Checkpointing à la Young/Daly: An Overview

Anne Benoit
anne.benoit@inria.fr
LIP, ENS Lyon, France

Yishu Du
yishu.du@inria.fr
Tongji Univ., Shanghai, China
& LIP, ENS Lyon, France

Thomas Herault
herault@icl.utk.edu
Univ. Tenn. Knoxville, USA

Loris Marchal
loris.marchal@inria.fr
LIP, ENS Lyon, France

Guillaume Pallez
guillaume.pallez@inria.fr
Inria Bordeaux, France

Lucas Perotin
lucas.perotin@inria.fr
LIP, ENS Lyon, France

Yves Robert
yves.robert@inria.fr
LIP, ENS Lyon, France
& Univ. Tenn. Knoxville, USA

Hongyang Sun
hongyang.sun@ku.edu
University of Kansas, USA

Frédéric Vivien
frederic.vivien@inria.fr
Inria & LIP, ENS Lyon, France

ABSTRACT

The Young/Daly formula provides an approximation of the optimal checkpoint period for a parallel application executing on a supercomputing platform. The Young/Daly formula was originally designed for preemptible tightly-coupled applications. We provide some background and survey various application scenarios to assess the usefulness and limitations of the formula.

KEYWORDS

Checkpoint, Optimal period, Young/Daly formula.

ACM Reference Format:
Anne Benoit, Yishu Du, Thomas Herault, Loris Marchal, Guillaume Pallez, Lucas Perotin, Yves Robert, Hongyang Sun, and Frédéric Vivien. 2022. Checkpointing à la Young/Daly: An Overview. In *Proceedings of ACM Conference (IC3).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Checkpointing is the standard technique to protect applications running on HPC (High-Performance Computing) platforms. Every day, the platform experiences a few fail-stop errors (or failures, we use both terms indifferently). After each failure, the application executing on the faulty processor (and likely on many other processors for a large parallel application) is interrupted and must be restarted. Without checkpointing, all the work executed for the application is lost. With checkpointing, the execution can resume from the last checkpoint, after some downtime (enroll a spare to replace the faulty processor) and a recovery (read the checkpoint).

Consider a parallel application executing on an HPC platform whose nodes are subject to fail-stop errors. How frequently should it be checkpointed so that its expected execution time is minimized? There is a well-known trade-off (see Figure 1): taking too many checkpoints leads to high overhead, especially when there are few failures while taking too few checkpoints leads to a large re-execution time after each failure. The optimal checkpointing period is (approximately) given by the Young/Daly formula as $W_{YD} = \sqrt{2\mu C}$ [21, 67], where μ is the application MTBF (Mean Time Between Failures) and C is the checkpoint duration.

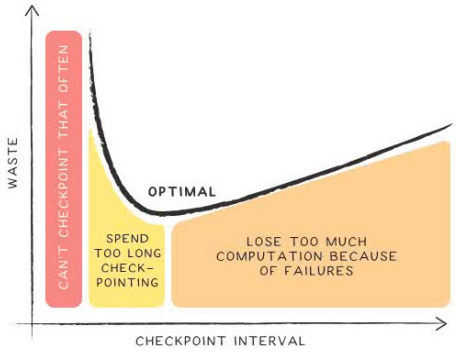

Figure 1: Trade-off for the optimal checkpoint period.

This paper provides an overview of the applicability and robustness of the Young/Daly formula for different application scenarios. There are two main frameworks. The first part of the survey deals with preemptible applications, which may be checkpointed at any time step. In this context, checkpointing is a coordinated process and involves all the processors enrolled in the execution of the application. The second part of the survey focuses on task systems, where applications are composed of a set of atomic tasks, possibly with inter-dependences. In this context, checkpoints are task-based and can be taken only at the end of a task. Only the processors that execute a task are involved in its checkpoint. The problem is then to decide which tasks to checkpoint.

The paper is organized as follows. We first survey preemptible applications in Section 2. Then we deal with task systems in Section 3. We address some questions related to uncertainty in Section 4. Finally, we conclude with some open questions in Section 6.

IC3, August 2022, Noida, India

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 PREEMPTIBLE APPLICATIONS

In this section, we deal with parallel applications that can be checkpointed at any time. In scheduling terminology, the applications are preemptible.

2.1 Background

Platform and applications. Consider a large parallel platform with m identical processors, or nodes. These nodes are subject to fail-stop errors, or failures. A failure interrupts the execution of the node and provokes the loss of its whole memory.

Consider a parallel application running on several nodes: when one of these nodes is struck by a failure, the state of the application is lost, and execution must restart from scratch unless a fault-tolerance mechanism has been deployed. The classical technique to deal with failures makes use of a checkpoint-restart mechanism: the state of the application is periodically checkpointed, i.e., all participating nodes take a checkpoint simultaneously. This is the standard coordinated checkpointing protocol, which is routinely used on large-scale platforms [19], where each node writes its share of application data to stable storage (checkpoint of duration C). When a failure occurs, the platform is unavailable during a downtime D, which is the time to enroll a spare processor that will replace the faulty processor [21, 34]. Then, all application nodes (including the spare) recover from the last valid checkpoint in a coordinated manner, reading the checkpoint file from stable storage (recovery of duration R). Finally, the execution is resumed from that point on, rather than starting again from scratch. Note that failures can strike during checkpoint and recovery, but not during downtime (otherwise there are no differences between downtime and recovery and we can simply include the downtime in the recovery time). When a failure hits a processor, that processor is replaced by a spare. This amounts to starting anew with a fresh processor. In the terminology of stochastic processes, the faulty processor is rejuvenated. However, all the other processors are not rejuvenated: this would be infeasible due to the multitudinous spares needed!

Failures. We assume that each node experiences failures whose inter-arrival times follow Independent and Identically Distributed (IID) random variables obeying an arbitrary probability distribution $\mathcal{D}$. We only assume that $\mathcal{D}$ is continuous and of finite expectation and variance, a condition satisfied by all standard distributions. We let μ_{ind} denote the expectation of $\mathcal{D}$, also known as the individual processor MTBF. Even if each node has an MTBF of several years, large-scale parallel platforms are composed of so many nodes that they will experience several failures per day [18, 28]. Hence, a parallel application using a significant fraction of the platform will typically experience a failure every few hours. More precisely, an application executing with p processors has an MTBF $\mu = \frac{\mu_{ind}}{p}$: intuitively, the application is struck by failures at a rate which is p times higher than that of each enrolled processor. We come back to this statement in Section 2.3.

