2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Assessing the impact of ABFT & Checkpoint composite strategies

George Bosilca[1], Aurelien Bouteiller[1], Thomas Herault[1],
Yves Robert[1,2] and Jack Dongarra[1]
1. University of Tennessee Knoxville, USA
2. Laboratoire LIP, Ecole Normale Supérieure de Lyon, France
{bosilca|bouteill|dongarra|herault}@icl.utk.edu, yves.robert@ens-lyon.fr

***Abstract*—Algorithm Based Fault Tolerant (ABFT) approaches promise unparalleled scalability and performance in failure-prone environments. With the advances in the theoretical and practical understanding of algorithmic traits enabling such approaches, a growing number of frequently used algorithms (including all widely used factorization) have been proven ABFT-capable. In the context of larger applications, these algorithms provide a temporal section of the execution when the data is protected by it's own intrinsic properties, and can be algorithmically recomputed without the need of checkpoints. However, while typical scientific applications spend a significant fraction of their execution time in library calls that can be ABFT-protected, they interleave sections that are difficult or even impossible to protect with ABFT. As a consequence, the only fault-tolerance approach that is currently used for these applications is checkpoint/restart. In this paper we propose a model to investigate the efficiency of a composite protocol, that alternates between ABFT and checkpoint/restart for effective protection of an iterative application composed of ABFT-aware and ABFT-unaware sections. We validate this model using a simulator. The model and simulator show that this composite approach drastically increases the performance delivered by an execution platform, especially at scale, by providing means to rarefy the checkpoints while simultaneously decreasing the volume of data needed to be checkpointed.**

I. Introduction

As processor count increases with each new generation of high performance computing systems, the long dreaded reliability wall is materializing, and threatens to derail the efforts and milestones on the road toward Exascale computing. Despite continuous evolutions, such as improvements to the individual processor reliability, the integration of a large number of components leads, by simple probabilistic amplification, to a stern decrease in the overall capability of High Performance Computing (HPC) platforms to execute long running applications spanning a large number of resources. Already today, leadership systems encompassing millions of nodes experience a Mean Time Between Failures (MTBF) of a few hours [1], [2], [3]. Even considering an optimistic scenario with "fat" nodes, featuring heavily many-core systems and/or GPU accelerators, projections of Exascale machine exhibit unprecedented socket counts and will thereby suffer in terms of reliability [4].

The high performance community is not without resources to face this formidable threat. Under the already serious pressure that failures pose to currently deployed systems, checkpointing techniques have seen a large adoption, and many production quality software effectively provide protection against failures with application-level rollback recovery. During the execution, periodic checkpoints are taken that capture the progress of the application. When a failure occurs, the application is terminated, but can be later restarted from the last checkpoint. However, checkpointing techniques inflict severe overhead when failure frequency becomes too high. Checkpoints generate a significant amount of I/O traffic and often block the progression of the application; in addition, they must be taken more and more often as the MTBF decreases in order to enable steady progress of the application. Analytical projections clearly show that sustaining Exascale computing solely with checkpointing will prove challenging [5], [6].

The fault-tolerance community has developed a number of alternative recovery strategies that do not employ checkpoint and rollback recovery as their premise. Strategies such as Algorithm Based Fault Tolerance (ABFT) [7], naturally fault tolerant iterative algorithms [8], resubmission in master slave applications, etc., can deliver more scalable performance under high stress from process failures. As an example, ABFT protection and recovery activities are not only inexpensive (typically less than 3% overhead observed in experimental works [9], [10]), but also have a negligible asymptotic overhead when increasing node count, which makes them extremely scalable. This is in sharp contrast with checkpointing, which suffers from increasing overhead with system size. ABFT is a useful technique for production systems, offering protection to important infrastructure software such as the dense distributed linear algebra library ScaLAPACK [9]. In the remainder of this paper, without loss of generality, we will use the term ABFT broadly, so as to describe any technique that uses algorithm properties to provide protection and recovery without resorting to rollback recovery.

However, typical HPC applications do spend some time where they perform computations and data movements that are incompatible with ABFT protection. The ABFT technique, as the name indicates, allows for tolerating failures only during the execution of the algorithm that features the ABFT properties. Moreover, it then protects only the part of the user dataset that is managed by the ABFT algorithm. In case of a failure outside the ABFT-protected operation, all data is lost; in case of a failure during the ABFT-protected operation, only the data covered by the ABFT scheme is restored. Unfortunately, these ABFT-incompatible phases force users to resort to general-purpose (presumably checkpoint based) approaches as their sole protection scheme.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.79

Yet, many HPC applications also spend quite a significant part of their total execution time inside a numerical library, and in many cases, these numerical library calls can be effectively protected by ABFT. We believe that the missing link enabling fault tolerance at extreme scale is the ability to effectively compose broad spectrum approaches (such as checkpointing) and algorithm based recovery techniques, as is most appropriate for different phases within a single application.

Possible target applications are based on iterative methods applied across an additional dimension such as time or temperature. Examples of such applications range from heat dissipation to radar cross-section, all of them being extremely time consuming applications, with the usual execution time for real-size problems ranging from several days to weeks. At the core of such applications, a system of linear equations is factorized, and the solution is integrated into a larger context, across the extra dimension. Upon closer inspection of the execution of such application, it becomes obvious that the most costly step is the factorization. Conveniently, factorization algorithms are some of the first algorithms to be extended with ABFT properties, both in the dense and sparse [11], [12], [13] linear algebra world.

The main contribution of this paper is the design and evaluation of a new composite algorithm that allows for taking advantage of ABFT techniques in applications featuring phases for which no ABFT algorithm exists. We investigate a composition scheme corresponding to the above mentioned type of applications, where the computation alternates between ABFT protected and checkpoint protected phases. This composite algorithm imposes forced checkpoints when entering (and in some cases leaving) library calls that are protected by ABFT techniques, and uses traditional periodic checkpointing, if needed, between these calls. When inside an ABFT-protected call, the algorithm disables all periodic checkpointing. We describe a fault tolerance protocol that allows for switching between fault tolerance mechanisms, and depicts how different parts of the dataset are treated at each stage. Based on this scheme, we provide a performance model and use it to predict the expected behavior of such a composite approach on platforms beyond what is currently possible through experimentation. We validate the model by comparing its predicted performance to that obtained with a discrete event simulator.

The rest of the paper is organized as follows. We start with a brief overview of related work in Section II. Then we provide a detailed description of the composite approach in Section III, and derive the corresponding analytical performance model in Section IV. Section V is devoted to evaluating the approach, comparing the performance of traditional checkpointing protocols with that of the composite approach with realistic scenarios. This comparison is performed both analytically, instantiating the model with the relevant parameters, and in simulation, through an event-based simulator that we specifically designed for this purpose. We obtain an excellent correspondence between the model and the simulations, and we perform a weak-scalability study that demonstrates the full potential of the composite approach at very large scale. Finally, we provide concluding remarks in Section VI.

