DOI 10.1109/CCGRID.2017.40

Designing and Modelling Selective Replication for Fault-tolerant HPC Applications

Omer Subasi[1,2], Gulay Yalcin[3], Ferad Zyulkyarov[1], Osman Unsal[1], Jesus Labarta[1,2]
[1]*Barcelona Supercomputing Center,* [2]*Universitat Politecnica de Catalunya, Spain,* [3]*Abdullah Gul University, Turkey*
{*omer.subasi, ferad.zyulkyarov, osman.unsal, jesus.labarta*}*@bsc.es, gulay.yalcin@agu.edu.tr*

***Abstract*—Fail-stop errors and Silent Data Corruptions (SDCs) are the most common failure modes for High Performance Computing (HPC) applications. There are studies that address fail-stop errors and studies that address SDCs. However few studies address both types of errors together. In this paper we propose a software-based selective replication technique for HPC applications for both fail-stop errors and SDCs. Since complete replication of applications can be costly in terms of resources, we develop a runtime-based technique for selective replication. Selective replication provides an opportunity to meet HPC reliability targets while decreasing resource costs. Our technique is low-overhead, automatic and completely transparent to the user.**

I. Introduction

As High Performance Computing (HPC) systems grow in size and complexity, they become more vulnerable to faults [4]. There are two types of errors that threaten the computations of HPC applications. The first is fail-stop errors in which the failure is detected and the application is aborted to avoid further propagation of the error. The second one is the data corruptions that are not detected by hardware ECCs. In this case, the data corruption is called *silent* since it is undetected. Silent data corruptions (SDCs) jeopardize the correctness of the results of HPC applications [8] and as a result they pose a significant threat. Although there are studies that address fail-stop errors or SDCs, there are few studies that address both types of errors.

In this work we combine redundant computation and checkpoint/restart to address SDCs and fail-stop errors of HPC applications to increase reliability. Redundant computation and checkpoint/restart are two well-known techniques to achieve fault-tolerance. In redundant computation, multiple replicas of a program are executed in parallel. Redundant computation can be used for recovering from task failures as well as for detecting silent errors. It recovers from fail-stop errors since if a replica fails, the remaining replicas can still continue their computations. It detects silent errors, such as data corruptions, by comparing the results of the replicas. However, detecting SDCs is not sufficient, it is also necessary to recover from SDC errors. Checkpoint/restart can be utilized for SDC error recovery. In checkpoint/restart, the state of the computation, called checkpoint, is saved periodically and when a SDC error is detected, the computation restarts from the latest checkpoint thus recovering from SDC.

The straightforward way to protect against fail-stop errors and SDCs is the complete replication of applications[1]. However complete replication may be prohibitive due to the high resource cost and in fact might be excessive due to the uneven susceptibility of the different application phases to SDCs [10]. Therefore effective and efficient techniques are needed to selectively replicate tasks. However the optimal selective replication is NP-hard which can be formalized as a bounded knapsack problem [11]. Consequently, practical selective replication solutions must employ heuristics. In our main contribution, we propose a runtime-based, fully automatic and completely transparent heuristic, called Target_Rep, to selectively choose tasks for replication. Our design does not require any modifications at all to application code or operating system.

Target_Rep is useful in cases when the system is not fully utilized and the idle/spare resources can be used for replication. For example, 10% of the nodes might be idle, and we could utilize this spare capacity by replicating 10% of the tasks. This is especially attractive since our Target_Rep heuristic chooses the tasks which would improve the program reliability the most. Moreover, HPC systems are typically over-engineered with spare resources to handle unexpected surges - although usually less than by 100% as would be required by complete task replication - and these existing resources could be utilized by Target_Rep. With Target_Rep, users can set the maximum resource utilization for replication and our heuristic maximizes the application reliability by transparently and automatically replicating tasks but without exceeding the resource utilization.

