this segment in the destination process involves replacing the entire data between the start and the end addresses.

MPI communicator handles in the replica process will have to be backed up in the stack before initiating a data segment replacement and should be restored back after replacing the data segment. This is done so that the process does not lose access to its communicator structures and hence retain its identity in the MPI environment, such as the process rank and other MPI specific information.

Our current proof-of-concept prototype implementation performs this saving and restoring of MPI communicators. Our implementation currently does not handle other MPI objects including MPI data types. In future, we plan to use the existing migration mechanisms similar to those present in the BLCR library [13] and the work by Wang et al. [12].

5.2 Heap Data Structures

Many of the MPI related data structures are stored in the heap space of a process. A complete heap replacement will result in the replacement of these MPI data structures. Since we require the process to keep its identity, these data structures will have to be retained. However, keeping track of the locations of all the MPI data structures would require the modification of the standard MPI source code. We adopt a different strategy where we keep track of only the user data structures in the heap and selectively replace them in the destination process. The heap data structures from the source process are copied to the corresponding locations of the same data structures in the destination process.

The above strategy to handle heap data structures gives rise to another problem. The pointers to the heap data structures reside in the data segment. After the data segment is replaced, these pointers would be pointing to addresses corresponding to the locations of these data structures in the source process. Since these locations may have changed

in the destination process, the pointers will have to be updated. For this, it is necessary to keep track of the addresses of these pointers in the data segment along with keeping track of the data structures itself. For example, suppose a user process allocates memory in the heap for an array of 10 integers using the following C statement.

$$int * p = (int *)\ malloc\ (10 * sizeof\ (int)).$$

To implement the selective heap data structure replacement, for each data structure, we should have the information about the actual data structure, i.e., its address in the heap and its total size in both the source and destination processes. Apart from this, we should also have the address of the pointer p in the data segment. Data structure information can be obtained by using wrappers for $malloc\ ()$ and $free\ ()$ to keep track of memory allotted and freed from the heap. To enable keeping track of the addresses of the pointers, we use a script that does text replacement of all the $malloc$ statements in the application source code with custom $malloc$ statements that will also pass the address of the pointer variable along with the size parameter of $malloc$. For example, the above $malloc$ statement in a C program will be replaced by the script as follows.

$$int * p = (int *)\ myMalloc\ (\&p, 10 * sizeof\ (int)).$$

Here, $myMalloc\ ()$ is the custom $malloc$ implementation that internally keeps track of the required heap data structure information in a linked list and calls the real $malloc$ function for dynamic memory allocation.

5.3 Stack Segment

Saving and restoring the stack context, and replacing the stack segment are implemented similar to Condor Checkpoint/Migration [27] using standard C functions $setjmp\ ()$ and $longjmp\ ()$, shifting the current execution stack to a temporary location in the data segment, and using this location as a temporary stack while the real stack space is getting replaced.

5.4 Illustration of Process Image Replacement: A Scenario with Three Nodes and Three Processes

Consider a system with three nodes and each node has an application process running on it. Suppose A, B' and B are the three processes in the system, such that B' is the replica process of B and A does not have a replica. Now, suppose that at a given point of time, the compute-replica map of the system is modified and according to the new map, the process B' should be a replica of A and B should not have a replica, i.e, B' should be changed to A'. Fig. 3 shows the execution time line of the three processes starting from the point where the processes read the compute-replica map modification, to the point where B' has changed to A' and has resumed execution. Following are the steps taken by the processes during this time line.

- All processes synchronize to check for pending non-blocking communication requests in any of the processes. If there are pending requests, processes proceed further only after they are completed.

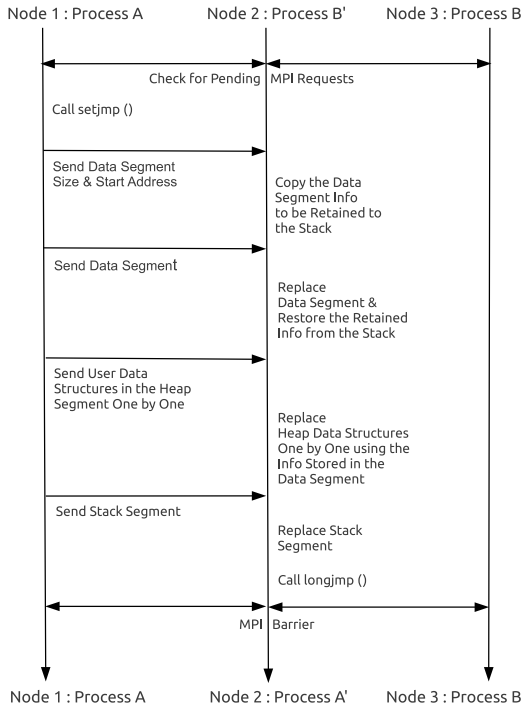


Fig. 3. Application execution time line during process replacement.