II. Related work

Both hardware and software errors can lead to application failure. The consequence of such errors can take various forms in a distributed system: a definitive crash of some processes, erroneous results or messages, or, at the extreme, corrupted processes exhibiting malignant behavior. In the context of HPC systems, most memory corruptions are captured by ECC memory or similar techniques, leaving process crashes as the most commonly observed type of failures.

The literature is rich in techniques that permit recovering the progress of applications when crash failures strike. The most commonly deployed strategy is checkpointing, in which processes periodically save their state, so that computation can be resumed from that point when some failure disrupts the execution. Checkpointing strategies are numerous, ranging from fully coordinated checkpointing [14] to uncoordinated checkpoint and recovery with message logging [15]. Despite a very broad applicability, all of these fault tolerance methods suffer from the intrinsic limitation that both protection and recovery generate an I/O workload that grows with failure probability, and becomes unsustainable at large scale [5], [6] (even when considering optimizations such as diskless or incremental checkpointing [16].)

In contrast, Algorithm Based Fault Tolerance (ABFT) is based on adapting the algorithm so that the application dataset can be recomputed at any moment, without involving costly checkpoints. ABFT was first introduced to deal with silent error in systolic arrays [7]. In recent work, the technique has been employed to recover from process failures [17], [10], [9] in dense and sparse linear algebra factorizations [11], [12], [13], but the idea extends widely to numerous algorithms employed in crucial HPC applications. So called *Naturally Fault Tolerant* algorithms can simply obtain the correct result despite the loss of portions of the dataset (typical of this are master-slave programs, but also iterative refinement methods, like GMRES or CG [8], [18]). Although generally exhibiting excellent performance and resiliency, ABFT requires that the algorithm is innately able to incorporate fault tolerance and therefore stands as a less generalist approach. Another aspect that hinders its wide adoption and production deployment is that it can protect *an algorithm* and its dataset, but applications assemble *many* algorithms that operate on different datasets, and which may not all have a readily available ABFT version or employ different ABFT techniques.

To the best of our knowledge, this work is the first to introduce an effective protocol for alternating between generalist (typically checkpoint based) fault tolerance for some parts of the application and custom, tailored techniques (typically ABFT) for crucial, time consuming computational routines.

Many models are available to understand the behavior of checkpoint/restart [19], [20], [21], [22], and thereby to define an optimal checkpoint period. [23] proposes a scalability model to evaluate the impact of failures on application performance. Compared with these works, we include several new key parameters to refine the model. A significant new contribution is to propose a generalized model for a protocol that alternates between checkpointing and ABFT sections. Although most ABFT methods have a complete complexity analysis (in terms of extra-flops, communications incurred

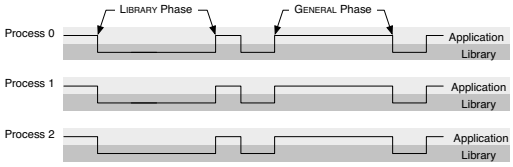


Figure 1: Typical Application

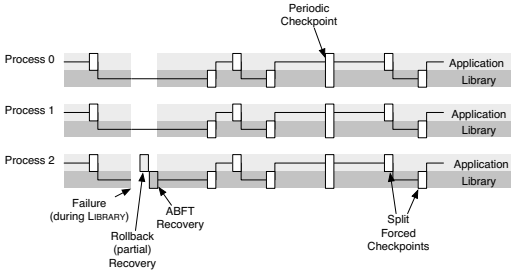


Figure 2: ABFT&PERIODICCKPT composite approach

by both protection activity and per-recovery cost [10], [9]), modeling the runtime overhead of ABFT methods under failure conditions has never been proposed. The composite model captures both the behavior of checkpointing and ABFT phases, as well as the cost of switching between the two approaches, and thereby permits investing the prospective gain from employing this mixed recovery strategy on extreme scale platforms.

III. COMPOSITE APPROACH

We consider a typical HPC application whose execution alternates GENERAL phases and LIBRARY phases (see Figure 1). During GENERAL phases, we have no information about the application behavior, and an algorithm-agnostic fault-tolerance technique, like checkpoint and rollback recovery, must be used. On the contrary, during LIBRARY phases, we know much more about the application, and we can apply special-purpose fault-tolerance techniques, such as ABFT, to ensure resiliency.

During a GENERAL phase, the application can access the whole memory; during a LIBRARY phase, only the LIBRARY dataset (a subset of the application memory, which is passed as a parameter to the library call) is accessed. We call REMAINDER dataset the part of the application memory that does not belong to the LIBRARY dataset. A strong feature of ABFT is that, in case of failure, the ABFT algorithm can recompute the lost ABFT-protected data based only on the LIBRARY dataset of the surviving processors. The main goal of this paper is to compare two fault tolerant approaches:

PUREPERIODICCKPT Pure (Coordinated) Periodic Checkpointing refers to the traditional approach based on coordinated checkpoints taken at periodic intervals, and using rollback recovery to recover from failures.

ABFT&PERIODICCKPT Algorithm-Based Fault Tolerance & Periodic Checkpointing refers to the proposed algorithm, that combines ABFT techniques in LIBRARY phases with Periodic Checkpointing techniques in GENERAL phases. It is described below.

Both approaches use PERIODICCKPT techniques, but to a different extent: while PUREPERIODICCKPT uses PERIODICCKPT throughout the execution, ABFT&PERIODICCKPT uses it only within GENERAL phases of the application.

A. ABFT&PERIODICCKPT *Algorithm*

The ABFT&PERIODICCKPT composite approach consists in alternating between periodic checkpointing and rollback recovery on one side, and ABFT on the other side, at different phases of the execution. Every time the application enters a LIBRARY phase (that can thus be protected by ABFT), a partial checkpoint is taken to protect the REMAINDER dataset. The LIBRARY dataset, accessed by the ABFT algorithm, need not be saved in that partial checkpoint, since it will be reconstructed by the ABFT algorithm inside the library call.

When the call returns, a partial checkpoint covering the modified LIBRARY dataset is added to the partial checkpoint taken at the beginning of the call, to complete it and to allow restarting from the end of the terminating library call. Said otherwise, the combination of the partial entry and exit checkpoints forms a split, but complete, coordinated checkpoint covering the entire dataset of the application.

If a failure is detected while processes are inside the library call, the crashed process is recovered using a combination of rollback recovery and ABFT. ABFT recovery is used to restore the LIBRARY dataset before all processes can resume the library call, as would happen with a traditional ABFT algorithm. The partial checkpoint is used to recover the REMAINDER dataset (everything except the data covered by the current ABFT library call) at the time of the call, and the process stack, thus restoring it before quitting the library routine, see Figure 2. The goal of this strategy is that ABFT recovery will spare some of the time spent redoing work, while periodic checkpointing can be completely de-activated during the library calls.

