Efficient Fault Tolerance through Dynamic Node Replacement

Suraj Prabhakaran*
Intel Corporation
Germany
suraj.prabhakaran@intel.com

Marcel Neumann*
AXA Konzern AG
Aachen, Germany
marcel.neumann@axa.de

Felix Wolf
Technische Universitaet Darmstadt
Darmstadt, Germany
wolf@cs.tu-darmstadt.de

***Abstract*—The mean time between failures of upcoming exascale systems is expected to be one hour or less. To be able to successfully complete execution of applications in such scenarios, several improved checkpoint/restart mechanisms such as multi-level checkpointing are being developed. Today, resource management systems handle job interruptions due to node failures by restarting the affected job from a checkpoint on a fresh set of nodes. This method, however, will add non-negligible overhead and will not allow taking full advantage of multi-level checkpointing in future systems. Alternatively, some spare nodes can be allocated for each job so that only processes that die on the failed nodes need to be restarted on spare nodes. However, given the magnitude of the expected failure rates, the number of spare nodes to be allocated for each job would be high, causing significant resource wastage. This work proposes a dynamic way handling node failures by enabling on-the-fly replacement of failed nodes with healthy ones. We propose a dynamic node replacement algorithm that finds replacement nodes by utilizing the flexibility of moldable and malleable jobs. Our evaluation with a simulator shows that this approach can maintain high throughput even when a system is experiencing frequent node failures, thereby making it a perfect technique to complement multi-level checkpointing.**

I. Introduction

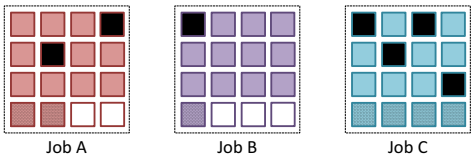
As we move into the exascale era, fault tolerance has been conceded as the most important challenge, owing to the high failure rates that exascale systems are expected to have. The mean time between failures (MTBF) is expected to be less than an hour for an exascale system. Although checkpoint/restart is a popular technique in petascale systems, existing methods cannot be directly transferred to exascale systems.

In current practice, a running application is check-pointed on a regular basis. Checkpointing is initiated either by the application itself or the batch system. When the job is affected by node failure, batch systems cancel the job and restart the job from the latest checkpoint on a fresh allocation of nodes. Checkpoints are usually made to disk-based storage and are therefore time consuming. Applications that need 30 minutes per checkpoint are not uncommon. If the checkpoint time is close to the MTBF, then most of the time is spent checkpointing and restarting with little application progress. This also lengthens job execution and turnaround time, which affects the overall throughput and availability of the system. An effective way to reduce the checkpoint time is in-memory checkpointing [1] and multi-level checkpointing such as FTI [2] and SCR [3]. Multi-level checkpointing involves combining different storage technologies pertaining to multiple levels of the storage hierarchy to store a checkpoint. The first few levels of storage are in-memory and remote-memory storage, while the last level is the file system. Despite the reduced checkpoint/restart time facilitated by this approach, the resource management limitations of current batch systems add other significant overheads.

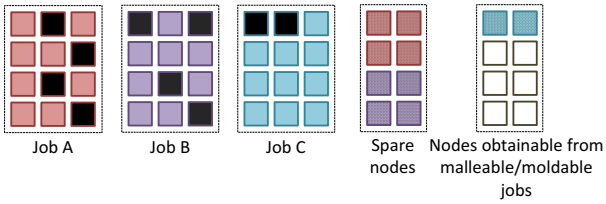
Canceling and restarting a job when it is affected by node failure introduces additional overhead. Even if the new allocation of nodes consists of a subset of the nodes previously used by the job (before failure), the processes and the data must be inserted afresh into the memory. Thus, the advantage of multi-level checkpointing is reduced. One way of circumventing this problem is to allocate dedicated spare nodes for each job, beyond what is actually required by the application as shown in Figure 1(a). In the event of a node failure, these spare nodes can be put to use immediately without having to resubmit the job. The application can then be restarted using the data from in-memory checkpoints. However, this approach requires a relatively large number of spare nodes to stay prepared for faults. This leads to a significant amount of nodes to remain unused and results in poor system utilization and throughput. Furthermore, a seemingly sufficient number of spare nodes allocated for a job may still turn out to be inadequate, as the pattern of faults may drastically vary. When no spare nodes are available, the job has to be cancelled and restarted afresh. Hence, current resource management mechanisms cannot be effectively used for fault tolerance at exascale.

In this work, we propose a *dynamic node-replacement* strategy that enables on-the-fly replacement of failed nodes with other healthy nodes. More precisely, we propose a dynamic node replacement algorithm that is deployed at the central scheduler and triggered at the event of a node failure. The algorithm finds nodes for replacement from various sources such as idle nodes and nodes obtainable from *malleable* and *moldable* jobs. We evaluate the proposed

*This work was performed when authors S. Prabhakaran and M. Neumann were affiliated to Technische Universitaet Darmstadt and RWTH Aachen University respectively.



(a) Static allocation: failed nodes replaced by statically allocated spare nodes



(b) Dynamic allocation: failed nodes dynamically replaced by nodes from a shared pool of spare nodes

Figure 1: Static and dynamic allocation of spare nodes in the event of node failure.

technique with a discrete event simulator and the results show that it can maintain high throughput even under high failure rates.