- A calls $setjmp$ (JMP_BUF) in an if condition that checks for a return value of 0. $setjmp$ returns 0 in A and hence executes the instructions in the if block (as follows in the remaining steps until $longjmp$ is called). A saves the start and end addresses of its stack segment in a global data structure, $stackSegAddr$.
- A sends the start address and size of its data segment to B'. B' receives them and takes a backup of MPI communicator handles.
- A sends its entire data segment. B' receives the data segment at the previously received start address. It restores the backed up data to the new data segment. This completes the data segment replacement.
- A sends the heap data structures one by one to B'. Heap data structures' information is stored in a link list in the heap. B' uses the information in its own list to receive the heap data structures at their corresponding locations in the heap. This completes the replacement of the heap data structures.
- A sends its stack segment to B'. Note that once the data segment is replaced, B' has the global data structure, $stackSegAddr$ containing the start and end addresses of the stack segment of A. B' shifts its execution to a temporary stack in its data segment and receives the stack segment from A at the stack start address in $stackSegAddr$.
- B' calls $longjmp$ (JMP_BUF, n), where n is a non-zero integer. This restores the stack context of A in B'. Now B' has become A', i.e., the replica of A. The process resumes at the point where $setjmp$ had returned when it was first called in A, with the return value n. The process continues execution from the next instruction after the if block. This completes the entire process image replacement.
- All processes synchronize and continues execution.

6 Experiments and Results

We evaluate our adaptive replication framework based on the efficiency achieved by the application in the presence of failures. Efficiency is defined as the percentage of work W_{opt} that the application completes in the presence of failures in a given time duration, where W_{opt} is the work done by the application in the time duration in a failure free environment. It is assumed that the application has linear scalability. Evaluation is done based on simulations using a failure simulator, which takes as input the failure trace of a system and accuracy metrics of the failure predictor. All experiments were done for an application execution time of one week.

For our experiments, synthetic failure traces are generated for large scale systems assuming two kinds of node failure distributions—exponential and Weibull, which are commonly used to model node failures in large scale systems [10], [28]. A projection based on failure statistics in the Jaguar supercomputer of Oak Ridge National Laboratory with $45{,}208$ processors has shown that the per processor MTBF in the platform can be estimated as approximately 125 years [29]. For our experiments, we consider a per processor MTBF of 25 years. The parameter λ for exponential distribution is calculated as $\lambda = \frac{1}{MTBF}$. For Weibull distribution which is defined by two parameters λ and k, k is set as 0.7 based on the study in [28] and λ is calculated as $\lambda = MTBF/\Gamma(1 + \frac{1}{k})$, where Γ denotes the gamma function. Synthetic failure traces were generated for a period of one year and for each of the generated traces, the subset of the trace corresponding to the first week of the sixth month was taken for our experiments involving the simulation of application execution for one week.

Our framework divides the set of nodes allocated for an application into two mutually exclusive sets, namely, compute set and replica set. In our analysis, we found that the maximum number of node failures experienced by a system with number of nodes as high as $200{,}000$ is only less than 400 over a one week period. Based on this analysis, we fix the number of nodes in the replica set as 1 percent of the total number of nodes. This assumption would result in $2{,}000$ nodes in the replica set for a $200{,}000$ node system, which is much higher than the observed number of node failures. A 1 percent reduction in the number of nodes used for application execution will only result in a maximum of 1 percent reduction in efficiency.

We have evaluated our framework against periodic checkpointing and dual redundancy. We have also compared our adaptive replication framework with proactive live process migration, another approach that uses process image replacement and failure predictions. For periodic checkpointing, the optimal checkpoint interval is calculated using Daly's higher order checkpoint/restart model [26], which takes as input, the platform MTBF and the overhead of checkpointing. Platform MTBF required by this model is determined as the average observed platform MTBF during a one month failure trace history before the start of application execution. Dual redundancy is implemented as given in [10].

Evaluations on large scale systems were done for number of nodes ranging from $10{,}000$ to $200{,}000$. A system with $100{,}000$ nodes can be considered as representative of the existing petascale systems [1]. Similarly, a system with $200{,}000$ nodes can be considered as representative of a projected exascale system based on exascale computing studies [4].

For the purpose of our evaluation, we assume the following overheads for checkpoint/restart: checkpoint time $= 5$ minutes, down time $= 1$ minute, recovery time $= 5$ minutes. These values are in accordance with the values given for the 2011 cost scenario in [8]. We have also performed an experiment with varying checkpoint/recovery overheads. We set the time interval between failure predictions as 30 minutes. This selection is based on the results in previous efforts on failure predictors [17] that report the best accuracy metrics for a time window between 15 minutes to 1 hour depending on the system. In our experiments, unless otherwise mentioned, both precision and recall of the failure predictor are assumed as 0.7. While we use this failure prediction accuracy as a baseline, we show the results on the sensitivity analysis of our adaptive replication strategy to the accuracy of failure predictions in Section 6.7.

6.1 Runtime Overhead of PaRep-MPI

6.1.1 Overhead due to Replication

We define replication overhead in terms of the percentage loss in work (e.g., number of iterations) done by the application on the computation nodes. The primary replication overhead is due to the extra messages sent to the replicated nodes. We assume a communication pattern in the application in which communications happen to all the nodes including the replicated nodes (e.g., broadcast, scatter, all-to-all etc.). With this assumption, the overhead due to partial replication using C computation nodes and PR replication nodes is x times the overhead due to dual redundancy using C computation nodes and $DR(= C)$ replicated nodes, where $x(= PR/C)$ is the percentage of the nodes replicated. For example, in a 1.5-degree partial replication, the overhead will be half of the dual redundancy overhead.