During GENERAL phases, regular periodic coordinated checkpointing is employed to protect against failures. In case of failure, coordinated rollback recovery brings all processes back to the last checkpoint (at most back to the split checkpoint capturing the end of the previous library call).

B. *Efficiency Considerations and Application-Specific Improvements*

A critical component to the efficiency of the PERIODICCKPT algorithm is the duration of the checkpointing interval. A short interval increases the algorithm overheads, by introducing many coordinated checkpoints, during which the application experiences slowdown, but also reduces the amount of time lost when there is a failure: the last checkpoint is never long ago, and little time is spent re-executing part of the application. Conversely, a large interval reduces overhead, but increases the time lost in case of failure. The PERIODICCKPT protocol has been extensively studied, and good approximations of the optimal checkpoint interval exist (known as Young and Daly's formula [19], [20]). These approximations are based on the machine MTBF, checkpoint duration, and other parameters. We will consider two forms of PERIODICCKPT algorithms: the PUREPERIODICCKPT algorithm, where a single checkpointing interval is used consistently during the whole execution, and

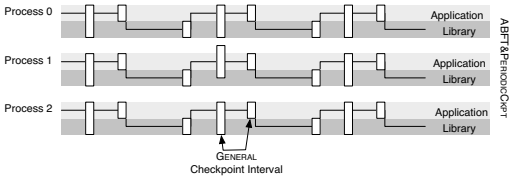


Figure 3: ABFT&PERIODICCKPT composite (time spent in GENERAL phases is large)

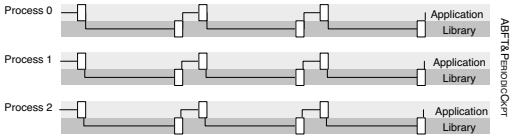


Figure 4: ABFT&PERIODICCKPT composite (time spent in GENERAL phases is small)

the BIPERIODICCKPT algorithm, where the checkpointing interval may change during the execution, to fit different conditions (see Section IV-C, Figures 5 and 6).

In the ABFT&PERIODICCKPT algorithm, we interleave PERIODICCKPT protected phases with ABFT protected phases, during which periodic checkpointing is de-activated. Thus, different cases have to be considered:

- When the time spent in a GENERAL phase is larger than the optimal checkpoint interval, periodic checkpointing is used during these phases in the case of ABFT-&PERIODICCKPT (see Figure 3)
- When the time spent in a GENERAL phase is smaller than the optimal checkpoint interval, the ABFT&PERIODIC-CKPT algorithm already creates a complete valid checkpoint for this phase (formed by combining the entry and exit partial checkpoints), so the algorithm will not introduce additional checkpoints (see Figure 4).

Moreover, the ABFT&PERIODICCKPT algorithm forces (partial) checkpoints at the entry and exit of library calls; thus if the time spent in a library call is very small, this approach will introduce more checkpoints than a traditional PERIODICCKPT approach. The time complexity of library algorithms usually depends on a few input parameters related to problem size and resource number, and ABFT techniques have deterministic, well known time overhead complexity. Thus, when possible, the ABFT&PERIODICCKPT algorithm features a safeguard mechanism: if the projected duration of a library call with ABFT protection (computed at runtime thanks to the call parameters and the algorithm complexity) is smaller than the optimal periodic checkpointing interval, then ABFT is not activated, and the corresponding LIBRARY phase is protected using the PERIODICCKPT technique only.

Furthermore, since only a subset of the entire dataset is modified during a library call (the LIBRARY dataset), incremental checkpointing techniques can benefit PERIODICCKPT approaches. This consists of saving only the subset of the memory that has been modified since the last checkpoint, when taking a new process checkpoint. This influences the duration of the checkpointing operation, and thus the optimal checkpoint interval. In our models, we will take this parameter in consideration.

IV. MODEL

In this section, we detail the application model and the various parameters used to quantify the cost of checkpointing and ABFT protection. Then we analytically derive the minimal overhead for all scenarios. In Section IV-A, we start by defining some parameters, and then proceed in Section IV-B with determining the cost of the composite approach. We compare this cost to that of classical approaches in Section IV-C.

A. Application and checkpoint parameters

The execution of the application is partitioned into epochs. Within an epoch, there are two phases for the application: the first phase is spent outside the library (it is a GENERAL phase), and only periodic checkpointing can be employed to protect from failures during that phase. Then the second phase (a LIBRARY phase) is spent into a compute intensive library routine that has the potential to be protected by ABFT.

Such a scenario is very general, and many scientific applications obey this scheme, alternating phases spent outside and within a library call that can be protected by ABFT techniques. Since each epoch can be analyzed independently, without loss of generality, we focus on a single epoch. Let us introduce some notations. The total duration of the epoch is $T_0 = T_G + T_L$, where T_G and T_L are the durations for the GENERAL and LIBRARY phases, respectively. Let α be the fraction of time spent in a LIBRARY phase: then we have $T_L = \alpha \times T_0$ and $T_G = (1 - \alpha) \times T_0$.

As mentioned earlier, another important parameter is the amount of memory that is accessed during the LIBRARY phase (the LIBRARY dataset). This parameter is important because the cost of checkpointing in each phase is directly related to the amount of memory that needs to be protected. The total memory footprint is M, and the associated checkpointing cost is C. We write $M = M_L + M_{\bar{L}}$, where M_L is the size of the LIBRARY dataset, and $M_{\bar{L}}$ is the size of the REMAINDER dataset. Similarly, we write $C = C_L + C_{\bar{L}}$, where C_L is the cost of checkpointing M_L, and $C_{\bar{L}}$ the cost of checkpointing $M_{\bar{L}}$. We can define the parameter ρ that defines the relative fraction of memory accessed during the LIBRARY phase by $M_L = \rho M$, or, equivalently, by $C_L = \rho C$.

B. Cost of the composite approach

We now detail the cost of resilience during each phase of the composite approach. We start with the intrinsic cost of the method itself, i.e., assuming a fault-free execution. Then we account for the cost of failures and recovery.

1) Fault-free execution: During the GENERAL phase, we separate two cases. First, if the duration T_G of this phase is short (smaller than P_G, defined below), then we simply take a partial checkpoint at the end of this phase, before entering the ABFT-protected mode. This checkpoint is of duration $C_{\bar{L}}$, because we need to save only the REMAINDER dataset before switching modes. Otherwise, if T_G is larger than P_G, we rely on periodic checkpointing during the GENERAL phase: more specifically, the regular execution is divided into periods of

duration $P_G = W + C$. Here W is the amount of work done per period, and the duration of each periodic checkpoint is $C = C_{\bar{L}} + C_L$, because the whole application footprint must be saved during a GENERAL phase. The optimal value of P_G will be computed below. Without loss of generality, we assume an integer number of periods, and the last periodic checkpoint replaces that of size $C_{\bar{L}}$ preceding the switch to ABFT-protected mode.