In order to understand how much we would improve the overall system reliability by replicating a *process*[2], we develop a reliability model based on Markov chains. The reliability model provides a quantitative way to estimate the reliability of HPC applications. It is basically a mathematical framework to characterize the reliability of HPC programs with or without replication. Our model predicts the application FITs very accurately with only 0.26% deviation from the actual FITs obtained by fault-injection experiments. *To the best of our knowledge, this is the first mathematical model which quantifies the reliability of HPC programs while*

[1]We use replication to refer to replication and checkpoint/restart together.
[2]We use *process* as a unit of execution which can be a thread, task or MPI rank.

considering both fail-stop errors and SDCs. In particular, our model is the first that distinguishes the fail-stop errors and SDCs. The proposed reliability model is a powerful tool which enables us to develop efficient selective replication heuristics. Moreover these techniques are the first to solve the problems regarding the reliability and resource budgets of HPC applications.

As a use case, we apply our framework (i.e., model and heuristic) to task-parallel dataflow HPC applications where we replicate tasks. Results show that our Target_Rep heuristic stays within 5% of the optimum solutions with 50% target replication percentage. Moreover, our overhead results show that our selective replication heuristic has very low overheads.

Briefly, our contributions are:

- Development, validation and implementation of a reliability model for HPC programs based on Markov Chains.
- An automatic and efficient heuristic to selectively replicate tasks while reducing costs significantly.
- Design, implementation and evaluation of our heuristic using task-parallel programs as a case study.

The rest of this paper is organized as follows: Section II presents background information. Section III presents our reliability model. Section IV discusses our heuristic. Section V presents the experimental evaluation. Section VI surveys related work. Finally, Section VII summarizes this work.

II. Error Classification and Failure Model

Throughout this study, we refer to failures or errors as the manifestation of faults. Errors are classified into three categories based on their propagation (or lack thereof) from typical error detection/correction hardware. The first class is the Detected and Corrected Errors (DCE) where an error is detected and corrected by the hardware. The second class consists of errors that are Detected and Uncorrected Errors (DUE) where the hardware is unable to recover from the detected error. DUEs are expected to become more frequent in the future with the increasing likelihood of double-bit and multi-bit flips [6, 15] for caches and memory. Moreover, a single bit flip in parity protected processor structures such as register files could also lead to a DUE. DUEs typically result in the crash (fail-stop) of applications since it is not possible for the faulted processor/hardware to recover [21]. The third class of errors consists of Silent Data Corruptions (SDCs). In SDC, the error is not detected, and the application terminates with wrong results. Recent research suggests that SDC can be a serious threat for HPC and exascale [8, 14]. A previous study at CERN found that SDC could be a serious concern since the observed SDC rate was orders of magnitude higher than manufacturer specifications [12]. Thus in this study we target SDCs and fail-stop errors.

III. Reliability Modelling HPC Applications

We now introduce and elaborate on our theoretical model for estimating the reliability of HPC programs. We start by introducing the reliability model when program processes are not replicated. Then we extend the model using Markov Chains to account for the case when processes can be replicated.

A. Reliability Model for HPC Programs without Replication

In this section we present our formal reliability model in which we define the intrinsic reliability of processes, and the overall and instantaneous reliability of HPC programs. This model sets the base for our model that incorporates process replication in the next section.

Let $ExecTime(P)$ be the amount of time that the process P takes to finish its computation. We define *the intrinsic reliability* of the process P during $0 - t$ period, denoted by $R_{intr}(P,t)$, as the probability it will *not crash* and *not experience SDCs* from the beginning of its computation ($t = 0$) until time t where $0 <= t <= ExecTime(P)$ due to process-local errors. We assume that the system on which the HPC programs are running is in its useful lifetime which means that the failure rate is constant and thus the intrinsic reliability has an exponential distribution [20]. Mathematically, $R_{intr}(P,t) = e^{-\lambda(P)t}$ where $\lambda(P)$ is the failure rate of process P. Note that the unit of $\lambda(P)$ is Failures in Time (FIT) showing the number of failures in one billion hours. The intrinsic reliability of the process when it finishes, i.e., $t = ExecTime(P)$, is denoted by $R_{intr}(P,\perp)$, and $R_{intr}(P,\perp) = e^{-\lambda(P)\times ExecTime(P)}$.