In principle, today's resource management systems can be extended with some software development effort to enable on-the-fly node replacement from a pool of spare nodes. This will already provide benefits in improving resource availability and utilization. For example, Rezaei and Mueller [4] have inferred that a dynamic pool of nodes can reduce the required number of spare nodes by an order of magnitude as compared to attaching spare nodes to each job. However, the technique proposed in this paper goes one step further and takes advantage of the *dynamic* nature of jobs to provide fast node replacements and improve throughput.

As defined by Feitelson and Rudolph [5], jobs can be classified in four categories based on their flexibility. The first type is the *rigid* job, which requires a fixed number of processors throughout its execution. The second type is called the *moldable* job, whose resource set can be *molded* or *modified* by the batch system before starting the job (e.g., to effectively fit alongside other rigid jobs). Once started, its resource set cannot be changed anymore. The third type is called the *evolving* job, which requests expansion or shrinkage of its resource allocation during job execution. The fourth class of jobs is called the *malleable* job. In contrast to evolving jobs, the expansion and shrinkage of the resources belonging to a malleable job are initiated by the batch system. The application adapts itself to the changing resource set. While rigid jobs are most common in today's HPC environments, the other job types are slowly becoming more visible since many programming models such as MPI, OpenMP, Charm++ and OmpSs are starting to feature dynamic runtime systems. We have earlier demonstrated how to create and schedule malleable and evolving jobs using Charm++ and MPI, respectively [6], [7]. Similarly, how malleable jobs can be created using MPI has been demonstrated by Utrera et al. [8] and El Maghraoui et. al. [9]. Moldable and malleable jobs can already be seen in cloud environments. Therefore, as dynamic usage scenarios begin to emerge in HPC environments, it is important to both effectively manage them as well as exploit their flexibility. Our proposed solution proficiently uses the adaptability of these job types to enable fast replacements. A high level illustration is shown in Figure 1(b). The main advantages of our solution are:

1) **No resubmission overhead**: Instead of resubmitting and restarting a job, it can be paused until failed nodes are replaced.
2) **More efficient use of resources**: Avoids resource wastage at the expense of a user's compute-time budget by eliminating the need for allocating spare nodes for each job.
3) **Less full job restarts required**: Under static allocation, the job has to be restarted completely after every node failure when there are no statically allocated spare nodes available. With dynamic node replacement, a partial restart of a job could be sufficient, thereby reducing the time taken for bringing the application back to the running state.
4) **Better use of in-memory checkpoints and reduced frequency of disked checkpoint/restart**: Single-node and multi-node failures (up to a certain number limited by the checkpointing method) can be recovered with in-memory checkpoints.

Thus, dynamic node replacement is an important aspect for fast application recovery in current and future systems. The focus of this work is restricted to the functionality of the batch system and we show that proposed method can maintain high throughput even under high failure rates. The remainder of this paper is organized as follows: Section II discusses other related work. Section III details the dynamic node replacement algorithm, after which Section IV describes the implementation and evaluation setup. Section V provides a detailed discussion of the results. Section VI concludes the paper.

II. Related Work

Although fault tolerance and resilience have been an active area of research for a long time, they have gained practical momentum in recent years as HPC progresses to exascale. Exascale systems are predicted to have high failure rates, which is motivating the development of many methods for enabling a robust and reliable cluster environment.

In a broad perspective, failures can be categorized into two types: software and hardware failures. Software failures include the sudden death of important middleware such as the central batch system, control daemons on nodes, and runtimes and libraries used by the application. Batch systems typically write cluster and job state information

to file systems. This includes running and queued jobs, resources currently used by jobs, and future reservations and ensures that a batch system can be restarted without losing the information on the state of the jobs.

Tolerance to hardware failures involves multiple aspects: fault detection, resource monitoring, and application recovery. Current production batch systems are already equipped with effective tools for resource monitoring and fault detection. These tools are also constantly improved for faster detection and low overhead monitoring. However, application recovery is a challenging aspect. The most popular approach to application recovery is through checkpoint/restart. The application is check-pointed regularly on disk storage and restarted from the latest checkpoint in the event it is affected by hardware failure. Some of the commonly used checkpoint/restart frameworks include BLCR [10], FTI [2] and SCR [3]. Since current batch systems only support static allocations, the saved job is first removed from the existing allocation and restarted on a fresh allocation. Batch systems such as Platform LSF [11], OAR [12], Moab HPC Suite [13], and HTCondor [14] follow this method. Also, since most batch systems do not support job types other than rigid jobs, failed jobs often have to wait a considerable amount of time before a fresh allocation can be provided. The SLURM resource manager contains the prototypical implementation of a node-replacement facility. However, since it does not support malleable or evolving jobs, the waiting time for node replacement is not improved compared to providing a fresh allocation.

Research in fault tolerance and resiliency mostly focused on improved application recovery techniques such as enhanced checkpoint/restart methods [15], [2], transparent and proactive process migration [16], [17], process replication [18], [19] and algorithm-based fault tolerance [20], [21]. Batch systems are usually coupled with checkpoint/restart frameworks and tools that perform process replication and migration to ensure transparent recovery [22] [23]. In this paper, we explore a new dimension on fault tolerance through dynamic node replacement.

III. Dynamic Node-Replacement Algorithm

The node-replacement algorithm is designed to be a supplement to a main job scheduling algorithm, which we call the *base scheduling algorithm*. The node-replacement algorithm is invoked only in the event of node failure. Any scheduling algorithm can be used as the base scheduling algorithm. However, it is assumed that the base scheduling algorithm is also capable of scheduling moldable and malleable jobs.