Checkpointing strategies. Given a parallel application of length T_{base} (base time without checkpoints nor failures), the optimization problem is to decide when and how often to take a checkpoint, to minimize the expected execution time of the application. The application is divided into N_c segments of length W_i, $1 \le i \le N_c$, each followed by a checkpoint of length C. Of course $\sum_{i=1}^{N_c} W_i = T_{base}$. We add a final checkpoint at the end of the last segment, e.g., to write final outputs to stable storage. Symmetrically, we add an initial recovery when re-executing the first segment of an application (e.g., to read inputs from stable storage) if it has been struck by a failure before completing the checkpoint. Adding a final checkpoint and an initial recovery brings symmetry and simplifies formulas, but it is not at all mandatory: see [12] for an extension relaxing either or both assumptions. The question is then to determine the number N_c of segments and their lengths W_i.

2.2 The Young/Daly Formula

Here is an intuitive (but simplified) derivation of the Young/Daly formula for the optimal checkpoint period. Owing to the addition of the final checkpoint and the initial recovery, all segments of the application have the same shape. It is thus natural (by symmetry) to assume that they have the same length W in the optimal solution. Thus we assume that checkpoints are taken periodically, after every W unit of work. Now, after every W unit of work, we spend C seconds to checkpoint, which corresponds to a first source of waste $S_1 = \frac{C}{W+C}$. Here the waste is defined as the fraction of time during which the application is not performing useful computations; checkpoint, recovery, downtime, and re-execution do not count as useful computations. S_1 is the *failure-free* waste. The second source of waste S_2 is due to failures: each time a failure strikes, which happens every μ seconds on average, we lose $D + R$ for downtime and recovery, and then we have to re-execute some work, namely the work performed since the last checkpoint (or from the beginning of the execution if none has been taken yet). On average again, the failure strikes in the middle of the segment: sometimes before, sometimes after, hence, on average after $\frac{W+C}{2}$ seconds. We obtain $S_2 = \frac{1}{\mu}(D + R + \frac{W+C}{2})$. S_1 is the *failure-induced* waste. Altogether, both sources of waste approximately add up, so we have to find W that minimizes $S_1 + S_2$. We further simplify the solution by assuming that W must be an order of magnitude higher than the fault-tolerance parameters D, C, R. This is a necessary condition for the waste to remain reasonably low. This leads to $S_1 \approx \frac{C}{W}$ and $S_2 \approx \frac{W}{2\mu}$. The total waste $S_1 + S_2 \approx \frac{C}{W} + \frac{W}{2\mu}$ is minimum for

$$W_{YD} = \sqrt{2\mu C} \tag{1}$$

This is nothing else than the famous Young/Daly formula! Finally, note that $S_1 = S_2$ for W_{YD}, which corroborates the intuition given in Figure 1 that both sources of waste, failure-free and failure-induced, should be balanced in the optimal solution. See [34] for a more detailed derivation using the waste argument.

2.3 Accuracy

Recall that each node experiences failures whose inter-arrival times follow IID random variables obeying a probability distribution $\mathcal{D}$. When $\mathcal{D}$ is $\textsc{Exp}(\lambda)$, i.e., an Exponential distribution of rate λ, the framework is well-understood. This is because the inter-arrival times of the failures that strike an application with p processors are IID random variables obeying an Exponential distribution $\textsc{Exp}(p\lambda)$. This is due to the memoryless property of the Exponential distribution: when a failure strikes one processor, that processor is rejuvenated, while the remaining $p - 1$ processors are not. With an

arbitrary distribution $\mathcal{D}$, the time to the next failure would depend upon the history of these $p-1$ processors: for each of them, the time to their next failure depends upon when their last failure struck. This is not the case for an Exponential distribution, owing to its memoryless property: after a failure on any of the p processors, the time to the next failure remains the same random variable $\text{Exp}(\lambda)$ for each of them, rejuvenated or not. Therefore, the time to the next failure for the application obeys an $\text{Exp}(p\lambda)$ distribution, as the minimum of p $\text{Exp}(\lambda)$ distributions. From the resilience point of view, the application executes on a single processor of fault rate $p\lambda$! Owing to this observation, one can formally derive that the optimal checkpoint strategy is periodic, and compute the optimal checkpoint period. The derivation is a bit technical and the optimal segment length W_{opt} is obtained using the Lambert W function. But comfortingly, a first-order approximation of W_{opt} is W_{YD}, the value given by the Young/Daly formula. See [12, 16] for details on the derivation.

Now, any continuous distribution $\mathcal{D}$ other than Exponential is not memoryless, and the optimal checkpoint strategy is unknown in that case. The bad news is that the most accurate probability distributions modeling processor failures are LogNormal [33] and Weibull [52, 53, 60, 61] instead of Exponential. For instance, LANL failure traces are best fit by Weibull distributions of different shapes [27]. Weibull distributions with a shape parameter smaller than 1 experience infant mortality; for those distributions, it is known that periodic checkpointing is not optimal. Intuitively, the length of a segment between two consecutive checkpoints should increase with time, as the instantaneous failure rate decreases. However, the good news is that the MTBF can still be defined as the limit:

$$\lim_{T\to\infty} \frac{n(T)}{T} = \frac{\mu_{ind}}{p}$$

where $n(T)$ is the expected number of failures striking an application with p processors in the time interval $[0, T]$. This limit exists for any regular distribution $\mathcal{D}$. A natural heuristic is to use a periodic checkpoint strategy, with a segment length given by the Young/Daly formula and using that latter value for the MTBF. It is unknown how this approach is close to the optimal but it seems good enough in many scenarios. See [12, 16] for an assessment of this heuristic, and for a comparison with other checkpoint strategies which aim at maximizing work or efficiency until the next failure.

2.4 Extensions

In Section 2.2, we have shown how to derive the optimal checkpoint period when the objective is to minimize the expected completion time of the application. We used a simplified model where no computation could take place while checkpointing. Modern processors could run several threads in parallel and compute while executing I/O transfers. A first extension to the framework of Section 2.2 is to extend the model with a linear slowdown factor $\alpha \in [0, 1]$, where, say, $\alpha = 0.5$ means that computations progress at half the main speed when checkpointing. The two extreme values are $\alpha = 0$ when checkpoints are blocking (no overlap), and $\alpha = 1$ when execution can progress with no penalty while a checkpoint is taken (full overlap). The Young/Daly formula becomes $W_{YD} = \sqrt{2\mu(1-\alpha)C}$. Note that $\alpha = 0$ leads to the original Young/Daly formula, while $\alpha = 1$ leads to $W_{YD} = 0$, which means that one should checkpoint all the time if checkpointing is free! Of course in practical scenarios we expect $\alpha < 1$. See [34] for more details.