Process replication strategies in our prototype implementation of PaRep-MPI are similar to those used in *r*MPI [10] and RedMPI [11]. Evaluations using *r*MPI have concluded that with dual redundancy, the overhead due to replication for a worst-case application, SAGE, on a projected exascale system is 4.9 percent [10]. Based on this, for our experiments, we use a overhead of 4.9 percent for dual redundancy and a overhead of $x(= PR/C) \times 4.9\%$ for partial replication when both the replication strategies use C computation nodes. We also show results with varying percentages of dual-redundancy and partial replication overheads in Section 6.4.

6.1.2 Overhead of Adaptive Replica Change

Overhead of adaptive replica change will depend on the memory footprint of the application. This overhead also includes the overhead related to reading the compute-replica mapfile for changes. In adaptive replication, modification of the compute-replica map at a given time can result in the need for multiple replica changes. PaRep-MPI can handle these multiple process replacements in parallel since a process replacement involves only the source and the

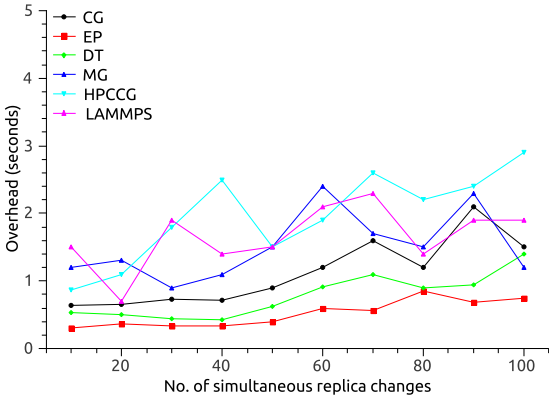


Fig. 4. Overhead of dynamic replica changes using PAREP-MPI on IBM Bluegene/L.

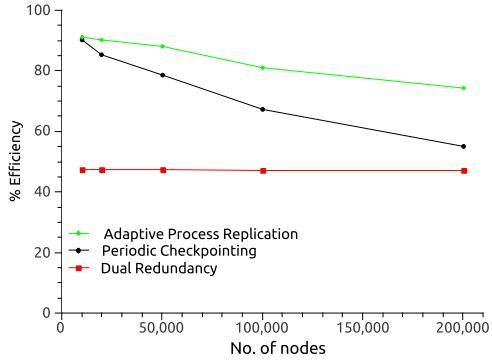
destination processes. Hence the overhead due to single process replacement is the same as the overhead due to multiple process replacements.

For the overhead analysis, we have taken four applications from NAS Parallel Benchmarks (NPB 3.3) [30], namely, CG, EP, DT and MG and two other scientific applications, namely, HPCCG (version 0.5) [31] for unstructured implicit finite element or finite volume computations and LAMMPS [32] for molecular dynamics simulations. All the NPB applications were executed with Class C data. HPCCG was executed with a problem dimension of $64 \times 64 \times 64$ grid points and LAMMPS was executed with the *Rhodo* spin protein benchmark from the LAMMPS website. The replica change overheads were analyzed on an IBM Bluegene/L system running on Linux. The applications were executed on 1,024 processors. An extra 100 processors were used for replication, making the total number of processors used for execution as 1,124. The evaluation on 1,024 processors was to analyze the possible network contentions under large number of simultaneous process replacements.

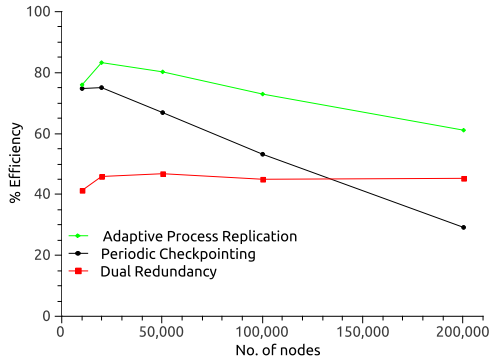
Runtime overhead was observed for applications for increasing number of simultaneous replica changes, ranging from 1 to 100. Fig. 4 shows the results of the experiments. Each value in the graph is the average of values obtained from five runs. It can be observed that for all the applications, the overhead is a few seconds, with a worst-case of only less than 3 seconds. This overhead of adaptive replica change is slightly smaller than the overhead of 1-6.5 seconds for proactive live-process migration using BLCR library reported in the work by Wang et al. [12]. Though our experiments have shown runtime overheads of only a few seconds for a single set of parallel replica changes, for all our experiments, we have assumed this overhead to be 1 minute for worst-case analysis.

6.1.3 Overhead due to False Positives

In a fault tolerant system that relies on failure predictions, it is important to evaluate the effect of overheads due to false positives (FPs) [33], since FPs lead to unwanted proactive action (in our case, unwanted replica changes). Since false positives affect precision (precision = TP/(TP + FP)), we performed a simulation with a worst-case precision of 10 percent for exascale configuration of 200,000 nodes with Weibull distribution of failures. We experimented with three different recall values of 25 percent, 50 percent, and 70 percent. In all cases, we found that the overhead or the wasted time due to adaptive replica change for false positives is only about 4 percent. False negatives do not lead to any proactive action and hence do not result in replica change overhead.



(a) Exponential Distribution



(b) Weibull Distribution

Fig. 5. Comparison of adaptive replication with other techniques.