As an example, we use as base scheduling algorithm a combination of FCFS with backfilling, DBES [6] for scheduling malleable jobs, and Supercomputer AppLeS (SA) [24] for scheduling moldable jobs. The DBES algorithm schedules malleable jobs first with their minimum set of required resources and later expands them with available idle resources. It consists of a two stage expansion process. In the the first stage, dependencies among all current and future job reservations are computed and those malleable jobs that are likely to allow queued jobs to be started earlier are expanded. In the second stage, remaining idle resources are equally distributed among the running malleable jobs. More information about the algorithm has been described by Prabhakaran et al. [6]. The SA algorithm simulates the various configurations possible for each job submission and estimates the turnaround time for each job using a resource allocation list and a submit-time greedy strategy based on the current system state. The scheduler then allocates processors to jobs that are expected to deliver the least turnaround time. Note that DBES and SA are selected for demonstration purposes only. The node-replacement algorithm is agnostic to the choice of base scheduling algorithm and therefore any moldable/malleable job scheduling algorithms could be used instead. The node-replacement algorithm's only responsibility is to assign replacement nodes to jobs affected by failure. For the sake of simplicity, we make the following assumptions:

Relation to checkpointing: Dynamic node-replacement is only worth when the application is checkpointed regularly and can be continued from a checkpoint after the failed nodes are replaced. However, an application can have restrictions of its checkpointing intervals or may employ different checkpointing strategies such as in-memory checkpointing and multi-level hierarchical checkpointing. In general, our node-replacement algorithm can be combined with any checkpoint/restart method available. Therefore, when an application affected by failure gets an instant node replacement, it may have to rollback to a previous checkpoint and restart from there. This means the application may have to recompute parts of what has already been computed but is now lost because it was not checkpointed. In such situations, the wall time limit of the job needs to be extended accordingly to allow for those re-computations. This time may vary between applications and the scheduler may compute the extension by subtracting the time of the last checkpoint from the time at which the failure occurred. For simplicity, we assume perfect checkpointing of applications, that is, an application can restart from exactly the same point at which it experienced the node failure. At the same time, we claim that imperfect checkpointing could be handled in principle, while emerging technologies such as non-volatile memory are expected to lower the degree of imperfection as time progresses.

Checkpointing different job types: As practiced today, we assume that rigid jobs can only be restarted from a checkpoint with the same total number of processes present at the point of the last checkpoint. That is, if a rigid job using n processes was checkpointed at a certain point in time, it can be restarted from that checkpoint only when

it is possible to run it again with n processes. The same applies to moldable jobs, as by definition a moldable job becomes rigid after it starts executing with a given number of processes. However, the only exception are malleable jobs. They can be checkpointed and restarted with different number of processes, given their malleable nature. For example, Charm++ applications can behave in this manner using in-memory checkpointing scheme.

Replacement waiting period: When a job affected by node failure has to go through a waiting period until nodes are replaced caused by the scheduler, the job's wall time limit is extended by this waiting time.

When the node-replacement algorithm is invoked, it cancels all job reservations made for the future and treats the affected jobs with highest priority. The scheduler collects the list of jobs that are affected by failure and attempts to replace the failed nodes for each affected job through the following options, in the order of presentation.

1) Local shrink
2) Use of idle processors
3) Remote shrink
4) Restarting moldable jobs
5) Waiting for processors to be come idle

1) Local shrink: The first option is considered only for malleable jobs, wherein the scheduler attempts to shrink the job and continue execution by removing the failed nodes out of the job's allocation. This can be successful only if at least the minimum number of nodes to execute the job remain allocated to it after it has been shrunk. This avoids replacement while maintaining the conditions requested during the malleable job's submission, and therefore is a faster solution. The base scheduling algorithm can later expand the job again.

2) Use of idle processors: When the first option cannot be applied, the scheduler attempts to replace the failed nodes with idle ones as the second option. Idle resources may be found from the cluster's regular partition or a spare pool of nodes as described in Section I, with priority given to the spare pool.

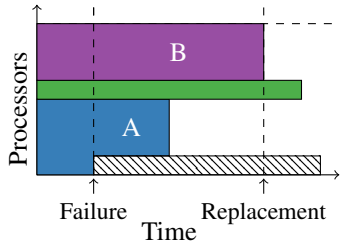
3) Remote shrink: When the first two options are unsuccessful, as the third option the scheduler considers replacing failed nodes with healthy nodes obtained by shrinking running malleable jobs. Malleable jobs that have been previously expanded with additional nodes are considered for this purpose. Note that the algorithm can be configured to prioritize certain malleable jobs over others when selecting the malleable jobs to be shrunk. This can be based on simple algorithms to sort running malleable jobs such as earliest expanded first, latest expanded first, and frequency of resource expansion/shrink. It can also be configured with more sophisticated methods to complement the malleability decisions of the base scheduling algorithm. For example, when DBES is used as the base malleable scheduling algorithm, priority of jobs selected for shrinking follows the order of malleable jobs expanded in the second stage of DBES (where resources are equipartitioned among malleable jobs) followed by malleable jobs expanded in the first stage of DBES (where resources are distributed to malleable jobs based on the dependencies in the reservation flow). This helps to maintain the malleability decisions made by DBES to improve the general throughput of the system. The jobs shrunk during this step may later be expanded again by the base scheduling algorithm if and when sufficient nodes become available. Note that during this step of the node-replacement algorithm, a combination of option 3 and option 2 is also considered.