Altogether, the length T_G^{ff} of a fault-free execution of the GENERAL phase is the following:

- If $T_G < P_G$, then $T_G^{\text{ff}} = T_G + C_{\bar{L}}$
- Otherwise, we have $\frac{T_G}{W}$ periods of length P_G, so that

$$T_G^{\text{ff}} = \frac{T_G}{P_G - C} \times P_G \tag{1}$$

Now consider the LIBRARY phase: we use the ABFT-protection algorithm, whose cost is modeled as an affine function of the time spent: if the computation time of the library routine is t, its execution with the ABFT-protection algorithm becomes $\phi \times t$. Here, $\phi > 1$ accounts for the overhead paid per time-unit in ABFT-protected mode. This linear model for the ABFT overhead fits the existing algorithms for linear algebra, but other models could be considered.

In addition, we pay a checkpoint C_L when exiting the library call (to save computed data). Therefore, the fault-tree execution time is

$$T_L^{\text{ff}} = \phi \times T_L + C_L \tag{2}$$

Finally, the fault-free execution time of the whole epoch is

$$T^{\text{ff}} = T_G^{\text{ff}} + T_L^{\text{ff}} \tag{3}$$

where T_G^{ff} and T_L^{ff} are computed according to the Equations (1) and (2).

2) Cost of failures: Next we have to account for failures. During t time units of execution, the expectation of the number of failures is $\frac{t}{\mu}$, where μ is the mean time between failures of the platform. Note that if the platform comprises N identical resources whose individual mean time between failures is μ_{ind}, then $\mu = \frac{\mu_{\text{ind}}}{N}$. This relation is agnostic of the granularity of the resources, which can be anything from a single CPU to a complex multi-core socket.

For each phase, we have a similar equation: the final execution time is the fault-free execution time, plus the number of failures multiplied by the (average) time lost per failure:

$$T_G^{\text{final}} = T_G^{\text{ff}} + \frac{T_G^{\text{final}}}{\mu} \times t_G^{\text{lost}} \tag{4}$$

$$T_L^{\text{final}} = T_L^{\text{ff}} + \frac{T_L^{\text{final}}}{\mu} \times t_L^{\text{lost}} \tag{5}$$

Equation (4) reads as follows: T_G^{ff} is the failure-free execution time, to which we add the time lost due to failures; the expected number of failures is $\frac{T_G^{\text{final}}}{\mu}$, and t_G^{lost} is the average time lost per failure. We have a similar reasoning for Equation (5). Then,t_G^{lost} and t_L^{lost} remain to be computed.

For t_G^{lost} (GENERAL phase), we discuss both cases:

- If $T_G < P_G$: since we have no checkpoint until the end of the GENERAL phase, we have to redo the execution from the beginning of the phase. On average, the failure strikes at the middle of the phase, hence the expectation of loss is $\frac{T_G^{\text{ff}}}{2}$ time units. We then add the downtime D (time to reboot the resource or set up a spare) and the recovery R. Here R is the time needed for a complete reload from the checkpoint (and $R = C$ if read/write operations from/to the stable storage have the same speed). We derive that

$$t_G^{\text{lost}} = D + R + \frac{T_G^{\text{ff}}}{2} \tag{6}$$

- If $T_G > P_G$: in this case, we have periodic checkpoints, and the amount of execution which needs to be re-done after a failure corresponds to half a checkpoint period on average, so that

$$t_G^{\text{lost}} = D + R + \frac{P_G}{2} \tag{7}$$

For t_L^{lost} (LIBRARY phase), we derive that

$$t_L^{\text{lost}} = D + R_{\bar{L}} + \text{Recons}_{\text{ABFT}}$$

Here, $R_{\bar{L}}$ is the time for reloading the checkpoint of the REMAINDER dataset (and in many cases $R_{\bar{L}} = C_{\bar{L}}$). As for the LIBRARY dataset, there is no checkpoint to retrieve, but instead it must be reconstructed from the ABFT checksums, which takes a time $\text{Recons}_{\text{ABFT}}$.

3) Optimization: We check from Equations (2) and (5) that T_L^{final} is always a constant. Indeed, we derive that

$$T_L^{\text{final}} = \frac{1}{1 - \frac{D + R_{\bar{L}} + \text{Recons}_{\text{ABFT}}}{\mu}} \times (\phi \times T_L + C_L) \tag{8}$$

As for T_G^{final}, it depends on the value of T_G: it is constant when T_G is small. In that case, we derive that

$$T_G^{\text{final}} = \frac{1}{1 - \frac{D + R + \frac{T_G + C_{\bar{L}}}{2}}{\mu}} \times (T_G + C_{\bar{L}}) \tag{9}$$

The interesting case is when T_G is large: in that case, we have to determine the optimal value of the checkpointing period P_G which minimizes T_G^{final}. From Equations (1), (4) and (7), we derive that

$$T_G^{\text{final}} = \frac{T_G}{X} \text{ where } X = \left(1 - \frac{C}{P_G}\right)\left(1 - \frac{D + R + \frac{P_G}{2}}{\mu}\right) \tag{10}$$

We rewrite

$$X = (1 - \frac{C}{2\mu}) - \frac{P_G}{2\mu} - \frac{C(\mu - D - R)}{\mu P_G}$$

The maximum of X gives the optimal period P_G^{opt}. Differentiating X as a function of P_G, we find that it is obtained for

$$P_G^{\text{opt}} = \sqrt{2C(\mu - D - R)} \tag{11}$$

Plugging the value of P_G^{opt} back into Equation (10) provides the optimal value of T_G^{final} when T_G is large.