6.2 Performance on Large Scale Systems

Fig. 5 shows the performance of our framework against the other two techniques for up to 200,000 nodes with both exponential and Weibull distributions for failures. The graphs show that adaptive replication significantly outperforms the other techniques for both the failure distributions. Our strategy performs 20 percent better than the best of the other two techniques for exponential distribution and 16 percent better than the best of the other two techniques for Weibull distributions even for 200,000 nodes, which is the projected number of nodes for an exascale system.

It can be observed that the performance of periodic checkpointing decreases drastically as the number of nodes increases, which proves again that periodic checkpointing is not an efficient fault tolerance technique for future systems. An interesting observation is that dual redundancy consistently gives around 45 percent efficiency for any number of nodes. This is because in this technique, all nodes have a replica and the probability of a node and its replica failing at the same time is very less. In fact, in our analysis, we have observed that with dual redundancy, the application experienced at most only a single failure during a 1 week execution. But, though dual redundancy is extremely effective in avoiding failures, its disadvantage is that the maximum possible efficiency is 50 percent, while adaptive replication can provide much better efficiencies as shown

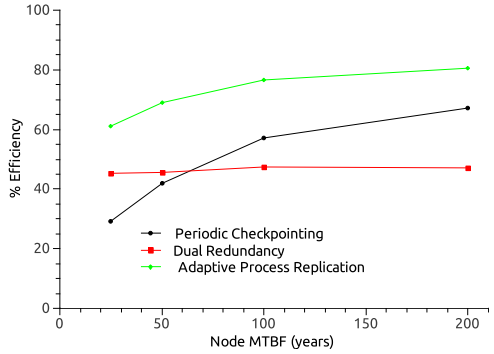


Fig. 6. Effect of varying node MTBF on adaptive replication.

by the results. For 200,000 nodes, our strategy makes more number of replica changes (371) than checkpointing (288), which asserts that adaptive replication is a promising fault tolerance solution for future systems like exascale systems.

Though adaptive replication outperforms dual redundancy by a significant margin even for a projected exascale system with 200,000 nodes, it should also be observed from Fig. 5 that there is a downward trend in the efficiency achieved by adaptive replication as we increase the number of nodes. If this trend continues, it is possible that at some higher number of nodes, the efficiency of adaptive replication goes below that of dual redundancy. But, projecting the graphs for higher number of nodes show that this will happen only at around 300,000 nodes for Weibull distribution and at around 1 million nodes for exponential distribution. Based on current trends and an anticipated node count of around 200,000 for exascale [4], it is likely to take at least 10 years for the node count in supercomputing systems to reach these large numbers. Moreover, our adaptive partial replication will still have advantages over dual redundancy, since any increase in replication overhead for certain applications in the future will affect dual redundancy much more significantly than our adaptive replication.

We also performed simulations for one million nodes with platform MTBF of 20 minutes. Our simulation results show that the application efficiency achieved with our adaptive process replication strategy is 60.71 percent, while the efficiencies obtained using dual redundancy and periodic checkpointing approach are 47.25 percent and 25.8 percent, respectively. Thus our adaptive replication method can give higher efficiencies for larger configurations of the projected exascale systems as well.

Results in Fig. 5 show that a Weibull distribution of failures provides a much more challenging failure management scenario than an exponential distribution. Moreover, studies have shown that real systems experience failures that follow Weibull distribution [28]. Hence, for the rest of our evaluations, we present results using Weibull distribution.

6.3 Performance for Different Node MTBFs

Fig. 6 shows the performance of our framework for node MTBFs ranging from 25 to 200 years for 200,000 nodes with Weibull failure distribution. As expected, both adaptive replication and periodic checkpointing shows better performance for higher node MTBFs due to the reduced number of node failures, and dual redundancy remains consistent in performance. Though adaptive replication outperforms periodic checkpointing as expected for all cases, it can be observed that the rate of increase for periodic checkpointing is higher than that for adaptive replication. Such a trend could mean that if the reliability of hardware increases, then some day in the future, periodic checkpointing can again turn out to be an effective fault tolerance solution.

6.4 Performance with Varying Replication Overheads

We have made two assumptions related to replication overheads in our experiments. First, based on the projections using *r*MPI in the work by Ferreira et al [10], we used a overhead of 4.9 percent for dual-redundancy. The above mentioned work also showed worst-case dual-redundancy overheads of less than 5 percent to about 20 percent for different applications on current large-scale systems. MR-MPI [22], which is another MPI process replication solution reports widely varying replication overheads which are as low as 0 percent for embarrassingly parallel applications (NAS EP benchmark) and as high as 70-90 percent for communication intensive applications (NAS IS and FT benchmarks).

Second, we assumed the partial replication overhead with C computation nodes as x times the dual-redundancy overhead with C computation nodes, where x is the percentage of replicated notes. This is true for applications in which communications happen to all nodes including the replicated nodes. In applications that involve near-neighbor communications (e.g., every pair of neighbors communicating), the partial replication overhead will be the same as the dual-redundancy overhead. In the experiments with MR-MPI [22], the partial replication overhead with 1.5-degree replication varied between 25-75 percent of the dual-redundancy overhead. In general, the percentage difference in overheads between dual and partial replication for a fixed number of computation nodes, C, depends on both the replication degree and the communication pattern in the applications.