4) Restarting moldable jobs: The fourth option considers two special cases only encountered with moldable jobs. In the first case, the affected job itself is a moldable job, which can be restarted from the beginning with a valid lower number of nodes – typically specified by the user during job submission. During failure, the latest point in time at which sufficient replacement nodes can be availed for the affected job can be determined using the job wall-time information. Depending on this waiting time and the runtime estimates of the moldable job's request alternatives, restarting the moldable job with a lower number of nodes can lead to an earlier time of completion than waiting for replacement. Figure 2 exemplifies this scenario where a moldable job A is affected by a node failure. The algorithm restarts the affected moldable job on a reduced set of nodes rather than waiting for job B to terminate, leading to an earlier completion time.

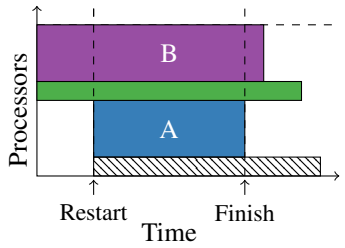
However, this approach may not yield the best result in certain situations. For example, if the affected moldable job has almost reached its completion, although restarting the job on a lower number of nodes may provide a better throughput than waiting for a replacement, it drastically increases the turnaround time of the job. Therefore, for such scenarios and for scenarios where the affected job is not a moldable job, the second case is considered, which attempts to restart other running moldable jobs with a lower number of nodes in order to free up nodes for replacement. This is illustrated in Figure 3. Job A is affected by node failure when it has almost reached the wall time. However, its waiting time for replacement is long in contrast to the extra time which job B, which just started, would need if it was restarted on a reduced set of processors. The algorithm therefore decides to restart job B. Thus, the problem here is to determine the set of moldable jobs that must be restarted to serve the recovery needs of the affected job while minimizing the cost to free enough nodes. This is formally defined as follows. We define a variable $x_{j,c}$ for every job $j \in J$ and their allowed job sizes $c \in C_j$ such that

$$x_{j,c} = \begin{cases} 1 & \text{if job } j \text{ should restart on } c \text{ processors} \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

The cost of restarting job j on c processors in terms of the

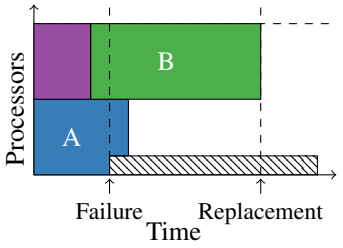


(a) Job A is affected by a node failure. The earliest time for a possible replacement is when job B terminates.

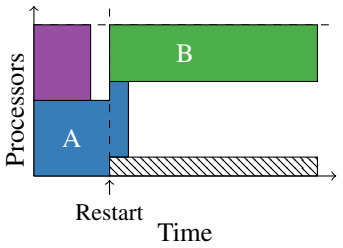


(b) The algorithm decides to restart job A on a smaller resource set, which leads to a completion time that is earlier than if it had waited for job B to terminate.

Figure 2: An example for the local restart of a moldable job.



(a) Job A is affected by a node failure. The earliest time for a possible replacement is when job B terminates.



(b) The algorithm decides to restart job B on a smaller resource set as its delay in completion time is smaller than if job A had waited for job B to terminate.

Figure 3: An example for restarting a remote moldable job.

delay in the completion of the job is given by:

$$r_{j,c} = x_{j,c}(e_j(c) - \bar{s}_j), \tag{2}$$

where $e_j(c)$ gives the completion time of job j when restarted instantly on c processors and $\bar{s}_j$ is the guarantee given by the scheduler for the completion time of job j when it was started for the very first time. Since the main aim is to minimize the delay in completion time for every job, this definition prevents the same job being considered multiple times for restart. When the current job size is a_j for every job $j \in J$ and the number of processors needed for the failed job to recover is d, the values for $x_{j,c}$ can be obtained by solving the following integer programming problem:

$$\begin{aligned} \min_{x_{j,c}} & \sum_{j \in J} \sum_{c \in C_j} x_{j,c}(e_j(c) - \bar{s}_j) \qquad (3) \\ \text{such that} \quad & \sum_{j \in J} \sum_{c \in C_j} x_{j,c}(a_j - c) \geq d \\ \text{where} \quad & \sum_{c \in C_j} x_{j,c} \leq 1 \quad \forall j \in J \\ \text{and} \quad & x_{j,c} \in \{0,1\} \quad \forall j \in J, c \in C_j \end{aligned}$$

Note that J consists of only all running moldable jobs including the affected job if it is moldable. Also, C_j does not have to include a_j for every $j \in J$. Job j is not restarted when $x_{j,c}$ is zero for every $c \in C_j$. The integer programming problem can then be solved using, for example, a branch-and-bound algorithm. This provides a solution $x_{j,c}$ for every job. The dynamic node-replacement algorithm considers restarting jobs for which $x_{j,c} = 1$ when the delay in completion time of the failed job is less than the time that it would spend waiting for processors to be released by normal termination of other running jobs.

5) Waiting for processors to become idle: Finally, if all of these options are not able to provide an efficient replacement to the affected job, the fifth and the last option is considered, which is waiting for the completion of other jobs to acquire resources. In this case, the affected job is put at the top of the queue of pending jobs. As soon as nodes are available, they are assigned to the affected job.

IV. Implementation and Evaluation Setup

In this section, we present the evaluation environment used to study the dynamic node-replacement algorithm.