We have successfully computed the final execution time T^{final} of our composite approach in all cases. In the experiments provided in Section V, we report the corresponding

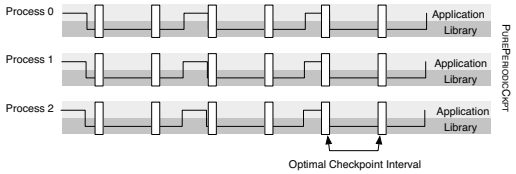


Figure 5: PUREPERIODICCKPT

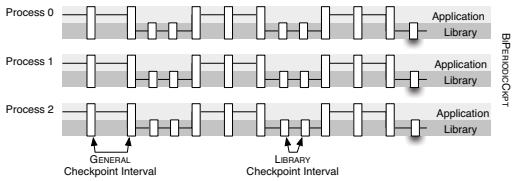


Figure 6: BIPERIODICCKPT

waste. The waste is defined as the fraction of time when platform resources do not progress the application's computation (due to the intrinsic overhead of the resilience technique and to failures that strike the application during execution). The waste is given by

$$\text{WASTE} = 1 - \frac{T_0}{T^{\text{final}}} \tag{12}$$

We conclude this section with a word of caution: the optimal value of the waste is only a first-order approximation, not an exact value. Equation (11) is a refined version of well known formulas by Young [19] and Daly [20]. But just as in [19], [20], the formula only holds when μ, the value of the MTBF, is large in front of the other parameters. Owing to this hypothesis, we can neglect the probability of several failures occurring during the same checkpointing period. However, to further assess the accuracy of the model, when doing simulations in the experiments, we account for all unlikely scenarios and re-execute the work until each period is successfully completed.

C. Comparison with conservative approaches

A fully conservative approach, agnostic of the ABFT library, would perform periodic checkpoints throughout the execution of the whole epoch. As already mentioned, we call this approach PUREPERIODICCKPT (see Figure 5). Let T_{PC}^{final} be the final execution time with this PUREPERIODICCKPT approach; it can be computed from the results of Section IV-B as follows:

- No ABFT: $\alpha = 0$ and $T_L^{\text{final}} = 0$
- We optimize $T_{PC}^{\text{final}} = T_G^{\text{final}}$ just as before, with the same optimal period $P_{PC}^{\text{opt}} = P_G^{\text{opt}}$, employed throughout the epoch.

One can reduce the cost of PUREPERIODICCKPT by noticing that during the LIBRARY epoch, only the LIBRARY dataset is modified (namely M_L). Employing incremental checkpointing would, in this case, yield a checkpoint cost reduction (down to C_L). Obviously, with a different cost of checkpointing, the optimal checkpoint period is different. Therefore, a *semi-conservative approach* (called BIPERIODICCKPT, see Figure 6) assumes that the checkpoint system can recognize that the program has entered a library routine that modifies only a subset of the dataset, and switches to the optimal checkpoint period according to the application phase. During the GENERAL phase, the overhead of failures and protection remains unchanged, but during the LIBRARY phase, the cost of a checkpoint is reduced to C_L (instead of C); however, the cost of reloading from a checkpoint remains R (since the different incremental checkpoints must be combined to recover the entire dataset at rollback time). This leads to two different checkpointing periods, one for each phase. The new optimal checkpoint period can be modeled as follows:

- $T_{PC}^{\text{final}} = T_G^{\text{final}} + T_{LPC}^{\text{final}}$, where T_G^{final} is computed as before
- T_{LPC}^{final} is computed similarly as T_G^{final}, but with different parameters:

$$T_{LPC}^{\text{final}} = \frac{1}{1 - \frac{D+R+\frac{P_{BPC}}{2}}{\mu}} \times \frac{P_{BPC}}{P_{BPC} - C_L} \times T_L \tag{13}$$

and the optimal period is

$$P_{BPC,L}^{\text{opt}} = \sqrt{2C_L(\mu - D - R)} \tag{14}$$

V. EVALUATION

In this section, we evaluate the ABFT&PERIODICCKPT protocol in simulation, and compare its performance to PUREPERIODICCKPT and BIPERIODICCKPT in different scenarios. We start with a description of the simulator and experiments in Section V-A. Then we detail the results of the comparison of the different protocols in Section V-B. In Section V-B, we compare simulation results and predicted performance results analytically computed from the models presented in Sections IV-B and IV-C, and we do obtain a very good correspondence. Finally, we conduct a weak scalability study in Section V-C, in order to assess the performance of the various protocols at very large scale.

A. Validation

To validate the performance models, we have implemented a simulator, based on discrete event simulation, that reproduces the behavior of the different algorithms, even in cases that the performance models cannot cover. Indeed, as mentioned in Section IV-B3, a few approximations have been made when considering the mathematical models, to make their expressions tractable. For example, the models assume that a single failure may hit the system, until its recovery. The effect of events like overlapping failures, which is uncommon when the MTBF is large enough, is neglected in the proposed performance model. The simulator, however, takes these events into account, accurately reproducing the corresponding costs.

In the simulator, failures are generated following an Exponential distribution law parameterized to fix the MTBF to a given value. Then the application, and the chosen fault tolerance mechanism, are unfolded on that set of failures, triggering rollbacks, and other protocol-specific overheads, to measure the duration of the execution. For each scenario, and each parameter, the average termination time over a thousand executions is returned by the simulator.

We present in [24] an exhaustive evaluation of the different parameters independently, comparing the results as predicted

by the models, and the simulation. In this paper, we focus the analysis on a smaller subset. We consider an application that executes for a week when there is neither a fault tolerance mechanism nor any failure. The time to take a checkpoint and rollback the whole application is 10 minutes (C, R), a consistent order of magnitude for current applications at large scale [5]. We consider that the ratio of the memory that is modified by the LIBRARY phase (ρ) is fixed at 0.8 (to vary a single parameter at a time in our simulation), and the overhead due to ABFT is $\phi = 1.03$ (again, typical from production deployments [9]).

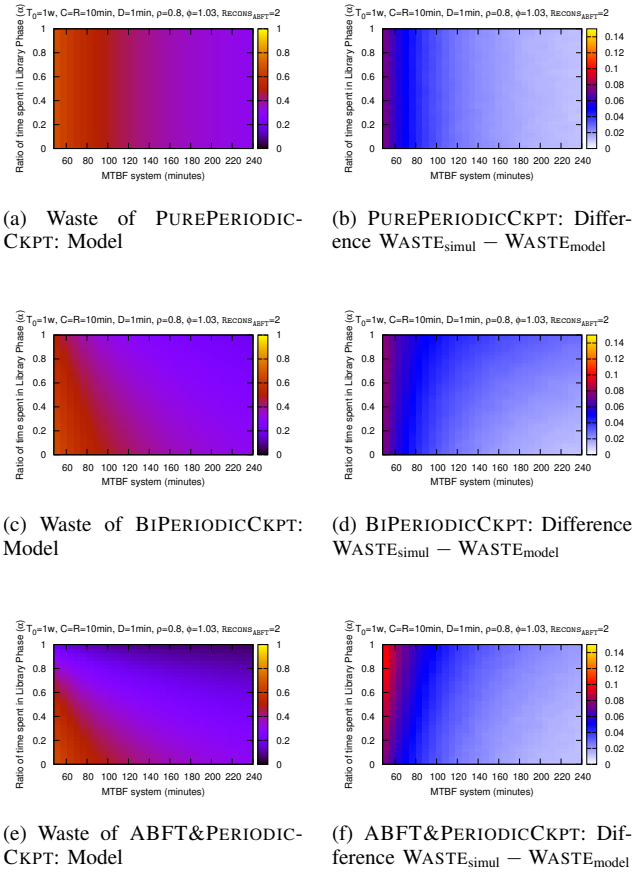
Figure 7 presents 6 evaluations of that scenario. The MTBF of the system varies on the x-axis, and the ratio of time spent in the LIBRARY phase (α) on the y-axis. In Figures 7a to 7f, we present the waste predicted by the model, and validate the model by observing the difference between the model prediction and the waste measured from the simulator for a given combination of parameters and protocol. From the validation perspective, the figures on the right side show an excellent correspondence between predicted (from the model) and actual (obtained from simulation) values. For small MTBF values, the model tends to slightly underestimate the waste. That underestimation does not exceed 12% in the worst case and quickly decreases to below 5%. Qualitatively, this underestimation is expected, because an approximation that must be done to allow a closed formula representation is to assume that failures will not hit processors while they are recovering from a previous failure. In reality, when the MTBF is very small, this event can sometimes happen, forcing the system to start a new recovery, and introducing additional waste.