We also conducted experiments with different dual and partial replication overheads for our exascale configuration of 200,000 nodes with 1 percent of the nodes replicated for Weibull failure distribution. For dual redundancy overhead, we investigated varying overhead percentages. For partial replication overheads, we investigated three scenarios, first, the *worst-case overhead*, in which the partial replication overhead is the same as the dual redundancy overhead with 198,000 computation nodes, second, the *best-case overhead*, in which the partial replication overhead is x times the dual redundancy overhead with x being the percentage of replicated nodes, and third, the *median overhead* in which the partial replication overhead is the median value of the above two scenarios. Fig. 7 shows the effects of these overheads on application efficiency. The x-axis denotes the dual-redundancy overhead using 198,000 computational nodes. The figure also shows the application efficiencies with periodic checkpointing approach and dual-redundancy strategy using 100,000 computational nodes.

The figure shows that the application efficiency is almost constant for partial replication with best-case overhead. This is because the amount of overhead in this case corresponds to the percentage of replication. Since we use 1

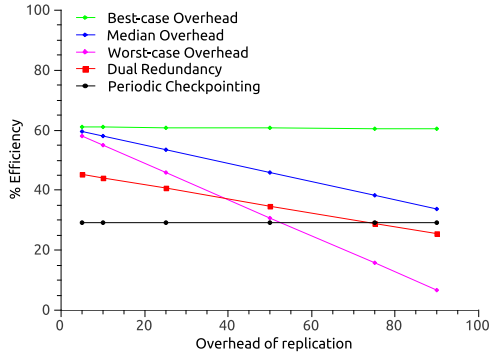


Fig. 7. Effect of varying replication overhead.

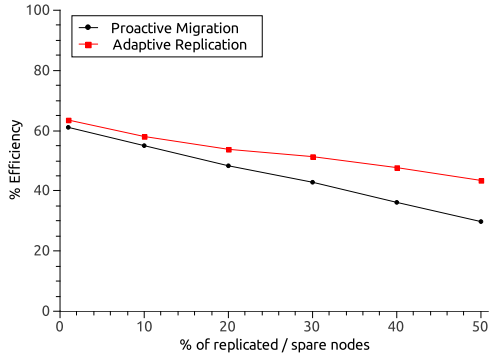


Fig. 8. Proactive migration versus adaptive replication (varying degree of replication).

percent replication in our experiments, the effect of this overhead is negligible on application efficiency. We find that the median overhead, which can be considered as average case in most of the applications, also gives better performance than periodic checkpointing and dual-redundancy approaches even for 90 percent overhead. However, we find that that partial replication with the worst-case overhead gives lower application efficiencies than dual-redundancy for overheads greater than 40 percent and than even periodic checkpointing approach for overheads greater than 55 percent. This shows that while our adaptive partial replication strategy can give the best efficiencies in most cases, the best fault tolerance strategy for a given case depends on the communication pattern in the application and the replication overhead.

6.5 Comparison with Proactive Process Migration

Proactive process migration [12] uses a similar strategy as adaptive process replication, using failure predictions and spare nodes. In proactive migration, the processes in failure prone nodes are migrated to healthy spare nodes at different points of application execution to avoid failures. Given that the overheads of process migration and adaptive replica change are comparable, both the strategies appear to be similar and hence it becomes essential to evaluate the benefits of adaptive replication over proactive migration.

A case when replication can lead to better application efficiencies is when the node to which a process is migrated in proactive process migration strategy fails, resulting in application failure. This can happen when the prediction on that node is a false negative. In replication, the process can still continue from its replica.

Theoretically, there are few other scenarios when our adaptive replication strategy can give better efficiencies than proactive process migration. Out of the four failure prediction scenarios of *true positives* (TP), *false positives* (FP), *true negatives* (TN), and *false negatives* (FN), our adaptive replication strategy has the potential to perform better than proactive migration in three scenarios, namely, TP, FP and FN, while both the strategies give equivalent performance in the remaining scenario of TN. The essential condition for adaptive replication to win in any of the three scenarios is that that the given node should already have a replica in the system, thereby avoiding a process image replacement or an application failure, which cannot be avoided in case of proactive migration.

One possible way to increase the probability that for the three potential winning scenarios for adaptive replication, the failure prone node has an already existing replica is to increase the replication degree or the percentage of nodes replicated. One motivation to increase the replication degree in our adaptive replication framework that uses failure predictions, is in cases when the recall values of the failure predictors are low. For low recall values, the number of failures predicted will be very less. In such a scenario where we have limited confidence on the ability of the predictor to predict impending node failures, having a high replication degree will be beneficial because of the high number of replicated nodes.

Fig. 8 shows the effect of varying the percentage of nodes replicated on application efficiency for both the strategies. For proactive migration, the x-axis values represent the percentages of nodes allocated as spare nodes. It can be observed that as the percentage of nodes replicated increases, though the overall application efficiency decreases for both the strategies due to the decrease in the number of compute nodes, replication outperforms proactive migration by greater margins as the replication degree increases. For example, adaptive replication gives 13.5 percent higher efficiency than proactive migration for 20 percent replication, and about 21 percent higher efficiency for 30 percent replication.

Our adaptive replication can be considered as combining the benefits of replication and proactive process migration strategies, since it uses process replacement (a light-weight process migration) for adaptive replica change. Thus, adaptive replication is expected to perform better than proactive migration for higher replication degrees and provide similar application efficiencies as proactive migration for lower replication degrees.