A. Implementation

The dynamic node-replacement algorithm and all base job scheduling algorithms used were implemented and evaluated on the custom discrete-event simulator based on the SimJava2.0 package [25]. In a discrete-event simulation, the state of a system changes at discrete points in time according to the occurrence of certain *events*. In this context,

the events that change the state of our system are: job arrival, job completion, node failure, and node recovery. A set of variables defines the state of the system at a given point in time. The variables of importance to this context are the job queue, the list of jobs currently in execution, and the list of jobs that are affected by node failure. Each event is processed with the respective action after updating the state of the system. For example, a node failure event moves all jobs running on a failed node to the list of affected jobs. This is followed by a call to the dynamic node-replacement algorithm to attend to the affected jobs. The base resource mapping and backfilling algorithm is used from the implementation provided by the GridSim toolkit [26]. For solving the linear programming problem defined in the node-replacement algorithm, we use the GNU Linear Programming Kit [27].

B. Workload model

Although there are many real workloads and workload models publicly available [28], they only reflect rigid jobs. We require workload models that already include or allow the integration of moldable and malleable jobs. Therefore, we chose two models for rigid and moldable jobs proposed by Cirne [24]. Malleable jobs were generated by extending the moldable job model. For the purpose of evaluation, we generate workloads of the required size and type composition from the SDSC workload trace [24] for a given system size. The jobs in the workload are characterized by the job and speedup models explained below.

1) Speedup model: Since our methods use moldable and malleable jobs, it was essential to use a speedup model to capture the flexibility of the jobs according to their types. Downey proposed a speedup model that constructs a hypothetical parallelism profile to reflect the behavior of common parallel applications [29]. This was used to create the speedup profiles of all rigid, moldable, and malleable jobs.

2) Rigid job model: Downey observed that the distribution of job sizes (i.e., number of processors used by a job) and the estimated runtime follows approximately a log-uniform distribution [29], which was later confirmed by Cirne [24]. The cumulative distribution function of a log-uniform distribution is given by:

$$\mathrm{CDF}(x) = \chi \log_2(x) + \rho, \tag{4}$$

where χ and ρ are the slope and the intercept of the line in the log space. The summary of the parameters describing a rigid job is shown in Table I

3) Moldable job model: A moldable job consists of a set of a rigid requests with each request having a job size n, a runtime $T(n)$, and a runtime estimate or wall time $T_e(n)$. For each job size, the speedup model can be used to determine the job runtime. To determine the maximum size

Table I: Summary of parameters for the rigid job model [24].

Characteristic	Model	Parameters
Job size	Log-uniform distribution	$\chi = 0.12$, $\rho = 0.20$
Power-of-2 job sizes	Probability p	$p = 0.75$
Estimated runtime	Log-uniform distribution	$\chi = 0.10$, $\rho = -0.75$

Table II: Summary of parameters for the moldable job model [24].

Characteristic	Model	Parameters
Minimum job size	Log-uniform distribution	$\chi = 0.06920$, $\rho = 0.6279$
Number of user requests	Log-uniform distribution	$\chi = 0.1918$, $\rho = 0.1876$
Average parallelism	Joint log-uniform distribution	$\varphi = 0.009548$, $\gamma = -0.01877$ $\eta = 0.07468$, $\rho = -0.009198$

c_{max} of a job, Downey's model was used and is given by:

$$c_{max} = \begin{cases} 2A - 1 & \text{if } \sigma \leq 1 \\ A + A\sigma - \sigma & \text{if } \sigma \geq 1. \end{cases} \tag{5}$$

where A is the average parallelism and σ is the coefficient of variance in parallelism. The speedup model is applied to each using the relationship between c_{min} and A, which follows a joint log-uniform distribution given by:

$$\mathrm{CDF}(x, y) = \varphi \cdot \log_2(x) \cdot \log_2(y) + \gamma \cdot \log_2(x) + \eta \cdot \log_2(y) + \rho. \tag{6}$$

Table II shows the specific values of the parameters used in our evaluation.

4) Malleable job model: Malleable jobs are submitted with a range of processors within which the job is assumed to be malleable. For describing these jobs, the same parameters of the moldable job model explained above are used. Additionally, instead of allowing only a set of job sizes within the minimum and the maximum job-size range, a malleable job can accept all the sizes within that range.

C. Failure model

A failure model describes the occurrence of failures in a supercomputer. This can be modeled by studying a variety of failure traces that are publicly available [30]. A study by Schroeder and Gibson [31] showed that it is difficult to describe a system after the first year of installation using a common probability distribution. The remaining years can be described using a Weibull distribution. Schroeder and Gibson observed that the mean time to repair can be well described by a log-normal distribution. For the purpose of evaluation, we set the MTBF of the system to 1 hour, which has been estimated to be typical for an exascale system [32] and the mean time to repair is about 2.8 hours, following the above distributions.

Table III: Composition of workloads.

Workload name	Job types		
	Rigid	Moldable	Malleable
Mixed workload	400	300	300
Moldable-rigid workload	500	500	-
Malleable-rigid workload	500	-	500

V. Experimental Results

For the purpose of evaluation, we simulate a cluster computer with a size of 125 nodes with 4 processors per node, reaching a total of 500 processors. Three different workloads, which are listed in Table III, are investigated. As mentioned earlier, the base scheduling algorithm is a FCFS scheduler with support for moldable jobs, malleable jobs and backfilling. Moldable jobs are scheduled using the SA scheduling algorithm. Malleable jobs are scheduled with the algorithms DBES, Precedence to Running Applications (PRA) and Precedence to Waiting Applications (PWA) in separate experiments to draw a comparison. PRA and PWA are malleable job scheduling algorithms proposed by Buisson et. al. [33]. PRA expands all running malleable jobs whenever idle nodes are available in the system instead of using them to start new jobs or perform backfilling. PWA uses idle nodes to start new jobs rather than expanding running malleable jobs. The nodes that still remain idle after this step are used to expand malleable jobs. We reiterate that the simulation assumes perfect checkpointing, as already explained in Section III. The composition of workloads used for evaluation is summarized in Table III.