B. PUREPERIODICCKPT, BIPERIODICCKPT, *and* ABFT&PERIODICCKPT

Consider Figures 7a and 7b, that represent the waste of PUREPERIODICCKPT as a function of the MTBF (μ) and the amount of time spent in the LIBRARY routine (α): it is obvious that the PUREPERIODICCKPT protocol, which is oblivious of the different phases of the application, presents a waste that is only a function of the MTBF. As already evaluated and explained in many other works, when the MTBF increases, the waste decreases, because the overheads due to failure handling tend toward 0, and the optimal checkpointing period can increase significantly, reducing the waste due to resilience in a fault-free execution.

Comparatively, for the protocol BIPERIODICCKPT presented in Figures 7c and 7d, the parameter α affects the optimal periods used both in the LIBRARY and general phases. Since the cost of checkpointing for these phases differs by 20% ($C_L = 0.8C$), when the relative time spent in the GENERAL routine increases (α is closer to 0), then the protocol behaves more and more as PUREPERIODICCKPT. When α is almost 1, on the contrary, the behavior is similar to PUREPERIODICCKPT, but with a checkpoint cost reduction of 20%. Thus, the waste becomes minimal when α tends toward 1.

In Figures 7e and 7f, we present the waste for the ABFT&PERIODICCKPT protocol. When α tends toward 0, as above, the protocol behaves as PUREPERIODICCKPT, and no benefit is shown. When 50% of the time is spent in the LIBRARY routine, the benefit, compared to PUREPERIODICCKPT, but also compared to BIPERIODICCKPT, is already visible: for 50% of the failures (when the failure hits during a LIBRARY phase), the cost of recovery is reduced to 20% of the rollback cost, plus the constant overhead of ABFT recovery. Moreover, periodic checkpointing is disabled 50% of the time, producing yet another gain compared to BIPERIODICCKPT which still requires saving 80% of the memory periodically. In this case, the gain in checkpoint avoidance compensates for the waste induced by additional computations done during the LIBRARY phase to provide the ABFT protection. When considering the extreme case of 100% of the time spent in the LIBRARY phases, the overhead tends to reach the overhead induced by the slowdown factor of ABFT ($\phi = 1.03$, hence 3% overhead).



(a) Waste of PUREPERIODICCKPT: Model

(b) PUREPERIODICCKPT: Difference $\text{WASTE}_{\text{simul}} - \text{WASTE}_{\text{model}}$

(c) Waste of BIPERIODICCKPT: Model

(d) BIPERIODICCKPT: Difference $\text{WASTE}_{\text{simul}} - \text{WASTE}_{\text{model}}$

(e) Waste of ABFT&PERIODICCKPT: Model

(f) ABFT&PERIODICCKPT: Difference $\text{WASTE}_{\text{simul}} - \text{WASTE}_{\text{model}}$

Figure 7: Waste as a function of MTBF and fraction of time α spent in LIBRARY phase; difference of the measured waste by simulation $\text{WASTE}_{\text{simul}}$ minus the predicted waste by the model $\text{WASTE}_{\text{model}}$. Here W =1 week, C =R =10 minutes, C_L =0.8C, ϕ =1.03, $\text{Recons}_{\text{ABFT}}$ =2s.

C. Weak Scalability

As illustrated above, the ABFT&PERIODICCKPT approach exhibits better performance when a significant time is spent in the LIBRARY phase, *and* when the failure rate implies a small optimal checkpointing period. If the check-

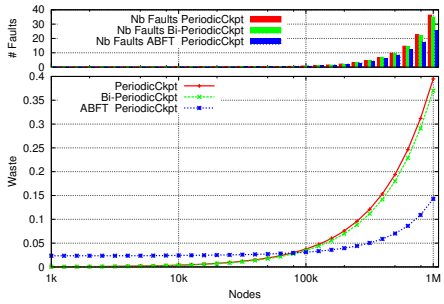


Figure 8: Total waste for ABFT&PERIODICCKPT, BIPERIODICCKPT and PUREPERIODICCKPT, when considering the weak scaling of an application with a fixed ratio of 80% spent in a LIBRARY routine.

pointing period is large (because failures are rare), or if the duration of the LIBRARY phase is small, then the optimal checkpointing interval becomes larger than the duration of the LIBRARY phase, and the algorithm automatically resorts to the BIPERIODICCKPT protocol. This can also be the case when the epoch itself is smaller than (or of the same order of magnitude as) the optimal checkpointing interval (i.e., when the application does a fast switching between LIBRARY and GENERAL phases).

However, consider such an application that frequently switches between (relatively short) LIBRARY and GENERAL phases. When porting that application to a future larger scale machine, the number of nodes that are involved in the execution will increase, and at the same time, the amount of memory on which the ABFT operation is applied will grow (following Gustafson's law). This has a double impact: the time spent in the ABFT routine increases, while at the same time, the MTBF of the machine decreases. In this section, we evaluate quantitatively how this scaling factor impacts the relative performance of the ABFT&PERIODICCKPT, PUREPERIODICCKPT and BIPERIODICCKPT algorithms. Owing to the good correspondence between results from the model and results from the simulation, we (confidently) use only the model in this scalability study.

First, we consider the case of an application where the LIBRARY and GENERAL phases scale at the same rate. We take the example of linear algebra kernels operating on $2D$-arrays (matrices), that scale in $O(n^3)$ of the array order n (in both phases). Following a weak scaling approach, the application uses a fixed amount of memory M_{ind} per node, and when increasing the number x of nodes, the total amount of memory increases linearly as $M = xM_{ind}$. Thus $O(n^2) = O(x)$, and the parallel completion time of the $O(n^3)$ operations, assuming perfect parallelism, scales in $O(\sqrt{x})$.