6.6 Comparison with Advanced Checkpointing Methods

In our previous results, we primarily compared adaptive replication with periodic checkpointing. Certain advanced checkpointing systems including multi-level checkpointing [14], [34], partial message-logging and in-memory checkpointing provide very less checkpointing overheads. One popular technique is multi-level checkpointing that uses a hierarchy of checkpointing methods including fast in-memory checkpointing at the lowest level and high-latency storage to parallel file system at the highest level.

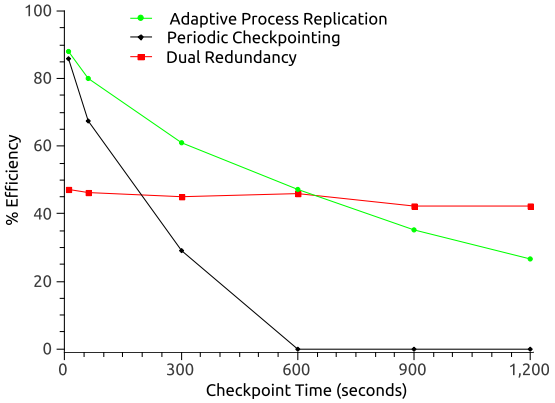


Fig. 9. Effect of varying checkpointing overhead.

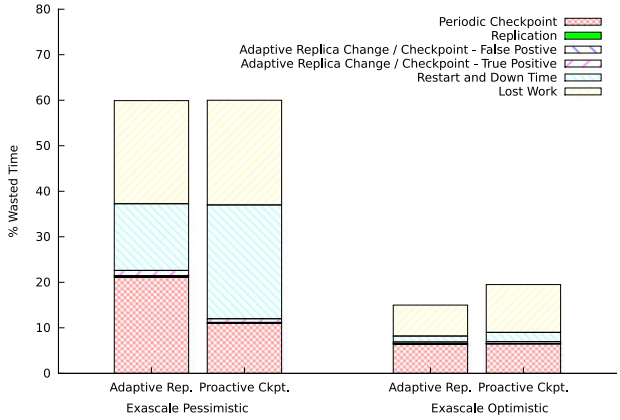


Fig. 10. Comparison with proactive checkpointing.

One such multi-level checkpointing scheme, FTI [34], provides less than 10 percent checkpointing overhead. Also, with anticipated hardware developments including phase change memories (PCM) and improvement in checkpointing techniques, checkpointing systems target overheads of only a few seconds for future large scale applications and systems.

In order to evaluate the performance of adaptive process replication with fast checkpointing, we obtained results with different checkpointing overheads, ranging from five seconds to twenty minutes, for our exascale configuration with 200,000 nodes using Weibull failure distribution. Fig. 9 shows the application performance for different checkpointing overheads. We find that even for small checkpointing overheads of a few seconds, our adaptive replication methods gives better results than the periodic checkpointing scheme. We observe that as the checkpointing time reaches 10 minutes, the application efficiency rapidly falls to 0 percent when using only periodic checkpointing. At this stage, the application is not able to make progress due to frequent failures and rollbacks. We also find that for checkpointing overheads of greater than 10 minutes, dual redundancy performs better than adaptive replication. These results emphasize the need to minimize checkpointing overheads to a few seconds or minutes for large scale systems, as also observed by Bouguerra et al. [33].

In a recent work, Bouguerra et al. [33] have developed fast proactive checkpointing techniques that uses failure predictions in addition to periodic preventive checkpointing in their multi-level checkpointing scheme. Using analytical models, they dynamically determine whether to take proactive checkpoint on a failure prediction, and also calculate the optimum intervals between preventive checkpointing. Their results showed improvement in computing efficiency of up to 30 percent when compared to periodic checkpointing.

We compared our adaptive replication with their proactive checkpointing approach by using the same simulation settings used in their work, namely, exascale configuration of 100,000 nodes, exponential distribution of failures, system MTBFs of 30 minutes (exascale pessimistic) and 2 hours (exascale optimistic), checkpointing overheads of 10 minutes (exascale pessimistic) and 2.5 minutes (exascale optimistic), down time of 1 minute, restart cost equal to the checkpointing overhead, and predictions with recall of 50 percent and precision of 80 percent. Fig. 10 shows the results. The results for the proactive checkpointing shown in the figure are replicated from the corresponding results in their work. We find that our adaptive process replication gives slightly better application efficiency (about 5 percent) than proactive checkpointing for exascale optimistic configuration. While both the strategies give equivalent performance for exascale pessimistic configuration, we find that the proactive checkpointing gives smaller periodic checkpointing overhead due to careful computation of optimal checkpointing intervals and our adaptive partial replication gives smaller restart and downtime due to smaller number of application failures. An interesting future work will be to develop a hybrid strategy that combines both these strengths.

6.7 Impact of Errors in Failure Prediction

Although our framework assumes the presence of a failure predictor, it is agnostic of the specific failure prediction model used by the predictor. Failure predictors in real world are not 100 percent accurate. Taking this into account, we consider a failure predictor as a black box which monitors the system and generates predictions with some observed accuracy metrics. One of the components of our failure simulator is the prediction component which uses the failure trace and the expected prediction accuracy metrics to simulate the behavior of a failure predictor. Failure trace of a system gives the time line of node failures in the system. Accuracy metrics of the failure predictor include *precision*, which is defined as the ratio of the number of correct predictions to the total number of predictions made (True Positives / (True Positives + False Positives)) and *recall*, which is defined as the ratio of the number of correct predictions to the total number of failures (True Positives / (True Positives + False Negatives)). Given the failure trace and the accuracy metrics, the prediction component generates failure predictions at regular intervals of time with the required ratios of true positives, false positives, true negatives and false negatives so as to maintain the specified accuracy metrics. Our method of simulating the behavior of a failure predictor using accuracy metrics allow us to evaluate the performance of our framework with varying prediction errors without assuming any particular prediction model.