A. Mixed workload

The mixed workload consists of 400 rigid jobs, 300 moldable jobs and 300 malleable jobs. The base scheduling algorithm used for scheduling the workload includes FIFO with conservative backfilling for rigid jobs and SA for moldable jobs. The malleable jobs were scheduled using PRA, PWA, and DBES in separate experiments. Figure 4 shows the makespan of the workload scheduled with the above mentioned algorithms under different failure conditions. The rigid scheduling strategy treats all jobs as rigid jobs with FIFO and backfill scheduling.

In the absence of node failures, one would expect that scheduling the workload with the awareness of malleable and moldable jobs will result in a lower makespan than scheduling all of them as rigid jobs. It can be noticed that while this is true for PWA and DBES, PRA has a larger makespan than even rigid scheduling. This is a result of prioritizing the use of idle processors for expanding malleable jobs instead of starting new ones. While expanding a malleable job can improve its speedup, it may not always deliver the best parallel efficiency. Thus the makespan in this cases increases more than even that of rigid scheduling as the execution of several malleable jobs had lower parallel efficiency. Between PWA and DBES, DBES performs negligibly better than PWA.

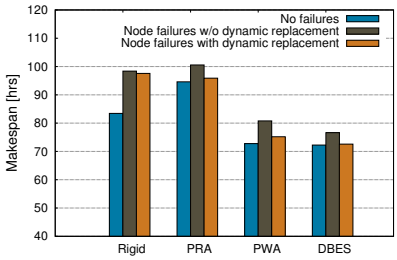


Figure 4: Time for completion of the mixed workload with various scheduling algorithms.

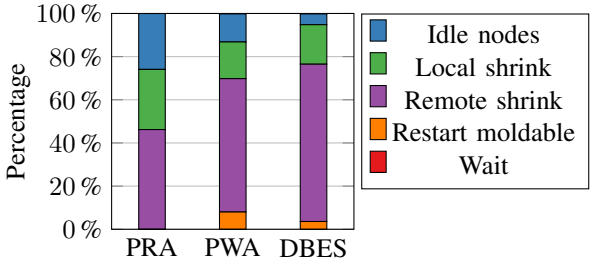


Figure 5: Node replacement sources in the mixed workload.

In the presence of failures and without using dynamic node replacement, the makespan of all the scheduling strategies increases as expected because of the regular node failures. The makespan of the rigid scheduling strategy in the presence of node failures shoots up substantially by about 18% when compared to the makespan of the same without node failures. On the other hand, the makespan of PRA, PWA and DBES in the presence of node failures shoot up only by about 6%, 11% and 5% compared to their own makespan without node failures, respectively. This is because of malleable and moldable jobs that can use resources flexibly and cause less wastage of resources even as failures occur. This shows that even when experiencing a high failure rate, simply enabling moldable and malleable job scheduling can bring about better system utilization and throughput than rigid scheduling.

When the dynamic node replacement algorithm is used, the makespan of adaptivity-aware scheduling strategies is further reduced to stay close to that of the scenario without node failures. Therefore, the dynamic node replacement algorithm can immensely improve the scheduling performance that sees a dip due to hardware failures.

Figure 5 shows the break down of the percentages of the sources from which replacements were found when the dynamic node replacement algorithm was used under PRA, PWA, and DBES. We can observe that the majority of the replacement nodes were found by shrinking other running malleable jobs under all three strategies. This percentage is considerably higher than the percentage of replacement nodes found from idle ones. This is a result of enabling malleability, which increases system utilization and leaves

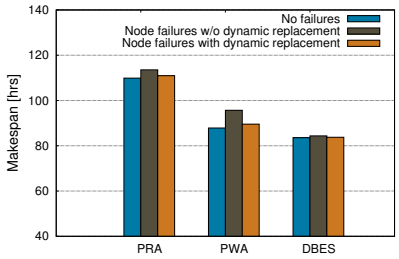


Figure 6: Time for completion of the malleable-rigid workload with various scheduling algorithms.

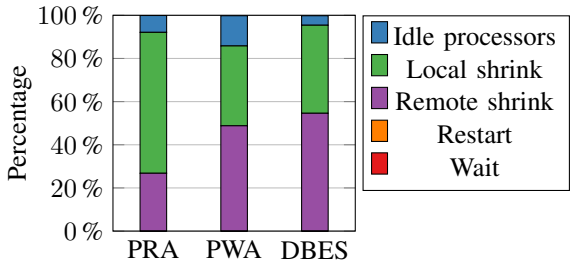


Figure 7: Node replacement sources in the malleable-rigid workload.

less nodes idle. The percentage of idle nodes used for replacement is highest in PRA compared to PWA and DBES because of the relatively low system utilization maintained by PRA in comparison to PWA and DBES.