Note that it is straightforward to modify the reliability definition to account for different phases in the lifetime of a computing system such as for the aging phase of a system by Weibull distribution where $R_{intr}(P,t) = e^{-\lambda(P)(t/\alpha)^{\beta}}$. In Weibull distribution α is known as the scale parameter and β is the shape parameter. Briefly, $\beta < 1$ models the burn-in phase, $\beta = 1$ models the useful lifetime phase (equivalent to the exponential distribution we use) and $\beta > 1$ models the aging (wear-out) phase in Weibull distribution [20]. Consequently, our model - without any further modification - can address varying failure rates of processes during their computations.

Finally we define *the overall* and *instantaneous* reliability of a distributed HPC program respectively as follows:

$$R(App) = \prod_{i}^{|\aleph|} R_{intr}(P_i,\perp) \tag{1}$$

$$R(App,t) = \prod_{i}^{|\varepsilon(t)|} R_{intr}(P_i,t) \tag{2}$$

where $\aleph$ is the set of all application processes and $\varepsilon(t)$ is the set of executing processes at time t. Basically the

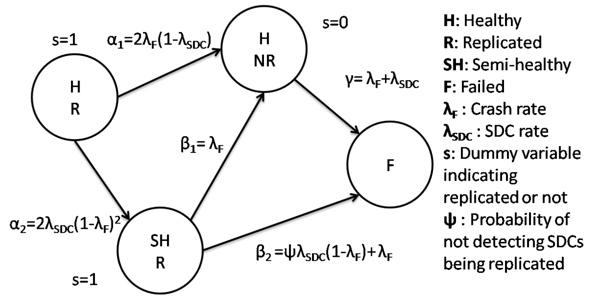

Figure 1. The Markov model for a single process

reliability of a distributed program is the product of intrinsic reliabilities of all processes in the program.

In this section we described the reliability model for programs where processes are not replicated. Now in the next section we continue by extending this model for programs in which processes are replicated.

B. Reliability Model for HPC Programs with Replication

When process replication is available, it is not possible (or may not be the best fit) to use only combinatorial techniques to model the intrinsic reliability for the processes. This is because in case of being replicated, a process has different states during its computation and moves among them with some error rates. Thus we establish a Markov model based reliability characterization formalism by leveraging Markov chains. The model is constructed for *a single process* and we use it to remodel the *intrinsic reliabilities* of the processes. The reliability definitions (Equations 1 and 2) for a distributed program remain the same.

A Markov model is a stochastic model assuming the *memoryless* property in which the future states are determined solely by the present state and are independent of past states. A Markov chain is a mathematical system where the system (in our case an individual process) goes through transitions from one state to another in the state diagram. We establish the state diagram where a process goes through the possible set of states with certain error rates. Figure 1 shows our model for a single process. We chose duplication (100%) of processes instead of triplication (200%) to incur less cost in the fault-free computation. State HR represents the case where the process is *healthy* and replicated. A process is *healthy* if it has not experienced any SDCs and has not failed (crashed/fail-stopped). State SHR represents the case where the process is replicated and is *semi-healthy*. A process is *semi-healthy* if none of the replicas has crashed but one of them has experienced some SDCs. State HNR refers to the case where exactly one of the replicas has crashed but the remaining one has no SDCs, i.e., it is *healthy*. Finally, state F indicates that the process is *in failure*. That is, both replicas have crashed or they have experienced SDCs that have not been detected.