Another extension to the framework of Section 2.2 is to target a different optimization objective: instead of minimizing the (expected) total execution time, one would aim at minimizing the (expected) total energy consumed to execute the application. This objective is important both for economic and environmental reasons. The optimal period W_{energy} to minimize energy consumption is different from the Young/Daly formula mainly because the power spent when computing is not the same as power spent when checkpointing. More precisely, the power consumption at each time step of the application relies on three components:

- $\mathcal{P}_{\textbf{Static}}$**:** base power consumed when platform is switched on,
- $\mathcal{P}_{\textbf{Cal}}$**:** when the platform is computing, we have to consider the CPU overhead in addition to the static power $\mathcal{P}_{\text{Static}}$.
- $\mathcal{P}_{\textbf{I/O}}$**:** similarly, this is the power overhead due to file I/O. This supplementary power consumption is induced by checkpointing, or when recovering from a failure.

A key parameter to compare W_{energy} and W_{YD} is the ratio $\frac{\mathcal{P}_{\text{Static}}+\mathcal{P}_{\text{I/O}}}{\mathcal{P}_{\text{Static}}+\mathcal{P}_{\text{Cal}}}$. See [3, 27, 29] and the references therein for further details.

Finally, another optimization objective is to minimize the expected volume of I/O operations due to checkpointing and recovery. This objective is important because I/O resources are scarce in HPC platforms. Typical HPC applications execute on dedicated computing nodes but share the I/O bandwidth of the platform with other applications. Hence, decreasing the volume of I/O operations by each application is likely to improve the global throughput of the platform. A natural question is then: given a single application that executes on the platform, can we increase the checkpoint period significantly beyond the Young/Daly formula without sacrificing too much in performance? Note that we have a bi-criteria optimization problem here because we need to trade off performance with I/O pressure. Note also that a single application running on the platform may be a *capability* workload that spans the entire platform. The answer to the question is yes: Arunagiri et al. [1] studied longer, sub-optimal periods for a single application, with the intent of reducing I/O pressure. They showed, both analytically and empirically using four real platforms, that a decrease in the I/O requirement can be achieved with only a small increase in waste.

However, *space-sharing* HPC platforms for the concurrent execution of multiple parallel applications is the prevalent usage strategy in today's HPC centers, and *capability* workloads that span the entire platform are much less common [65]. The question becomes how to avoid contention when several applications try to checkpoint at the same time: the I/O bandwidth will be shared among these applications, their checkpoint time will increase, and the Young/Daly formula that was computed for each application in isolation is no longer optimal due to these interferences. We will come back to this question in Section 4.2.

2.5 Loosely-Coupled Applications

The Young/Daly formula applies to a parallel application where all processors progress and cooperate continuously, e.g., by exchanging messages: the application cannot continue its execution when

one processor is struck by a failure; it has to wait until a spare is up and running. In other words, the application is assumed to be tightly coupled and behaves as if it were executed on a single (very powerful) processor. Recall that we already made such an analogy for failures: in fact, the parallel application can be viewed as a sequential one, executing on a single processor, very fast but very unreliable too.

What if the application is not tightly-coupled? If the application includes several tasks that can execute concurrently and independently on different subsets of resources, how frequently should each task be checkpointed? We use the word *task* here, but not in the traditional meaning where tasks are atomic and can only be checkpointed at the end of their execution (see Section 3 for such a framework). On the contrary, we assume that each task is preemptible and can be checkpointed at any time step. It is then natural to checkpoint each task using the Young/Daly period. But is this a good strategy, given that many tasks execute in parallel, and that the failure of one task will slow down the whole application?

Consider the simple example of a fork-join application that consists of 302 tasks: an entry task, 300 identical parallel tasks, and an exit task. Each parallel task runs on $p = 30$ processors for $T_{base} = 10$ hours, and is checkpointed in $C = 6$ minutes. The platform has at least 9,000 processors so that the 300 parallel tasks can indeed execute concurrently. Such applications are typical of HPC applications that explore a wide range of parameters or launch subproblems in parallel. Assume a short downtime $D = 1$ minute, and recovery time $R = C$. Finally, assume that each task has 0.5% chances to fail during execution; this setting corresponds to an individual MTBF μ_{ind} such that $1 - e^{-\frac{pT_{base}}{\mu_{ind}}} = 0.005$, i.e., $\mu_{ind} = 59,850$ hours (or 6.8 years). This is in accordance with MTBFs typically observed on large-scale platforms, which range from a few years to a few dozens of years [18].

In the following paragraphs, we refer to [11] for the details of computing the expectations of execution times. For each task, the Young/Daly period is $W_{YD} = \sqrt{2\frac{\mu_{ind}}{p}C} \approx 20$ hours, and the expected execution time of a single task $\mathbb{E}(T_{1\text{-}task})$ is minimized when only a single checkpoint is taken at the end of the execution. Recall that we always take a checkpoint at the end of the execution for simplification, thus the optimal solution for each task is to take no additional checkpoint. Then, one can derive that $\mathbb{E}(T_{1\text{-}task}) \approx 10.4$.

However, with 300 tasks executing concurrently, one can compute that the expectation of the total time required to complete all tasks is $\mathbb{E}(T_{all\text{-}tasks}) > 14$. The key point here is that the expectation $\mathbb{E}(T_{all\text{-}tasks})$ of the total time required to complete all tasks is far larger than the maximum of the expectations (which in the example all have the same value $\mathbb{E}(T_{1\text{-}task})$).

Because the exit task cannot start before the last parallel task is completed, the expectation of the total execution time of the fork-join application is $\mathbb{E}(T_{total}) = \mathbb{E}(T_{entry}) + \mathbb{E}(T_{all\text{-}tasks}) + \mathbb{E}(T_{exit})$, where $\mathbb{E}(T_{entry})$ and $\mathbb{E}(T_{exit})$ are the expected duration of the entry and exit tasks. Now, when adding four intermediate checkpoints to each task, we obtain $\mathbb{E}(T_{all\text{-}tasks}) < 12.75$. The tasks are then slightly longer (10.5 hours without failure), but the impact of a failure is dramatically reduced if a checkpoint is taken every 2 hours. By diminishing $\mathbb{E}(T_{all\text{-}tasks})$, we save 75 minutes (and in fact much more than that, because the lower and upper bounds for $\mathbb{E}(T_{all\text{-}tasks})$ are loosely computed).