To instantiate this case, we take an application that would iterate over a thousand epochs, each epoch consisting of 80% of a LIBRARY phase, and 20% of a GENERAL phase. At 10,000 nodes, the duration of a single epoch is arbitrarily set to 1 minute, and the scaling factor of the corresponding $O(n^3)$ operation is applied, when varying the number of nodes that participate in the computation. We set the duration of the complete checkpoint and rollback (C and R, respectively) to 1 minute when 10,000 nodes are involved, and we scale this value linearly with the total amount of memory, when varying the number of nodes. The MTBF at 10,000 nodes is set to 1 failure every day, and this also scales linearly with the number of components. The ABFT overheads, and the downtime, are set to the same values as in the previous section, and 80% of the application memory (M_L) is touched by the LIBRARY phase.

Given these parameters, Figure 8 shows (i) the relative waste of PUREPERIODICCKPT, BIPERIODICCKPT, and ABFT&PERIODICCKPT, as a function of the number of nodes, and (ii) the average number of faults that each execution will have to deal with to complete. The expected number of faults is the ratio of the application duration by the platform MTBF (which decreases when the number of nodes increases, generating more failures). The fault-free execution time increases with the number of nodes (as noted above), and the fault-tolerant execution time is also increased by the waste due to the protocol. Thus, the total execution time of PUREPERIODICCKPT or BIPERIODICCKPT is larger at 1 million nodes than the total execution time of ABFT&PERIODICCKPT at the same scale, which explains why more failures happen for these protocols.

When comparing BIPERIODICCKPT and PUREPERIODICCKPT, one can see the benefit of incremental checkpointing, which spares about 20% of the checkpoint time during 80% of the checkpoints: this benefit shows up by a small linear reduction of the waste for BIPERIODICCKPT. However, both approaches perform similarly with respect to the number of nodes in this weak-scaling experiment.

Up to approximately 100,000 nodes, the fault-free overhead of ABFT negatively impacts the waste of the ABFT&PERIODICCKPT approach, compared to BIPERIODICCKPT or PUREPERIODICCKPT. Because the MTBF of the platform is very large compared to the application execution time (and hence to the duration of each LIBRARY phase), periodic checkpointing approaches have a very large checkpointing interval, introducing very few checkpoints, thus a small failure-free overhead. Because failures are rare, the cost due to time lost at rollbacks does not overcome the benefits of a small failure-free overhead, while the ABFT technique must pay the linear overhead of maintaining the redundancy information during the whole computation of the LIBRARY phase.

Once the number of nodes reaches 100,000, however, two things happen: failures become more frequent, and the time lost due to failures starts to impact rollback recovery approaches. Thus, the optimal checkpointing interval of periodic checkpointing becomes smaller, introducing more checkpointing overheads. During 80% of the execution, however, the ABFT&PERIODICCKPT approach can avoid these overheads, and when they reach the level of linear overheads due to the ABFT technique, ABFT&PERIODICCKPT starts to scale better than both periodic checkpointing approaches.

All protocols have to resort to checkpointing during the GENERAL phase of the application. Thus, if failures hit during this phase (which happens 20% of the time in this example), they will all have to resort to rollbacking and lose some computation time. Hence, when the number of nodes increases and the MTBF decreases, eventually, the time spent in rollbacking

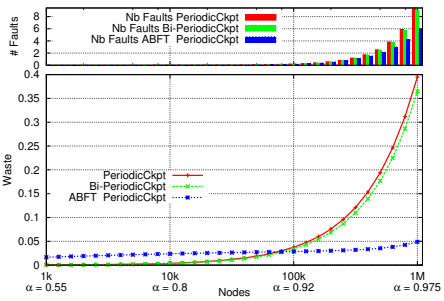


Figure 9: Total waste for ABFT&PERIODICCKPT, BIPERIODICCKPT and PUREPERIODICCKPT, when considering the weak scaling of an application with variable ratio of time spent in a LIBRARY routine.

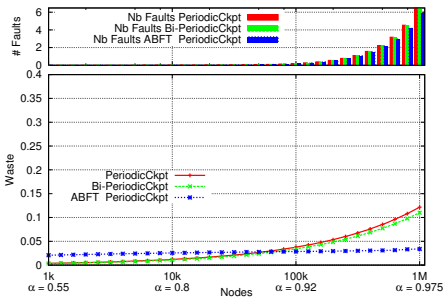


Figure 10: Total waste for ABFT&PERIODICCKPT, BIPERIODICCKPT and PUREPERIODICCKPT, when considering the weak scaling of an application with variable ratio of time spent in a LIBRARY routine, and constant checkpointing time

and re-computing, which is linear in the number of faults, will increase the waste of all algorithms. However, one can see that this part is better controlled by the ABFT&PERIODICCKPT algorithm.

Next we consider the case of an unbalanced GENERAL phase: consider an application where the LIBRARY phase has a cost $O(n^3)$ (where n is the problem size), as above, but where the GENERAL phase consists of $O(n^2)$ operations. This kind of behavior is reflected in many applications where matrix data is updated or modified between consecutive calls to computation kernels. Then, the time spent in the LIBRARY phase will increase faster with the number of nodes than the time spent in the GENERAL phase, varying α. This is what is represented in Figure 9. We took the same scenario as above for Figure 8, but α is a function of the number of nodes chosen such that at 10,000 nodes, $\alpha = T_L^{\text{final}}/T^{\text{final}} = 0.8$, and everywhere, $T_L^{\text{final}} = O(n^3) = O(\sqrt{x})$, and $T_{PC}^{\text{final}} = O(n^2) = O(1)$. We give the value of α under the number of nodes, to show how the fraction of time spent in LIBRARY phases increases with the number of nodes.

The PUREPERIODICCKPT protocol is not impacted by this change, and behaves exactly as in Figure 8. Note, however, that $T^{\text{final}} = T_L^{\text{final}} + T_{PC}^{\text{final}}$ progresses at a lower rate in this scenario than in the previous scenario, because T_{PC}^{final} does not increase with the number of nodes. Thus, the average number of faults observed for all protocols is much smaller in this scenario. Because more and more time (relative to the duration of the application) is spent in the LIBRARY phase, where 20% of the memory does not need to be saved, the BIPERIODICCKPT algorithm increases its benefit, compared to PUREPERIODICCKPT: less overhead is paid for checkpoints that happen during LIBRARY phases, and the optimal period of checkpointing during these phases are longer. The cost of failures, however, remains the same, since the state of the entire application (LIBRARY memory, and REMAINDER memory) must be restored at rollback time.