We now elaborate on the transitions between the states. Transitions encapsulate the rate and conditions for them to occur between states. $\lambda_{SDC}(P)$ refers to the rate of experiencing a SDC for process P. Similarly $\lambda_F(P)$ is the rate for failures or crashes due to fail-stop errors for process P. The transition from state HR to state HNR shows that only one of the replica crashes and the other does not experience any SDCs for which there are two possible combinations. We denote it by $\alpha_1 = 2\lambda_F(P)(1-\lambda_{SDC}(P))$. The transition from state HR to state SHR refers the case where none of the replicas crashes and exactly one of them experiences some SDCs for which there are two possible combinations. We denote it by $\alpha_2 = 2\lambda_{SDC}(P)(1-\lambda_F(P))^2$. The transition from state SHR to state HNR indicates that the replica which experienced some SDCs actually crashes and the remaining replica has no SDCs i.e., healthy. We denote it by $\beta_1 = \lambda_F(P)$. The transition from state SHR to state F represents the case where either the healthy replica crashes or the replicas experience the same number of SDCs in the same memory locations whose probability is represented by ψ. The latter causes SDCs to go undetected. If the size of the memory usage of a process is N bits in total and k errors occur in N bits, then $\psi = 1/C(N,k)$ ($C(N,k)$ denotes N *choose* k). We denote this transition by $\beta_2 = \psi\lambda_{SDC}(P)(1-\lambda_F(P)) + \lambda_F(P)$. Finally, the transition from state HNR to state F shows that the remaining replica crashes or experiences some SDCs. We denote it by $\gamma = \lambda_F(P) + \lambda_{SDC}(P)$.

We now establish the Markov equations from the state diagram from which we will derive *the intrinsic reliability* of a single process. Let $P_{HR}(t)$, $P_{SHR}(t)$, $P_{HNR}(t)$ and $P_F(t)$ be the probability of a process to be in the state HR, SHR, HNR and F respectively. We use the dummy s variable to indicate whether a process is replicated before its computation ($s=1$) or not ($s=0$). The s variable is set when a process is chosen to be selected by the runtime heuristic. Then the series of differential equations describing the state diagram for a single process is

$$\frac{dP_{HR}(t)}{dt} = -(\alpha_1+\alpha_2)P_{HR}(t) \tag{3}$$

$$\frac{dP_{SHR}(t)}{dt} = \alpha_2 P_{HR}(t) - (\beta_1+\beta_2)P_{SHR}(t) \tag{4}$$

$$\frac{dP_{HNR}(t)}{dt} = \alpha_1 P_{HR}(t) + \beta_1 P_{SHR}(t) - \gamma P_{HNR}(t) \tag{5}$$

$$\frac{dP_F(t)}{dt} = \beta_2 P_{SHR}(t) + \gamma P_{HNR}(t) \tag{6}$$

with the initial conditions:
$P_{HR}(0) = s, P_{SHR}(0) = 0, P_{HNR}(0) = 1-s$ and $P_F(0) = 0$.

The solution to this series of differential equations of Markov model is:

$$P_{HR}(t) = se^{-(\alpha_1+\alpha_2)t} \tag{7}$$

$$P_{SHR}(t) = C_1 \times (e^{-(\alpha_1+\alpha_2)t} - e^{-(\beta_1+\beta_2)t} \tag{8}$$

$$
\begin{aligned}
P_{HNR}(t) = {} & \frac{s\alpha_2}{\gamma - (\alpha_1 + \alpha_2)} \times e^{-(\alpha_1+\alpha_2)t} \\
& + C_2 \times e^{-(\alpha_1+\alpha_2)t} - C_3 \times e^{-(\beta_1+\beta_2)t} \\
& + (1 + s\frac{\gamma(\alpha_2 + \beta_1 + \beta_2) - \gamma^2 - \alpha_1\beta_2}{(\gamma - (\alpha_1 + \alpha_2))(\gamma - (\beta_1 + \beta_2))}) \times e^{-\gamma t}
\end{aligned} \tag{9}
$$

$$P_F(t) = 1 - P_{HR}(t) - P_{SHR}(t) - P_{HNR}(t) \tag{10}$$

where

$$C_1 = \frac{s\alpha_2}{\beta_1 + \beta_2 - (\alpha_1 + \alpha_2)} \tag{11}$$

$$C_2 = \frac{s\beta_1\alpha_2}{(\gamma - (\alpha_1 + \alpha_2))(\beta_1 + \beta_2 - (\alpha_1 + \alpha_2))} \tag{12}$$

$$C_3 = \frac{s\beta_1\alpha_2}{(\gamma - (\beta_1 + \beta_2))(\beta_1 + \beta_2 - (\alpha_1 + \alpha_2))} \tag{13}$$