The surface plot in Fig. 11 shows the impact of varying accuracy metrics (precision and recall) of the failure predictor on the performance of adaptive replication for the exascale configuration with 200,000 nodes using Weibull

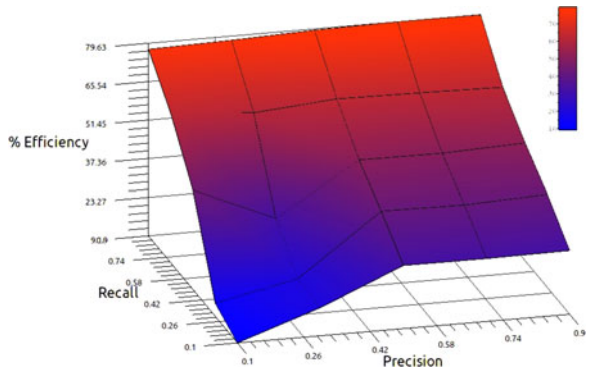


Fig. 11. Impact of errors in failure prediction on adaptive replication.

failure distribution. The figure shows that efficiency is more than 60 percent for all precision and recall values above 0.5. Failure predictors today have observed precision and recall values in the range of 0.5 to 0.8. Hence, adaptive replication gives high efficiency for all realistic values of precision and recall.

We also observe that the efficiency achieved by adaptive replication depends a lot on the recall of the predictor and does not depend much on its precision. The efficiency achieved is more than 75 percent for a recall of 0.9 even when the precision is only 0.1. But, if the recall is low, then the efficiency is poor even for a very high precision, as shown by the 32 percent efficiency observed with a precision of 0.9 and a recall of 0.1. Note that for a failure predictor, a high precision means that the number of correct predictions out of the total number of predictions is high, while a high recall means that the number of predicted failures out of the total number of failures is high. Hence application efficiency for adaptive replication is higher for larger number of correct failure predictions (true positives, i.e., predictions of failures that actually happen) and is independent of the number of wrong failure predictions (false positives, i.e., predictions of failures that do not happen). For example, suppose 10 nodes are actually going to fail in the next interval. Then irrespective of whether the predictor gives 20 or 200 predictions for the next interval, if the 10 correct predictions are included in the set of predictions, adaptive replication gives high efficiency. The above observation provides an interesting insight for future works on failure predictors. Most failure prediction methods today focus on achieving a high value for both precision and recall. Our analysis shows that a strategy like adaptive replication can perform well if the recall is high, even if the precision is very less. This correlates with the findings by Bouguerra et al. [33].

These results on the impact of prediction accuracy correlate with the findings by Tikotekar et al. [25] in which they also find that the prediction accuracy above 60 percent is necessary to maintain the overheads to below 20 percent, and thus to obtain about 80 percent application efficiency. While they evaluate this impact with only precision of the failure prediction, we also analyze the impacts due to recall of the prediction.

7 Conclusions and Future Work

We have developed a framework that uses partial replication along with adaptive changing of the set of replicated processes based on failure predictions, to provide effective fault tolerance for both medium and large scale systems. We have also developed an MPI prototype implementation PaRep-MPI, that supports partial replication and adaptive replica change for MPI applications with minimal runtime overhead. Simulations using failure traces of both exponential and Weibull distributions have shown that our adaptive replication strategy significantly outperforms periodic checkpointing and dual redundancy, providing up to 20 percent more efficiency even for a projected exascale system with 200,000 nodes. In future, we plan to consider application malleability to adaptively change the number of nodes in a replica set along with adaptive replica change and to develop better fault tolerance frameworks based on both the features.

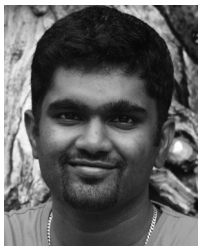
Acknowledgments

The authors would like to thank the reviewers for their very useful and detailed comments that helped them significantly improve the quality of the paper.