This little example shows that for loosely-coupled applications with a high degree of parallelism, checkpointing each task à la Young/Daly is not good enough! See [11] for a comprehensive analysis and evaluation.

3 TASK GRAPHS

In this section, we deal with non-preemptible, task-based applications. The application is structured as a task graph, or workflow. Each task is atomic and checkpointing is only possible right after the completion of a task. The task graph summarizes the dependencies between the tasks. The problem is then to determine which tasks should be checkpointed. It turns out that optimal, or even efficient, checkpoint strategies are much more difficult to derive than for preemptible applications.

3.1 Baseline

In task-based systems, checkpoint and rollback-recovery have been considered, but the granularity of the task system has motivated a different approach. Since each task represents an atomic application in itself, the inputs of tasks (that are usually the outputs of other tasks) are checkpointed to enable the re-execution of failed tasks.

The de-facto standard approach for task systems is the *checkpoint every task* approach. This approach is inspired by the work done in cloud workflow systems, as is typically done in [66] for a recent example. See [5, 24, 39, 41, 49] for a comprehensive survey of techniques. The outputs of all tasks, which will serve as inputs to other tasks later in the execution, are saved on stable storage as soon as each task completes. The stable storage is typically located in a data center whose disks are accessed by the virtual machines (VMs) that support the execution of the tasks. This approach guarantees that recovering from a failure only requires the re-execution of the task(s) that were executing when the failure stroke; no rollback to previous tasks is needed since their outputs have been checkpointed previously and can be retrieved from the disks.

Of course, checkpointing (the output of) every task may induce a huge overhead, in particular when there are many small tasks and limited I/O bandwidth to stable storage. We outline below a few cases where the optimal solution is known, before coming back to the general case of a workflow whose task graph is arbitrary.

3.2 Linear Chains

The simplest case is when the task graph of the workflow is a linear chain of (parallel) tasks $T_1, T_2, \ldots, T_n$. There is a dependence from T_i to T_{i+1} for $1 \le i \le n-1$. The optimal solution consists in determining which tasks should be checkpointed.

The execution time of T_i is w_i, its size is q_i processors, its checkpoint time C_i, and its recovery time is R_i. Assuming that failures obey an Exponential distribution $\textsc{Exp}(\frac{1}{\mu_{ind}})$, where μ_{ind} is the MTBF of each individual processor, the expected execution time $\mathbb{E}(T_i)$ to execute T_i and to checkpoint it at the end of the execution is well-known; we have:

$$\mathbb{E}(T_i) = \left(\frac{q_i}{\mu_{ind}} + D\right) e^{\frac{q_i}{\mu_{ind}} R_i} \left(e^{-\frac{q_i}{\mu_{ind}}(w_i + C_i)} - 1\right)$$

where D is the downtime (see [12, 34]). The expression for $\mathbb{E}(T_i)$ can be extended for a block of consecutive tasks followed by a checkpoint (simply replace w_i by the execution time of the block). This gives the baseline for a dynamic programming algorithm where one tries to place the first checkpoint at the end of task T_k for $1 \leq k \leq n$ and computes recursively the optimal solution for the remaining sub-chain $T_{k+1}, T_{k+2}, \ldots, T_n$. This is the approach followed by Toueg and Babaoglu [63].

3.3 Iterative Applications

The next problem after a linear chain is that of a *pipelined linear workflow*: we consider now a workflow made of a large number of iterations, each iteration being the same linear chain of parallel tasks. A typical example is an application consisting of an outer loop "`While convergence is not met, do`", and where the loop body includes a sequence of large parallel operations. As in Section 3.2, the objective is to find which task outputs should be saved on stable storage to minimize the expected duration of the whole computation. However, if the workflow consists of, say, ten thousand iterations, each with twenty tasks, one does not want to apply the dynamic programming algorithm of Toueg and Babaoglu [63] to a chain of two hundred thousand tasks!

A natural heuristic is to use the Young/Daly formula and checkpoint at the end of the current task as soon as the total work executed since the last checkpoint exceeds the quantity $\sqrt{2\mu C}$. Unfortunately, even if all tasks may well enroll the same number of processors q and, hence, have the same MTBF $\mu = \frac{\mu_{ind}}{q}$, they are not likely to have the same checkpoint duration C. One can approximate C by the minimum, maximum, or average values of the checkpoint duration of all tasks. This is the heuristic proposed in [26], and its performance is shown satisfactory for a wide range of application scenarios.

As a side note, when the number of iterations is infinite (or very large in practice), it is shown in [26] that there exists an optimal checkpointing strategy that is periodic. It consists of a pattern of task outputs to checkpoint, where this pattern spans over a set of iterations of bounded size. This pattern is repeated over and over throughout the execution. [26] also provides a dynamic programming algorithm, which is polynomial in the number of operations included in the outer loop to compute the optimal periodic checkpoint pattern. The complexity of the algorithm does not depend on the number of iterations of the outer loop. This pattern may well checkpoint many different tasks, across many different iterations. For a workflow with a fixed number of iterations, this periodic strategy is appealing. However, the cost of computing the optimal pattern may be high, and the Young/Daly extension described above may be preferred in some frameworks.

3.4 General Workflows

Another special case is that of a workflow whose dependence graph is arbitrary but whose tasks are parallel tasks that each executes on the whole platform. In other words, the tasks have to be serialized. The problem of ordering the tasks and placing checkpoints is proven NP-complete for simple join graphs in [2], which also introduces several heuristics.

For general workflows, the news is not good either. Consider the problem of scheduling an arbitrary workflow. As mentioned in Section 3.1, the common strategy used in practice is *checkpoint everything*, or CkptAll: all output data of each task is saved onto stable storage. While this strategy leads to fast restarts in case of failures, its downside is that it maximizes checkpointing overhead. At the other end of the spectrum would be a *checkpoint nothing* strategy, or CkptNone, by which all output data is kept in memory (up to memory capacity constraints) and no task is checkpointed. This corresponds to "in-situ" workflow executions, which have been proposed to reduce I/O overhead [68]. The downside is that, in case of a failure, a large number of tasks may have to be re-executed, leading to slow restarts. The objective of an efficient checkpoint strategy is to achieve a desirable trade-off between these two extremes. But the complexity of this problem is steep.