Hence the intrinsic reliability of a process P when replication is available is defined as

$$R^{Rep}_{intr}(P, t) = P_{HR}(t) + P_{SHR}(t) + P_{HNR}(t). \tag{14}$$

From the formula above MTTF (Mean Time To Failure) [20] of a process P is calculated as:

$$MTTF(P) = \int_0^\infty R^{Rep}_{intr}(P, t)\, dt. \tag{15}$$

After calculating MTTF, we calculate FIT in billion hours of process P as follows:

$$FIT(P) = \frac{1}{MTTF(P)} \times 10^9. \tag{16}$$

For brevity we omit the FIT formula, but it directly follows from applying Equation 15 to the solutions.

An interesting ramification of our Markov model is that the selective replication will be more significant and the difference among the impacts of the processes on the overall application reliability will be even more visible: The intrinsic reliability of T can be written if it is replicated as follows:

$$R^{Rep}_{intr}(P, t) = c_1 \times e^{-(\alpha_1+\alpha_2)t} + c_2 \times e^{-(\beta_1+\beta_2)t} + c_3 \times e^{-\gamma t} \tag{17}$$

for some constants c_1, c_2 and c_3. In addition, if P is not replicated, its intrinsic reliability (it is in HNR state at the beginning of its computation) is

$$R_{intr}(P, t) = c'_3 \times e^{-\gamma t} \tag{18}$$

for some constant c'_3. Thus *the reliability impact (improvement) factor* of P is

$$\frac{R^{Rep}_{intr}(P, t)}{R_{intr}(P, t)} = \frac{c_1}{c'_3} \times e^{(\gamma-(\alpha_1+\alpha_2))t} + \frac{c_2}{c'_3} \times e^{(\gamma-(\beta_1+\beta_2))t} + \frac{c_3}{c'_3}, \tag{19}$$

which is an exponential function.

Algorithm 1: Target_Rep Heuristic

```
   Input: $x$: Target % of replication to be performed.
   Output: $appFIT$ is the final FIT of the application
1  Algorithm begin
2      $appFIT = 0$; /* Global atomic variable */
3      $numSelected = 0$; /* Global atomic variable */
4      $target = x$; $locCounter = 0$;
5      For each thread in Threadpool:
6      repeat
7          $createprocessesAndAddtoQueue(RdQ)$;
8          if ($locCounter == 0$) then
9              if ($numSelected < target \times RdQ.size()$) then
10                 $SortedQueue\ queue$;
11                 $queue = sortandMarkOriginals(RdQ)$;
12                 $selectprocesses(queue)$;
13         process $p = RdQ.pollMarkedprocesses()$;
14         if $p == NULL$ then continue;
15         $executeprocess(p)$;
16         if $p.isTwin()$ then continue;
17         if $p.isSelected()$ then
18             $p.waitTwin()$; $locCounter --$;
19             $updateAppFITwithReplica(p, appFIT)$;
20         else
21             $updateAppFITwithNoReplica(p, appFIT)$
22     until (App is finished)
23 Function selectprocesses(SortedQueue queue)
24     while ($locCounter < x \times queue.size()$) do
25         process $temp = queue.peek()$;
26         $temp.selected = true$;
27         $RdQ.push(temp.duplicate())$;
28         $numSelected ++$; $locCounter ++$;
29         if ($numSelected == target$) then return;
```

IV. Selective Replication Heuristic

When selecting processes dynamically at runtime, our goal is to avoid requiring the knowledge of the entire execution (can be obtained by offline profiling) and to avoid keeping extensive information as the execution continues since both are expensive. Therefore we propose a heuristic that make use of only already existing information at runtime to achieve efficient, lightweight and near-optimal selective process replication.