References

[1] Top 500 Supercomputing Sites. [Online]. Available: http://www.top500.org/, 2012.
[2] B. Schroeder and G. Gibson,"Understanding Failures in Petascale Computers," *J. Phys.: Conf. Series*, vol. 28, no. 1, pp. 012–022, 2007.
[3] F. Cappello,"Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 3, pp. 212–226, 2009.
[4] P. Kogge, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, and R. Lucas, "Exascale computing study: Technology challenges in achieving exascale systems," 2008, (P. Kogge, Editor and Study Lead) DARPA Tech Rep. TR-2008-13.
[5] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Proc. IEEE Int. Parallel Distributed Process. Symp.*, 2009, pp. 1–12.
[6] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni, "Unified model for assessing checkpointing protocols at extreme-scale," INRIA, Tech. Rep. RR-7950, 2012.
[7] C. George and S. Vadhiyar, "An adaptive framework for fault tolerance on large scale systems using application malleability," in *Proc. Int. Conf. Comput. Sci.*, 2012, pp. 166–175.
[8] F. Cappello, H. Casanova, and Y. Robert, "Checkpointing vs. migration for post-petascale supercomputers," in *Proc. 39th Int. Conf. Parallel Process.*, 2010, pp. 168–177.
[9] Z. Lan and Y. Li, "Adaptive fault management of parallel applications for high-performance computing," *IEEE Trans. Comput.*, vol. 57, no. 12, pp. 1647–1660, Dec. 2008.
[10] 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 *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 2011.
[11] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for HPC," in *Proc. 32nd Int. Conf. Distributed Comput. Syst.*, 2012, pp. 615–626.
[12] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in *SC '08: Proc. 2008 ACM/IEEE Conf. Supercomputing*, 2008, pp. 1–12.
[13] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," in *Proc. SCIDAC*, 2006, p. 494.
[14] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2010, pp. 1–11.

[15] K. Mohror, A. Moody, and B. De Supinski, "Asynchronous checkpoint migration with mrnet in the scalable checkpoint/restart library," in *Proc. IEEE/IFIP 42nd Int. Conf. Depend. Syst. Netw. Workshops*, 2012, pp. 1–6.
[16] P. Roth, D. Arnold, and B. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proc. ACM/IEEE Conf. Supercomputing*, 2003, pp. 21–21.
[17] P. Gujrati, Y. Li, Z. Lan, R. Thakur, and J. White, "A meta-learning failure predictor for blue gene/l systems," in *Proc. Int. Conf. Parallel Process.*, 2007, p. 40.
[18] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, no. 3, pp. 10:1–10:42, 2010.
[19] G. Hoffmann, K. Trivedi, and M. Malek, "A best practice guide to resource forecasting for computing systems," *IEEE Trans. Reliability*, vol. 56, no. 4, pp. 615–628, Dec. 2007.
[20] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into HPC systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 77:1–77:11.
[21] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok, "VolpexMPI: An MPI library for execution of parallel applications on volatile nodes," in *Proc. EuroPVM/MPI*, 2009, pp. 124–133.
[22] C. Engelmann and S. Böhm, "Redundant execution of HPC applications with MR-MPI," in *Proc. 10th IASTED Int. Conf. Parallel Distributed Comput. Netw.*, 2011, pp. 15–17.
[23] 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 *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 1–12.
[24] J. Stearley, K. Ferreira, D. Robinson, J. Laros, K. Pedretti, D. Arnold, P. Bridges, and R. Riesen, "Does partial replication pay off?" in *Proc. IEEE/IFIP 42nd Int. Conf. Depend. Syst. Netw. Workshops*, 2012, pp. 1–6.
[25] A. Tikotekar, G. Vallee, T. Naughton, S. Scott, and C. Leangsuksun, "Evaluation of fault-tolerant policies using simulation," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2007, pp. 303–311.
[26] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Comput. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.
[27] Checkpoint and migration of unix processes in the condor distributed processing system. [Online]. Available: http://research.cs.wisc.edu/condor/doc/ckpt97.pdf/, 2012.
[28] B. Schroeder, and G. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2006, pp. 337–350.
[29] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *SC '11: Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–11.
[30] NAS parallel benchmarks. [Online]. Available: http://nas.nasa.gov/publications/npb.html/, 2012.
[31] Sandia national laboratory - mantevo project. [Online]. Available: https://software.sandia.gov/mantevo/, 2012.
[32] Sandia national laboratory - lammps molecular dynamics simulator. [Online]. Available: https://lammps.sandia.gov/, 2012.
[33] M.-S. Bouguerra, A. Gainaru, L. Bautista-Gomez, F. Cappello, S. Matsuoka, and N. Maruyama, "Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing," in *Proc. IEEE 27th Int. Symp. Parallel Distributed Process.*, 2013, pp. 501–512.
[34] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *Proc. 2011 Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 32:1–32:32.



Cijo George received the Bachelor's degree in computer science and engineering from Cochin University of Science and Technology, Cochin, India, in 2009, and the Master's degree from the Supercomputer Education and Research Centre of Indian Institute of Science, Bangalore, India, in 2013. He is currently with NetApp Advanced Technology Group, Bangalore, the research group of NetApp in India. He worked on adaptive fault tolerance strategies for large-scale systems as part of his thesis work at Indian Institute of Science. He was with Nokia Siemens Networks, Bangalore, as a Software Engineer before joining Indian Institute of Science. His current work at NetApp is in the domain of Data Science, involving application of statistical analysis and machine learning techniques for data-driven management of distributed systems.

Sathish Vadhiyar received the BE degree from the Department of Computer Science and Engineering at Thiagarajar College of Engineering, Madurai, India, in 1997, the Master's degree in computer science at Clemson University, Clemson, SC, in 1999, and the PhD degree from the Department of Computer Science at the University of Tennessee, Knoxville, TN, in 2003. He is currently an Associate Professor in Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India. His research areas are building application frameworks including runtime frameworks for irregular applications, hybrid execution strategies, and programming models for accelerator-based systems, processor allocation, mapping and remapping strategies for Torus networks for different application classes including irregular, multi-physics, climate and weather applications, middleware for production supercomputer systems and fault tolerance for large-scale systems.