The fundamental difficulty lies in the evaluation of a solution. A solution consists of an ordered list of tasks to execute for each processor, and for each task whether or not to save its output data to stable storage after its execution. In a failure-free execution, the total execution time, or makespan, of a solution is simply the longest path in the DAG, accounting for serialized task executions at each processor. With failures, the makespan becomes a random variable because task execution times are probabilistic, due to failures causing task re-executions. Consider a first simple case with the CkptAll strategy and a solution in which each task is assigned to a different processor. Computing the expected makespan amounts to computing the expected longest path in the schedule. Unfortunately, computing the expected length of the longest path in a DAG with probabilistic task durations is a known difficult problem [30, 48]. Even in the simplified case when task durations are random variables that can take only two discrete values, the problem is #P-complete [30].[1]

Now, at the other extreme, consider a second simple example with the CkptNone strategy and a solution in which each task is assigned to a different processor. Even if each task has the unitary cost and can fail only once, computing the expected makespan is a #P-complete problem again [31]. These two examples show all the difficulty of the problem, even when an ordered list of tasks to execute is already assigned to each processor. Several heuristics to tackle the general problem are proposed in [32].

4 DEALING WITH UNCERTAINTY

This section briefly addresses two scenarios where it is impossible to apply the Young/Daly formula directly, even though the target application is preemptible and tightly coupled as in Section 2. Basically, in the $W_{YD} = \sqrt{2\mu C}$ formula, this is when either μ or C is unknown!

4.1 Unknown MTBF

When the MBTF μ_{ind} of an individual processor is unknown, the MTBF $\mu = \frac{\mu_{ind}}{p}$ of the application is unknown too. There is no other solution than to learn the value of μ by trial and error. The initial guess for μ is arbitrary, say from a few hours to several

[1] Recall that #P is the class of counting problems that correspond to NP decision problems [14, 50, 64], and that #P-complete problems are at least as hard as NP-complete problems.

weeks depending upon the size of the application. Compute W_{YD} accordingly and schedule the first checkpoint. If a failure strikes before this checkpoint, decrease the current estimate of μ. If no failure strikes before this checkpoint, keep the current value for μ and proceed for a few periods of the same length. if there is still no failure at this point, it should be safe to increase the estimate of μ. The rates for decreasing/increasing the current estimate could follow some geometric progression, e.g., the next estimate is either half or twice the current one.

An interesting heuristic is proposed in [56]. The checkpoint period is dynamically adjusted so that the aggregate checkpointing cost always equals the expected rework cost after failure recovery. The intuition follows the discussion in Section 2.2: in the optimal solution, both sources of waste (checkpoint and re-execution) should be balanced.

4.2 Unknown Checkpoint Time (Due to Contention)

This section deals with the scenario where checkpoint cost C is unknown. In fact, this corresponds to a scenario where several applications are executing concurrently on the platform (recall *space-sharing* from Section 2.4). Each application has precise knowledge of the volume of data to be saved but the I/O bandwidth to stable storage that is granted is subject to variations over time. The main reason is contention: consider the simple case where two applications of the same size (number of processors) checkpoint simultaneously a file of the same size (volume). Each application will be assigned half the I/O bandwidth to checkpoint, therefore the commits will take twice as expected. In other words, the checkpoint time of each application is doubled, and the Young/Daly period $\sqrt{2\mu C}$ should have been increased by a factor $\sqrt{2}$; the checkpoint strategy is no longer optimal, and efficiency will decrease.

Several heuristics are described in [35] for this contention problem. Each application attempts to use its Young/Daly period. The I/O token is given to only one application at every time step, and I/O operations cannot be interrupted once started. If several applications post concurrent requests to checkpoint, one will be selected and the other ones will continue their execution. The selection is based upon several criteria, including the time already spent waiting for I/O and the risk incurred by all the applications (increased waste) that have not been selected. See [35] for details.

5 SILENT ERRORS

In this section, we consider another type of error: while all previous sections addressed fail-stop errors, we now deal with silent errors, first in isolation and then in combination with fail-stop errors. It turns out that the Young/Daly formula can be extended to deal with both types of errors.

5.1 Background

We start with some background on silent errors, a.k.a. silent data corruptions (SDCs). While fail-stop errors lead to fatal interruptions (such as a crash) and cause the loss of the entire memory of the processor, silent errors only impact a given process and lead to incorrect results. But a silent error strikes undetected and the processor can continue its execution; sometimes the silent error

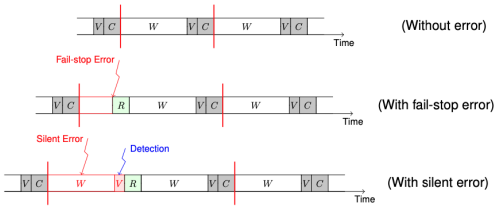

Figure 2: Fails-stop errors versus silent errors.

can be detected and corrected, and some other times it degenerates into a fatal fail-stop error.

Silent errors may be caused, for instance, by soft errors in the L1 cache, arithmetic errors in the ALU (Arithmetic and Logic Unit), or bit flips due to cosmic radiation [44, 47, 70, 71]. Many silent errors caused by one or multiple bits that spontaneously flip to the opposite state are caught by hardware mechanisms such as error correcting codes (ECCs) that have been implemented to protect memory. But in reality, some bit flips still manage to pass undetected [59]. Moreover, processor caches are not protected by ECC in general, but by weaker mechanisms, like simple parity, exposing a higher risk of undetectable error in case of multiple simultaneous bit flips. Buses also often are a weak link in the protecting chain, making all data transfers at higher risk. In addition, the constant need to reduce component size and voltage increases the likelihood of silent errors. In a nutshell, silent errors have become a major threat due to the increase in problem size [58]: the larger the problem, the more memory to be used to store the data, the more frequent the errors, and the higher the probability of overriding ECC protection, generating multiple errors.

A major problem with silent errors is *detection latency*: contrarily to a fail-stop error whose detection is immediate, a silent error is identified only when the corrupted data is activated and/or leads to an unusual application behavior. On the contrary, checkpoint and rollback recovery assumes instantaneous error detection, and this raises a new difficulty: if the error stroke before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted and cannot be used to restore the application. To solve this problem, one may envision keeping several checkpoints in memory and restoring the application from the last *valid* checkpoint, thereby rolling back to the last *correct* state of the application [42]. But even if it was at all possible to store many checkpoints (which is very demanding in memory), one would not know how to identify the last valid one. Some verification mechanism, or detector, must be enforced.

Considerable efforts have been directed at designing such verification mechanisms to reveal silent errors, because error detection is usually very costly. Hardware mechanisms, such as ECC memory, can detect and even correct a fraction of errors, but in practice, they are complemented with software techniques. The only general-purpose method is to replicate the execution of the target computational kernel on two sets of processors and to compare the results of both executions. If they do not coincide, an error has been detected, and the application must be executed a third time. To avoid a-posteriori re-execution, triplication can be enforced, which

allows for error correction in addition to error detection, using a simple majority vote. However, triplication (originally known as triple modular redundancy and voting [43]) is even more costly than replication, which already requires half the resources to execute redundant operations.