A. Target_Rep Heuristic

Target_Rep aims to selectively replicate the $x\%$ of the application processes in the best possible way. Here, $x\%$ is the percentage of spare computational resources in the system. In essence, Target_Rep is a greedy approach that tries to get the global optimal solution by approximating and aggregating the local optimal solutions using the set of processes as the computation progresses. Algorithm 1 is the pseudo-code for Target_Rep. We omit details such as the synchronization among the threads for brevity. Since Target_Rep will not have the entire global information, we design it as follows to get a near-optimal solution: Target_Rep inspects the set of processes as needed, marks and sorts the processes according to their FITs (Lines 8-11). It selects the first $x\%$ of processes from the set of marked processes (Line 12). It accumulates the application FIT as the execution continues according to whether a process has been selected for replication (Lines 17-21).

Table I
DETAILS OF OUR TASK-PARALLEL HPC BENCHMARKS

Shared-memory Benchmarks	
Sparse LU	LU decomposition Matrix size 12800x12800 doubles, block size 200x200
Cholesky	Cholesky factorization Matrix size 16384x16384 doubles and block size 512x512
FFT	Fast Fourier Transform Matrix size 16384x16384 complex doubles, block size 16384x128
Perlin Noise	Noise generation to improve realism in motion pictures Array of pixels with size of 65536, block size 2048
Stream	Linear operations among arrays Array size 2048x2048 (doubles), block size 32768
Distributed Benchmarks	
Nbody	Interaction between N bodies Array size 65536 bodies, block size depends on #nodes
Matrix Multiplication	Matrix Multiplication using CBLAS Matrix size 9216x9216 doubles and block size 1024x1024
Pingpong	Computation and communication between pairs of taskes Array size 65536 doubles, block size 1024
Linpack	HPL Linpack Matrix size 131072 doubles, block size 256, 8x8 grid

V. EVALUATION

In this section we provide the evaluation and analysis of our techniques. We apply our ideas to task-parallel HPC programs and evaluate them with these applications. We implement our ideas in OmpSs [5] and Nanos [18]. We perform our experiments on Marenostrum supercomputer [2]. Up to 64 nodes and 16 cores per node are used in the experiments. Table I summarizes our benchmarks [1]. In shared-memory benchmark experiments all 16 cores in one node are used. In distributed benchmark experiments 1024 cores over 64 nodes are used.

First, we validate the estimations of our model against the results obtained in Monte Carlo simulations. Then we evaluate the efficacy of the Target_Rep Heuristic by comparing it to the optimal solutions.

A. Experimental Results

1) Model Validation Results: The reliability model from Section III is utilized at runtime for selective task replication to estimate the possible improvement in application's reliability if a task is replicated. To see whether our model is accurate we provide an extensive validation using Monte Carlo simulations. For the Monte Carlo simulations, we execute each benchmark 100× and during its executions we inject faults. For a successful validation we would expect that the results from the Monte Carlo simulations closely match those that are obtained from our reliability model.

Table II shows the results for the validation of our reliability model. We perform validation for the baseline replication where all tasks are replicated; this is the most general version of our model. The table shows the average FIT estimated by our model and by our injection based experiments. It also shows the difference between the estimated FIT and the measured FIT for each benchmark. On average, the difference between them is 4.9%. The model underestimates FIT for Cholesky, Pingpong and SparseLU, and for others it overestimates. Note that FITs reported in the table are in billion hours. We also report the standard deviations of our results for each benchmark in the last column.