An important observation is the role of moldable jobs and the local/remote restart of moldable jobs by the dynamic node replacement algorithm. We can see from Figure 5 that the percentage of moldable jobs restarted to find replacements for handling a failure has been far less compared to other replacement sources. Restarting a moldable job is typically an expensive operation as the job has to start from the beginning. Therefore, the lower the number of moldable jobs restarted is, the better is the overall performance. However, this step in the dynamic node replacement algorithm was also vital. It effectively reduced the number of jobs that needed to be re-queued as a result of not having found any replacement to zero in all three strategies. For example, when the step for considering the restart of moldable jobs was disabled in the dynamic node replacement algorithm used with DBES as base, 8 jobs were re-queued out of the 220 jobs that were affected by failures. Thus, although its contribution in finding replacements is lower compared to other replacement sources, it plays an essential role. Also, it has the potential to make a larger contribution in the future as checkpoint/restart techniques mature as detailed in Section V-C.

B. Malleable-rigid workload

The malleable-rigid workload consists of a total of 1000 jobs with 500 malleable jobs and 500 rigid jobs. The makespans of this workload with the PRA, PWA and DBES

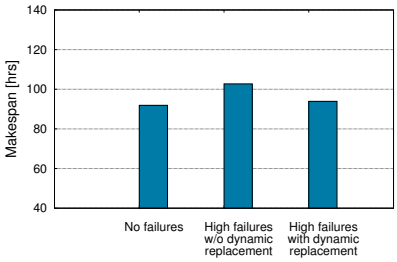
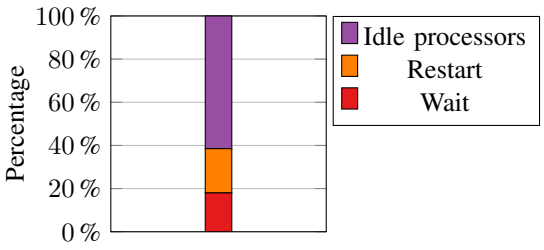


Figure 8: Time for completion of the moldable-rigid workload in various scenarios.



SA with Dynamic Replacement

Figure 9: Node replacement sources in the moldable-rigid workload.

schedulers and various failure scenarios are shown in Figure 6. The corresponding sources of replacement are shown in Figure 7. It can be observed that the results follow the same trend as the mixed workload. Without the presence of failures PRA has the longest makespan and DBES has the shortest. Similar to the mixed workload, the presence of node failures increase the makespan with all three base scheduling algorithms. When the dynamic node replacement algorithm is applied, the makespan is brought down close to the original makespan without node failures.

However, the most important observation in this workload is that the makespan with the DBES scheduling algorithm is almost the same in all three cases: absence of node failures, high node failure rate without dynamic node replacement, and high node failure rate with dynamic node replacement. This is because DBES performs an implicit dynamic node replacement. When a job is affected by node failures and no dynamic node replacement algorithm is present, the job is put back into the idle/pending queue with the highest priority. Therefore, in the next scheduling iteration, DBES can already allocate resources for this job through its basic scheduling mechanism. This is done by either providing idle healthy nodes or by obtaining them from shrinking other running malleable jobs. This is similar to the steps followed by the dynamic node replacement algorithm.

C. Moldable-rigid workload

The moldable-rigid workload consists of 500 rigid and moldable jobs each, totaling to 1000 jobs. As already

described, moldable jobs are scheduled using the SA algorithm. Figure 8 shows the makespan of the workload in the three failure scenarios: no failures, high failure rate without dynamic node replacement, and high failure rate with dynamic node replacement.

The moldable-rigid workload also exhibits the same performance trend as the mixed and malleable-rigid workload. The makespan of the workload when subjected to a high node failure rate increases by about 11% compared to having no hardware failures. However, with the dynamic node replacement strategy, the makespan is only 2% greater than that of the scenario without any failures. The split-up of the sources of replacement when using the dynamic node replacement algorithm is shown in Figure 9.

A high percentage (62%) of instant replacements came from idle processors due to the low system utilization (65%) obtained when scheduling this workload. However, continued node failures did not allow instant replacements all the time and some jobs had to wait until nodes were released by the normal termination of other running jobs. This constituted about 18% of the node replacements. Approximately 20% of instant replacements were provided by restarting moldable jobs (step 4 of the dynamic node replacement algorithm).

Thus, almost an equal percentage of replacements had been provided by restarting moldable jobs and waiting for node releases. The strategy of restarting moldable jobs alone could not completely avoid the waiting. This is typical because restarting moldable jobs is an expensive operation. Unlike malleable jobs that can seamlessly continue execution after a shrink or an expand step, moldable jobs have to be restarted from the beginning of the application. The results of the execution performed before terminating a moldable job cannot be used after restarting it with a different number of processors. Therefore, this strategy makes only a smaller contribution towards finding instant node replacements as opposed to using malleable jobs or idle resources for finding replacements.

However, this strategy can be expected to play a more prominent role in the future as checkpoint/restart techniques advance. Current disk-based checkpoint/restart techniques are only able to restart a job with the same number of processes from a checkpoint. On the other hand, restarting jobs with a different number of processors from a checkpoint is one of the important research directions in checkpoint/restart techniques and is not far from being a reality. Since moldability is easier to achieve than malleability, this replacement strategy will be able to take advantage of the improved checkpoint/restart methods and the larger number of moldable jobs that may be present in future workloads.

VI. Summary and Conclusion

Although the computational demands of user applications drive the effort to create exascale systems, high computational power at the cost of reliability is not desired by any user. Given that exascale systems are predicted to have a MTBF of one hour or less, achieving high resiliency is one of the top goals in making exascale systems real. Until now, applications could use the libraries for checkpoint/restart or replication separately for fault tolerance. However, moving forward, exascale systems call for an approach of tightly coupling multiple middleware components to achieve resiliency.