Application-specific information can be very useful to enable ad-hoc solutions, which dramatically decreases the cost of detection. Many techniques have been advocated. They include memory scrubbing [38] and ABFT techniques [15, 37, 55], such as coding for the sparse-matrix vector multiplication kernel [55], and coupling a higher-order with a lower-order scheme for PDEs [13]. These methods can only detect an error but do not correct it. Self-stabilizing corrections after error detection in the conjugate gradient method are investigated in [51]. Fault-tolerant iterative solvers for sparse linear algebra are introduced in [17, 20, 36], using extra checks such as re-computing inner products of vectors that should be orthogonal, or even re-computing the residual.

In summary, application-specific detectors are very appealing due to their low cost as compared to replication, but they suffer from some limitations. Most application-specific detectors can only detect errors, not correct them. Next, they are used to detect errors of a certain type, while many types can strike. For instance, with iterative methods, orthogonality tests will detect arithmetic errors but cannot do anything if we start with corrupted data in memory. Worse, even for a given type of error, the detector will not detect all the errors of that type, but only a fraction of them (one says that the detector recall is strictly smaller than one). Finally, ABFT is one of the few methods that enables error correction in addition to detection, but it is currently limited to correcting a single error due to the numerical instability of state-of-the-art methods.

5.2 Optimal Period for Silent Errors

We study a generic solution that is agnostic of the nature of the verification mechanism (replication, checksum, error correcting code, coherence tests, etc.). We assume that we can rely on a fully general-purpose detector, of cost V. The idea is to perform a verification just before taking each checkpoint. If the verification succeeds, then one can safely store the checkpoint. If the verification fails, it means that a silent error has struck since the last checkpoint, which was duly verified, and one can safely recover from that checkpoint to resume the execution of the application.

See Figure 2 for an illustration and comparison with fail-stop errors. If a silent error strikes, it is always detected only at the end of the period, when the verification reveals the error; on the contrary, a fail-stop error strikes in the middle of the period on average and is detected immediately. Note that there is no downtime for silent errors because the processor can continue its execution after a silent error and need not be replaced. For simplification, we have used $D = 0$ in the second row of Figure 2 (with fail-stop error), but recall there is a downtime after a fail-stop error in the general case.

Just as for fail-stop errors, we introduce the MTBE of individual nodes as μ_{ind}^{silent} . Here the MTBE is the Mean Time Between Errors, the counterpart for silent errors of the MBTF for fail-stop errors. The MTBE of an application with p processors will be $\mu^{silent} = \frac{\mu_{ind}^{silent}}{p}$: because the frequency of silent errors is proportional to the number of arithmetic operations executed, and/or to the volume of the memory footprint of the application; the MTBE scales linearly with the size of the application, just as the MTBF does.

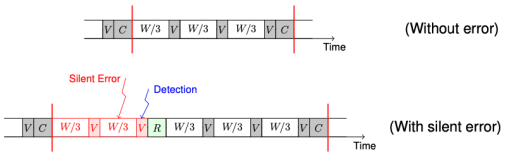

Figure 3: Introducing two intermediate verifications in the period.

Now consider a parallel application of MTBE μ^{silent}. At first sight, one could think that the optimal Young/Daly period for silent errors will be $W_{YD} = \sqrt{2\mu^{silent}(V+C)}$, because we have replaced each checkpoint of cost C by a verified checkpoint of cost $V+C$. However, because a silent error is always detected only at the end of the period, when the verification reveals the error, the formula will be different. With the notations of Section 2.2, the two sources of waste become $S_1 = \frac{V+C}{W+V+C}$ and $S_2 = \frac{1}{\mu^{silent}}(R+W+V)$. Altogether, both sources of waste approximately add up, so we have to find W that minimizes $S_1 + S_2$. Simplifying as before, we obtain $S_1 + S_2 \approx \frac{V+C}{W} + \frac{W}{\mu^{silent}}$, which is minimized for

$$W_{YD} = \sqrt{\mu^{silent}(V+C)} \tag{2}$$

Equation (2) is the Young/Daly formula for silent errors!

5.3 Extensions

A first natural extension is to deal with both fail-stop and silent errors. Indeed, both sources of errors are likely to strike simultaneously when executing a parallel application. In that case, the failure-free waste $S_1 = \frac{V+C}{W+V+C}$ remains unchanged but the failure-induced waste should be updated to account for both error types:

$$S_2 = \frac{1}{\mu^{fail}}\left(D + R + \frac{1}{2}(W+V+C)\right) + \frac{1}{\mu^{silent}}(R+W+V)$$

Here for clarity, we have used μ^{fail} instead of simply μ for the MTBF of the application. Simplifying again, we obtain that the total waste is minimized for

$$W_{YD} = \sqrt{\frac{1}{\frac{1}{2\mu^{fail}} + \frac{1}{\mu^{silent}}}(V+C)} \tag{3}$$

Equation (3) is the Young/Daly formula for fail-stop and silent errors combined! We check that Equation (3) reduces to Equation (1) when $\mu^{silent} = \infty$ and $V = 0$ (only fail-stop errors) and to Equation (2) when $\mu^{fail} = \infty$ (only silent errors). The general case uses the harmonic mean of μ^{fail} and μ^{silent} weighted by the average proportion of re-execution time in a period when struck by an error.

The second extension applies when application-specific information enables to decrease the cost of a verification well below the cost of a checkpoint, i.e., when $V \ll C$. In that case, it is useful to insert some intermediate verifications within the period to detect silent errors early on. Assume that we deal with silent errors only and see Figure 3 for an example of a period with 2 intermediate verifications (and a third one at the end of the period to verify the checkpoint).