Table II
MODEL VALIDATION RESULTS

Benchmark	Model FIT	Experiments FIT	% Difference	STD
Cholesky	3.31×10^0	3.57×10^0	7.28%	2×10^{-4}
FFT	1.0×10^0	9.2×10^{-1}	8%	3×10^{-4}
Linpack	1.7677×10^2	1.7633×10^2	5.33%	7.1
Matmul	3.48×10^{-2}	3.28×10^{-2}	5.74%	6×10^{-6}
Nbody	2.5×10^{-4}	2.4×10^{-4}	4%	3.6×10^{-7}
Perlin	2.35×10^{-1}	2.31×10^{-1}	0.002%	3×10^{-4}
Pingpong	1.14×10^{-4}	1.16×10^{-4}	1.75%	1.8×10^{-6}
SparseLU	3.6×10^{-3}	3.9×10^{-3}	7.69%	1.33×10^{-4}
Stream	2.2×10^{-2}	2.1×10^{-2}	4.54%	1.32×10^{-4}

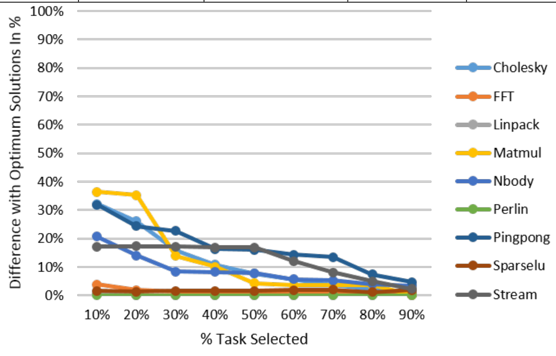

Figure 2. Target_Rep results

2) Evaluation of Target_Rep Heuristic: Target_Rep tries to maximize the reliability of an application while obeying the target percentage of task replication. We assess Target_Rep by comparing its output to the optimum solution to evaluate its efficacy. To get the optimum solution for a specified percentage, say $x\%$, of task replication, we first profile each benchmark and then we sort all the tasks according to their FITs (from smallest to the largest) calculated during profiling and we sort and store these FITs. We choose the first $x\%$ of the sorted tasks. The chosen set of tasks is the optimum solution for $x\%$ of tasks.

Figure 2 shows how close our solution is to the optimal one. The x-axis is the percentage of tasks being selected and the y-axis is the difference (in %) between the FITs that would be achieved with the tasks selected for replication by the optimum solutions and the FITs that Target_Rep achieves for a given percentage of task replication. As expected, as the replication percentage increases, the difference between the FITs achieved by Target_Rep and the optimal solution decreases. Overall, Target_Rep achieves close to the optimal solution. When half and more than half of the tasks are replicated (50% replication and more), the difference is only 5% and less than 5% on average respectively.

Moreover, our results show the overheads of Target_Rep with respect to fault-free execution (wall-clock) time are low and the average overhead is 1.2%. We omit overhead results for brevity.

VI. RELATED WORK

Replication is a well-known technique that has been adopted in various domains from aviation to distributed systems [13]. This technique has been used for reliability, performance and ensuring quality of service. However in most cases the complete replication of a system or an application can be prohibitively costly to achieve the intended purpose. As a result, selective replication becomes the only viable solution. For instance concerning the performance of systems, the work of Beckmann et al. [3] investigates selective replication to increase the performance of the caches of chip multiprocessors using a metric based on hit latency and misses. In case of aiming for better quality of service (QoS), Gruneberger et al. [9] propose a selective replication heuristic to increase QoS while keeping costs affordable for the distributed event-processing systems.

However selective replication as a way to address the trade-off between resource costs and reliability has not been investigated thoroughly, particularly in HPC community. Moreover, on one hand, the aforementioned studies [3, 9] cannot be employed to increase reliability while keeping cost affordable since those techniques and heuristics do not capture the reliability critical aspects of systems. On the other hand, there is the growing body of evidence showing that selective fault-tolerance support is of key-importance to decrease the resource costs while providing the required level of reliability. For instance, Luo et al. [10] and Fang et al. [7] find that different applications and different phases in applications exhibit different vulnerabilities. Although neither of these works state it explicitly, it follows that selective fault-tolerance is a natural fit to achieve a reasonable trade-off between costs and the required level of reliability for different applications. In our previous work [17] we proposed a runtime-based heuristic to obey application specific reliability thresholds. In this work we model HPC application reliability formally by Markov chains and we propose a dynamic runtime heuristic for utilizing idle resources to maximize the reliability of HPC applications. Our work [16] proposes a programmer-guided partial redundancy mechanism for SDCs and fail-stop errors.