The efficiency on ABFT&PERIODICCKPT, however, is more significant. The latter protocol benefits from the increased α ratio in both cases: since more time is spent in the LIBRARY phase, periodic checkpointing is de-activated for relatively longer periods. Moreover, this increases the probability that a failure will happen during the LIBRARY phase, where the recovery cost is greatly reduced using ABFT techniques. Thus, ABFT&PERIODICCKPT is capable of mitigating failures at a much smaller overhead than simple periodic checkpointing, and more importantly with better scalability.

In both previous evaluations, we have always considered that the checkpointing (and rollback recovery) time is proportional to the global amount of memory that needs to be saved in these checkpoints. This is realistic, if the checkpoint needs to be stored in a remote place, to guarantee its availability after a failure occurs. In this case, the interconnect (or the bandwidth capacity of the disks) eventually becomes a bottleneck, and the saving time becomes proportional to the number of computing resources saving their state simultaneously. To mitigate the negative effect of this bottleneck, system designers are studying a couple of alternative approaches. One consists of featuring each computing node with local storage capability, ensuring through the hardware that this storage will remain available during a failure of the node. Another approach consists of using the memory of the other processors to store the checkpoint, pairing nodes as "buddies," thus taking advantage of the high bandwidth capability of the high speed network to design a scalable checkpoint storage mechanism [25], [26], [27], [28].

Thus, it might be reasonable to consider in the future that the checkpoint storage time will not increase with the number of nodes, but on the contrary will remain constant. This is the scenario that we contemplate in Figure 10. The scenario is identical to the previous scenario of Figure 9, but the checkpoint time (C) and rollback recovery time (R) is independent of the number of nodes that checkpoint, and is fixed at 60s. One can see a noteworthy benefit on both periodic checkpointing protocols: even at 1 million nodes, the waste due to the protocols and the few faults that do occur during the execution (up to 6 failures in average during the whole execution) both add up to below 15%. At the same time, the ABFT technique continues to introduce its constant overhead (due to additional computation) during the whole execution, and appears to present a waste that is almost constant when the number of nodes increases.

Figure 10 shows that PUREPERIODICCKPT and BI-

PERIODICCKPT are less efficient than ABFT&PERIODICCKPT at 1 million nodes, despite the perfectly scalable checkpointing hypothesis. To reach comparable performance, we must reduce checkpointing overhead by a factor of 10 and use $C = R = 6s$. Such low figures can only be achieved through new hardware (like NVRAM), and new incremental and hierarchical checkpointing protocols.

VI. CONCLUSION

In this paper, we have formalized and quantified a novel method of composing fault tolerance approaches for applications that alternate between ABFT-aware and ABFT-unaware sections, where each of these sections is protected by its own mechanism, ABFT in one case and checkpoint/restart in the other. A performance model has been derived for such methods and thoughtfully validated using a simulator developed for this scope. We have compared our composite approach with a traditional periodic checkpointing approach using rollback and recovery, under different plausible scenarios. Our model predicts that the cost of a "checkpoint only" approach will maintain a reasonable overhead only under highly optimistic assumptions, where the checkpointing cost stagnates when the number of computational resources increases. Under more realistic assumptions, where the checkpointing cost increases with the number of resources, the composite approach will provide significantly greater benefits compared with checkpoint/restart, by minimizing the waste and thus increasing the platform throughput. Our weak scalability study shows that the gain of the composite approach will continue to grow with the increase in the number of computing resources, making it a more plausible and desirable approach at very large scale.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation (NSF #0904952 and #1063019), JST Japan, and the French Research Agency (ANR) through the Rescue project. Yves Robert is with Institut Universitaire de France.

REFERENCES

[1] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012022, 2007.

[2] G. Zheng, X. Ni, and L. Kale, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *Dependable Systems and Networks Workshop (DSN-W)*, 2012, pp. 1–6.

[3] D. Kondo, B. Javadi, A. Iosup, and D. Epema, "The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems," *Cluster Computing and the Grid, IEEE International Symposium on*, pp. 398–407, 2010.

[4] J. Dongarra *et al.*, "The International Exascale Software Project: a Call To Cooperative Action By the Global High-Performance Community," *Int. J. High Performance Computing Applications*, vol. 23, no. 4, pp. 309–322, 2009.

[5] K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the Viability of Process Replication Reliability for Exascale Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 44:1–44:12.

[6] G. Bosilca *et al.*, "Unified model for assessing checkpointing protocols at extreme-scale," *Concurrency and Computation: Practice and Experience*, 2013. [Online]. Available: http://dx.doi.org/10.1002/cpe.3173

[7] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, 1984.

[8] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proc. Int. Conf. Supercomputing*. ACM, 2011, pp. 152–161.

[9] P. Du, A. Bouteiller *et al.*, "Algorithm-based fault tolerance for dense matrix factorizations," in *PPoPP*. ACM, 2012, pp. 225–234.

[10] T. Davies, C. Karlsson, H. Liu, C. Ding, , and Z. Chen, "High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing," in *Proc. Int. Conf. Supercomputing*. ACM, 2011, pp. 162–171.

[11] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *Proc. Int. Conf. Supercomputing*. ACM, 2012, pp. 69–78.

[12] E. Agullo, L. Giraud, A. Guermouche, J. Roman, and M. Zounon, "Towards resilient parallel linear Krylov solvers: recover-restart strategies," INRIA, Research report RR-8324, Jul. 2013.

[13] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th international symposium on High performance distributed computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 73–84.

[14] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75.

[15] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Survey*, vol. 34, pp. 375–408, 2002.

[16] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Dist. Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[17] 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.

[18] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," in *PPoPP*. ACM, 2005, pp. 213–223.

[19] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Comm. of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[20] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *FGCS*, vol. 22, no. 3, pp. 303–312, 2004.

[21] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio, "Distribution-free checkpoint placement algorithms based on min-max principle," *IEEE TDSC*, pp. 130–140, 2006.

[22] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011, pp. 1–11.

[23] Z. Zheng and Z. Lan, "Reliability-aware scalability models for high performance computing," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 2009, pp. 1–9.

[24] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra, "Assessing the impact of ABFT and checkpoint composite strategies," University of Tennessee, Research Report ICL-UT-13-03, Sep. 2013.

[25] G. Zheng, L. Shi, and L. V. Kale, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *Cluster Computing, 2004 IEEE International Conference on*. IEEE Computer Society, 2004, pp. 93–103.

[26] X. Ni, E. Meneses, and L. V. Kalé, "Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE Computer Society, 2012, pp. 364–372.

[27] J. Dongarra, T. Herault, and Y. Robert, "Revisiting the double checkpointing algorithm," in *APDCM 2013*. IEEE Computer Society Press, 2013, pp. 706–715.

[28] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. D. Panda, "A 1 PB/s file system to checkpoint three million MPI tasks," in *HPDC*. ACM, 2013, pp. 143–154.