The failure-free waste S_1 is increased to $S_1 = \frac{3V+C}{W+3V+C} \approx \frac{3V+C}{W}$. However, the failure-induced waste is reduced to

$$\begin{aligned} S_2 &= \frac{1}{\mu^{silent}} \left[\frac{1}{3}(R + \frac{W}{3} + V) + \frac{1}{3}(R + \frac{2W}{3} + 2V) + \frac{1}{3}(R + W + 3V) \right] \\ &\approx \frac{1}{\mu^{silent}} \frac{(1+2+3)W}{9} = \frac{2W}{3\mu^{silent}} \end{aligned}$$

To see this, with equal probability $\frac{1}{3}$, the silent error will strike either third of the pattern, and re-execution will cost either $\frac{W}{3}$ (first third), or $\frac{2W}{3}$ (second third), or W (last third). This leads to $S_1 + S_2$ minimized for $W = \sqrt{\frac{3}{2}\mu^{silent}(3V + C)}$ and we get $(S_1 + S_2)_{\min} = 2\sqrt{\frac{2(3V+C)}{3\mu^{silent}}}$ for that value. In comparison, without intermediate verification, we had $(S_1 + S_2)_{\min} = 2\sqrt{\frac{V+C}{\mu^{silent}}}$. We check that adding two intermediate verifications is better than none as long as $V \leq \frac{C}{3}$. This is very likely to be the case with an application-specific detector.

We refer to [10] for the analysis of more general patterns. Also, we have assumed a perfect verification mechanism, while all application detectors have a limited recall (ratio of missed errors, or false negatives) and a limited precision (ratio of detected errors that are in fact not errors, or false positives). One can have an arsenal of several detectors of different recall, precision, and cost that can be used; see [6] for choosing the best approach.

6 CONCLUSION

This survey has dealt with checkpointing policies based upon the Young/Daly period and has assessed its usefulness and robustness together with its limitations. We discuss some extensions and open problems in this conclusion.

For preemptible applications (Section 2), we have focused on coordinated checkpointing onto stable storage, but most of the results hold for other methods that reduce checkpoint overhead, such as in-memory checkpointing [25, 46, 69], two-level checkpointing [23, 57] and multi-level checkpointing [8, 9, 22, 45].

For task-based applications, one could envision an extension of coordinated checkpointing designed for such systems. Using periodic coordinated checkpointing to decide which tasks to checkpoint in a distributed task system consists of finding a period between two checkpoint waves, and coordinating all the processes of the application to checkpoint their state. Applying this heuristic to a task-based system does not ensure optimal performance, because the amount of data to checkpoint depends on the number and input of the ready tasks and varies over time, which is outside the assumptions of the periodic checkpointing approach. However, by continuously adapting the period to the amount of work executed (either maximal or averaged across all processors), this strategy may provide an efficient solution in scenarios where tasks are small and where failures are rare.

For both application models (preemptible and task-based), we have assumed failure independence. Indeed, the standard model assumes IID failure inter-arrival times, or IATs, on each node, with a common distribution $\mathcal{D}$. As for *temporal* dependence, it has been observed many times that when a failure occurs, it may trigger other failures that will strike different system components [7, 33, 62]. As an example, a failing cooling system may cause a series of successive crashes of different nodes. Also, an outstanding error in the file system will likely be followed by several others [40, 54]. As for *spatial* dependence, it is clear that the overheating of some node in a cabinet is quite likely to be followed by the overheating of neighbor nodes (which comes atop of a temporal dependence as well!). Bautista-Gomez et al. [7] have studied nine systems, and they report periods of high failure density in all of them. They call these periods *cascade failures*. This observation has led them to revisit the temporal failure independence assumption, and to design bi-periodic checkpointing algorithms that use different periods in normal (failure-free) and degraded (with failure cascades) modes. [62] introduces a dynamic strategy called *lazy checkpointing* to adjust to changes in the failure rate. Another approach has been proposed in [4], using quantiles of consecutive IAT pairs. It is an open problem to derive an efficient checkpoint strategy that can account for temporal or spatial dependence between failures. For example, spatial dependence calls for a variant of in-memory checkpointing where the buddy of a processor (acting replica of a checkpoint) is chosen far away from that processor, while it is better to select a physical neighbor to optimize communication overhead if failures are truly independent. Complicated trade-offs must be achieved.

Finally, for silent errors, the first open problem remains which detector to use and when. Some parts of the application are critical (such as its execution code) and must be protected at all costs while some other parts (like non-critical data) may be loosely and infrequently verified by cheap mechanisms; we speak of *selective reliability* in such a framework. More generally, *trustworthy computing* is the problem of guaranteeing, at least with some high probability, that the final results of a parallel application are correct. The higher the flop count and the larger the data footprint, the more challenging to achieve this goal!

REFERENCES

[1] S. Arunagiri, J. T. Daly, and P. J. Teller. Modeling and Analysis of Checkpoint I/O Operations. In *Analytical and Stochastic Modeling Techniques and Applications: 17th International Conference*, pages 387–399. Springer, 2010.

[2] G. Aupy, A. Benoit, H. Casanova, and Y. Robert. Scheduling computational workflows on failure-prone platforms. *Int. J. of Networking and Computing*, 6(1):2–26, 2016.

[3] G. Aupy, A. Benoit, T. Hérault, and Y. Robert. Optimal checkpointing period: time vs. energy. In *PMBS 2013, the 4th Int. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. LNCS Springer Verlag, 2013.

[4] G. Aupy, Y. Robert, and F. Vivien. Assuming failure independence: are we right to be wrong? In *FTS'2017*, 2017.

[5] A. Bala and I. Chana. Fault tolerance-challenges, techniques and implementation in cloud computing. *International Journal of Computer Science Issues (IJCSI)*, 9(1):288, 2012.

[6] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. Raina, Y. Robert, and H. Sun. Coping with recall and precision of soft error detectors. *J. Parallel and Distributed Computing*, 98:8–24, 2016.

[7] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, and M. Snir. Reducing waste in extreme scale systems through introspective analysis. In *IPDPS*, pages 212–221. IEEE, 2016.

[8] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Proc. SC'11*, 2011.

[9] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. Towards optimal multi-level checkpointing. *IEEE Trans. Computers*, 66(7):1212–1226, 2017.

[10] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. *ACM Trans. Parallel Computing*, 3(2), 2016.

[11] A. Benoit, L. Perotin, Y. Robert, and H. Sun. Checkpointing Workflows à la Young/Daly Is Not Good Enough. Research report RR-9413, INRIA, 2021. Available at https://hal.inria.fr/hal-03264047/.

[12] A. Benoit, L. Perotin, Y. Robert, and F. Vivien. Checkpointing strategies to protect parallel jobs from non-memoryless fail-stop errors. Research report RR-9465, INRIA, 2022. Available at https://hal.inria.fr/hal-03610883.

[13] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *Int. J. High Performance Computing Applications*, 2014.

[14] H. L. Bodlaender and T. Wolle. A note on the complexity of network reliability problems. *IEEE Trans. Inf. Theory*, 47:1971–1988, 2004.

[15] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, 2009.

[16] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Proc. of SC'11*, 2011.

[17] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *ICS*. ACM, 2008.