Research on reliability modelling, such as [19, 22], has been conducted for various types of computing systems. However they all model the reliability of systems rather than that of applications to predict the failure rates of the systems. For instance, Thanakornworakij et al. [19] provide a model for HPC systems while Welke et al. [22] establish generic models based on Markov modelling that are applicable for hardware or software systems having well-known architectures such as simplex and triplex modular redundant architectures. However, none of these models can be utilized to achieve application-level reliability with selective redundancy under the failure rates of both SDCs and fail-stop errors.

VII. Conclusion

In this study we propose low-overhead and effective selective replication for HPC programs to mitigate SDCs and fail-stop errors. To achieve selective replication, we develop and validate a reliability model for HPC programs. Based on the model we present an automatic heuristic to select the tasks to replicate for using spare resources for redundancy.

This work is supported in part by the European Union Mont-blanc 2 Project (www.montblanc-project.eu), grant agreement no. 610402 and the FEDER funds under contract TIN2015-65316-P.

References

[1] BSC Application Repository: https://pm.bsc.es/projects/bar/wiki/applications.

[2] Marenostrum III: http://www.bsc.es/marenostrum-support-services/mn3.

[3] B. M. Beckmann, M. R. Marty, and D. A. Wood. Asr: Adaptive selective replication for cmp caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 443–454, Washington, DC, USA, 2006. IEEE Computer Society.

[4] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov. 2009.

[5] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[6] M. Ebrahimi, H. Asadi, and M. B. Tahoori. A layout-based approach for multiple event transient analysis. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 100:1–100:6, New York, NY, USA, 2013. ACM.

[7] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. Evaluating the error resilience of parallel programs. In *The 4th Fault Tolerance for HPC at eXtreme Scale Workshop (FTXS)*, 2014.

[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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 78:1–78:12, 2012.

[9] F. J. Grneberger, T. Heinze, and P. Felber. Adaptive selective replication for complex event processing systems. In *BD3@VLDB*, volume 1018 of *CEUR Workshop Proceedings*, pages 31–36. CEUR-WS.org, 2013.

[10] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 467–478, Washington, DC, USA, 2014. IEEE Computer Society.

[11] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

[12] B. Panzer-Steindel. Data integrity. In *CERN/IT Draft 1.3*, 2007.

[13] D. Siewiorek and R. Swarz. *Reliable Computer Systems: Design and Evaluation*. 1998.

[14] M. Snir, R. W. Wisniewski, J. A. Abraham, and A. et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2), 2014.

[15] V. Sridharan and D. Liberty. A study of dram failures in the field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 76:1–76:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[16] O. Subasi, J. A. Moreno, O. S. Unsal, J. Labarta, and A. Cristal. Programmer-directed partial redundancy for resilient HPC. In *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15, Ischia, Italy, May 18-21, 2015*, pages 47:1–47:2, 2015.

[17] O. Subasi, G. Yalcin, F. Zyulkyarov, O. S. Unsal, and J. Labarta. A runtime heuristic to selectively replicate tasks for application-specific reliability targets. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, pages 498–505, 2016.

[18] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé. Support for OpenMP tasks in nanos v4. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, pages 256–259, 2007.

[19] T. Thanakornworakij, R. Nassar, C. B. Leangsuksun, and M. Paun. Reliability model of a system of k nodes with simultaneous failures for high-performance computing applications. *Int. J. High Perform. Comput. Appl.*, 27(4):474–482, Nov. 2013.

[20] A. Verma, S. Ajit, and D. Karanki. *Reliability and Safety Engineering*. 2010.

[21] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 264–275, June 2004.

[22] S. R. Welke, B. W. Johnson, and J. H. Aylor. Reliability modeling of hardware/software systems. *IEEE Transactions on Reliability*, 44(3):413–418, Sep 1995.