To this end, this work proposes a dynamic node replacement algorithm which replaces failed nodes of a job with healthy ones on-the-fly. The algorithm uses the unique characteristics of all job types to find fast replacement nodes. The algorithm is implemented as a supplemental algorithm to a base scheduling algorithm and is triggered in the event of node failures. Evaluation of the algorithm with GridSim simulator shows that dynamic node replacement has a strong potential to curb the throughput loss that inevitably occurs when experiencing high failure rates.

The dynamic node replacement algorithm is a perfect complement to the new multi-level checkpointing features that are increasingly becoming popular. While this contribution used a simulator to demonstrate the benefits of the algorithm, it opens the scope for building a wide variety of interfaces between checkpoint/restart frameworks and the batch system to achieve functionality such as automatic checkpointing and proactive migration. Also, the batch system could use the checkpoint/restart framework to achieve pseudo-malleability in applications if they can be restarted from a checkpoint with a different number of processors. Overall, the role of batch systems in future cluster environments will be key to making systems more robust.

Acknowledgement

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 720270 (HBP SGA1).

References

[1] G. Zheng, X. Ni, and L. Kalé, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops*, June 2012, pp. 1–6.

[2] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: high performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 32.

[3] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proc. of IEEE/ACM SC'10*, 2010.

[4] A. Rezaei and F. Mueller, "Sustained resilience via live process cloning," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1498–1507.

[5] D. G. Feitelson and L. Rudolph, "Towards Convergence in Job Schedulers for Parallel Supercomputers," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1996.

[6] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kalé, "A Batch System with Efficient Scheduling for Malleable and Evolving Applications," in *Proc. of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Hyderabad, India*. IEEE Computer Society, May 2015, pp. 429–438. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2015.34

[7] S. Prabhakaran, M. Iqbal, S. Rinke, C. Windisch, and F. Wolf, "A Batch System with Fair Scheduling for Evolving Applications," in *Proc. of the 43rd International Conference on Parallel Processing (ICPP), Minneapolis, MN, USA*, Sep. 2014.

[8] G. Utrera, J. Corbalan, and J. Labarta, "Implementing Malleability on MPI Jobs," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.

[9] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela, "Malleable Iterative MPI Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, 2009.

[10] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for Linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006.

[11] Platform LSF. http://www-03.ibm.com/systems/platformcomputing/products/lsf/. Accessed: 2017-04-07.

[12] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux, "Supporting Malleability in Parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI," in *Proc. of the 11th International Conf. on Distributed Computing and Networking*. Springer-Verlag, 2010.

[13] "Moab hpc basic edition," http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/, accessed: 2016-01-29.

[14] HTCondor. http://research.cs.wisc.edu/htcondor/. Accessed: 2017-04-07.

[15] N. Abeyratne, H.-M. Chen, B. Oh, R. Dreslinski, C. Chakrabarti, and T. Mudge, "Checkpointing exascale memory systems with existing memory technologies," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. ACM, 2016.

[16] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive Fault Tolerance Using Preemptive Migration," in *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 252–257.

[17] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in hpc environments," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 43:1–43:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413414

[18] H. Casanova, F. Vivien, and D. Zaidouni, *Using Replication for Resilience on Exascale Systems*. Springer International Publishing, 2015, pp. 229–278.

[19] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, "Combining Process Replication and Checkpointing for Resilience on Exascale Systems," INRIA, Research Report RR-7951, May 2012.

[20] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410 – 416, 2009.

[21] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas, "Building algorithmically nonstop fault tolerant mpi programs," in *18th International Conference on High Performance Computing*, 2011, pp. 1–9.

[22] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1393–1404, 2014.

[23] H. Lee, D. Park, M. Hong, S.-S. Yeo, S. Kim, and S. Kim, "A Resource Management System for Fault Tolerance in Grid Computing," in *International Conference on Computational Science and Engineering*, 2009.

[24] W. Cirne and F. Berman, "Using moldability to improve the performance of supercomputer jobs," *Journal of Parallel and Distributed Computing*, vol. 62, no. 10, pp. 1571–1601, 2002.

[25] T. U. of Edinburgh, *SimJava2.0*, 2015 (accessed 09.09.2015), http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/.

[26] R. Buyya and M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *Concurrency and Computation: Practice and Experience*, 2002.

[27] G. project. GNU Linear Programming Kit. https://www.gnu.org/software/glpk/. Accessed: 2016-01-29.

[28] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with using the parallel workloads archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967–2982, 2014.

[29] A. B. Downey, "A parallel workload model and its implications for processor allocation," *Cluster Computing*, vol. 1, no. 1, pp. 133–145, 1998.

[30] B. Javadi, D. Kondo, A. Iosup, and D. Epema, "The failure trace archive: Enabling the comparison of failure measurements and models of distributed systems," *Journal of Parallel and Distributed Computing*, 2013.

[31] B. Schroeder, G. Gibson *et al.*, "A large-scale study of failures in high-performance computing systems," *Transactions on Dependable and Secure Computing*, 2010.

[32] F. Cappello, A. Geist, W. Gropp, S. Kalé, B. Kramer, and M. Snir, "Toward Exascale Resilience: 2014 Update," 2014.

[33] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema, "Scheduling malleable applications in multicluster systems," in *IEEE International Conference on Cluster Computing*, Sept 2007.