[18] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

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

[20] Z. Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proc. PPoPP*, pages 167–176, 2013.

[21] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *FGCS*, 22(3):303–312, 2006.

[22] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *IPDPS*. IEEE, 2014.

[23] S. Di, Y. Robert, F. Vivien, and F. Cappello. Toward an optimal online checkpoint solution under a two-level HPC checkpoint model. *IEEE Trans. Parallel & Distributed Systems*, 2016.

[24] Y. Ding, G. Yao, and K. Hao. Fault-tolerant elastic scheduling algorithm for workflow in cloud systems. *Information Sciences*, 393:47–65, 2017.

[25] J. Dongarra, T. Hérault, and Y. Robert. Performance and reliability trade-offs for the double checkpointing algorithm. *Int. J. of Networking and Computing*, 4(1):23–41, 2014.

[26] Y. Du, G. Pallez, L. Marchal, and Y. Robert. Optimal checkpointing strategies for iterative applications. *IEEE Trans. Parallel Distributed Systems*, 33(3):507–522, 2022.

[27] N. El-Sayed and B. Schroeder. To checkpoint or not to checkpoint: Understanding energy-performance-i/o tradeoffs in hpc checkpointing. In *CLUSTER*, pages 93–102, 2014.

[28] 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 *SC'11*. ACM, 2011.

[29] E. Gelenbe, P. Boryszko, M. Siavvas, and J. Domanska. Optimum checkpoints for time and energy. In *28th MASCOTS*, pages 1–8. IEEE, 2020.

[30] J. N. Hagstrom. Computational complexity of PERT problems. *Networks*, 18(2):139–147, 1988.

[31] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien. Checkpointing workflows for fail-stop errors. *IEEE Trans. Computers*, 67(8):1105–1120, 2018.

[32] L. Han, V. Le Fèvre, L.-C. Canon, Y. Robert, and F. Vivien. A generic approach to scheduling and checkpointing workflows. In *ICPP'2018, the 47th Int. Conf. on Parallel Processing*, 2018.

[33] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *Proc. SC'11*, 2011.

[34] T. Herault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag, 2015.

[35] T. Herault, Y. Robert, A. Bouteiller, D. Arnold, K. B. Ferreira, G. Bosilca, and J. Dongarra. Checkpointing strategies for shared high-performance computing platforms. *International Journal of Networking and Computing*, 9(1):28–52, 2019.

[36] M. Heroux and M. Hoemmen. Fault-tolerant iterative methods via selective reliability. Research report SAND2011-3915 C, Sandia Nat. Lab., 2011.

[37] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, 1984.

[38] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. *SIGARCH Comput. Archit. News*, 40(1):111–122, 2012.

[39] G. Kandaswamy, A. Mandal, and D. A. Reed. Fault tolerance and recovery of scientific workflows on computational grids. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 777–782. IEEE, 2008.

[40] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proc. 1st ACM Symposium on Cloud Computing*, SoCC '10. ACM, 2010.

[41] P. Kumari and P. Kaur. A survey of fault tolerance in cloud computing. *Journal of King Saud University - Computer and Information Sciences*, 2018.

[42] G. Lu, Z. Zheng, and A. A. Chien. When is multi-version checkpointing needed? In *Proc. 3rd Workshop on Fault-tolerance for HPC at extreme scale (FTXS)*, pages 49–56, 2013.

[43] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, 1962.

[44] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC*. ACM, 2010.

[45] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. SC'10*, 2010.

[46] 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*, pages 364–372. IEEE Computer Society, 2012.

[47] T. O'Gorman. The effect of cosmic rays on the soft error rate of a DRAM at ground level. *IEEE Trans. Electron Devices*, 41(4):553–557, 1994.

[48] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 5th edition, 2016.

[49] S. Prathiba and S. Sowvarnica. Survey of failures and fault tolerance in cloud. In *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, pages 169–172, 2017.

[50] J. S. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comp.*, 12(4):777–788, 1983.

[51] P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *ScalA '13*, 2013.

[52] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. of DSN*, pages 249–258, 2006.

[53] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[54] K. Schroiff, P. Gemsjaeger, and C. Bolik. Cascading failover of a data management application for shared disk file systems in loosely coupled node clusters, 2006. US Patent 6,990,606.

[55] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *ICS*. ACM, 2012.

[56] P. Sigdel, X. Yuan, and N. Tzeng. Realizing best checkpointing control in computing systems. *IEEE TPDS*, 32(2):315–329, 2021.

[57] L. Silva and J. Silva. Using two-level stable storage for efficient checkpointing. *IEE Proceedings - Software*, 145(6):198–202, 1998.

[58] M. Snir and et al. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, 2014.

[59] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *20th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–310. ACM, 2015.

[60] O. Subasi, G. Kestor, and S. Krishnamoorthy. Toward a general theory of optimal checkpoint placement. In *CLUSTER*, pages 464–474. IEEE, 2017.

[61] O. Subasi, T. Martsinkevich, F. Zyulkyarov, O. Unsal, J. Labarta, and F. Cappello. Unified fault-tolerance framework for hybrid task-parallel message-passing applications. *IJHPCA*, 32(5):641–657, 2018.

[62] D. Tiwari, S. Gupta, and S. S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *44th Int. Conf. on Dependable Systems and Networks*, pages 25–36. IEEE, 2014.

[63] S. Toueg and O. Babaoğlu. On the optimum checkpoint selection problem. *SIAM J. Comput.*, 13(3), 1984.

[64] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[65] O. Weidner, M. Atkinson, A. Barker, and R. Filgueira Vicente. Rethinking high performance computing platforms: Challenges, opportunities and recommendations. In *Proc. Data-Intensive Distributed Computing DIDC*. ACM, 2016.

[66] X. Xu, R. Mo, F. Dai, W. Lin, S. Wan, and W. Dou. Dynamic resource provisioning with fault tolerance for data-intensive meteorological workflows in cloud. *IEEE Transactions on Industrial Informatics*, 16(9):6172–6181, 2019.

[67] J. W. Young. A first order approximation to the optimum checkpoint interval. *Comm. of the ACM*, 17(9):530–531, 1974.

[68] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi. Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform. In *Proc. 26th IEEE Int. Parallel and Distributed Processing Symposium*, pages 1352–1363, 2012.

[69] 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*, pages 93–103. IEEE Computer Society, 2004.

[70] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, and C. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *IEEE Journal of Solid-State Circuits*, 33(2):246–252, 1998.

[71] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. IBM experiments in soft fails in computer electronics. *IBM J. Res. Dev.*, 40(1):3–18, 1996.