Checkpointing strategies for parallel jobs.

Marin Bougeret, Henri Casanova, Mikael Rabie, Yves Robert, Frédéric Vivien

Checkpointing strategies for parallel jobs

Marin Bougeret
ENS Lyon, France
Marin.Bougeret@ens-lyon.fr

Henri Casanova
Univ. of Hawai'i at Mānoa, Honolulu, USA
henric@hawaii.edu

Mikael Rabie
ENS Lyon, France
Mikael.Rabie@ens-lyon.fr

Yves Robert*
ENS Lyon, France
Yves.Robert@ens-lyon.fr

Frédéric Vivien
INRIA, Lyon, France
Frederic.Vivien@inria.fr

ABSTRACT

This work provides an analysis of checkpointing strategies for minimizing expected job execution times in an environment that is subject to processor failures. In the case of both sequential and parallel jobs, we give the optimal solution for exponentially distributed failure inter-arrival times, which, to the best of our knowledge, is the first rigorous proof that periodic checkpointing is optimal. For non-exponentially distributed failures, we develop a dynamic programming algorithm to maximize the amount of work completed before the next failure, which provides a good heuristic for minimizing the expected execution time. Our work considers various models of job parallelism and of parallel checkpointing overhead. We first perform extensive simulation experiments assuming that failures follow Exponential or Weibull distributions, the latter being more representative of real-world systems. The obtained results not only corroborate our theoretical findings, but also show that our dynamic programming algorithm significantly outperforms previously proposed solutions in the case of Weibull failures. We then discuss results from simulation experiments that use failure logs from production clusters. These results confirm that our dynamic programming algorithm significantly outperforms existing solutions for real-world clusters.

Keywords: Fault-tolerance, checkpointing, sequential job, parallel job.

1. INTRODUCTION

Resilience is a key challenge for post-petascale high-performance computing (HPC) systems [11, 25] since failures are increasingly likely to occur during the execution of parallel jobs that enroll increasingly large numbers of processors. For instance, the 45,208-processor Jaguar platform is reported to experience on the order of 1 failure per day [21, 2]. Faults that cannot be automatically detected and corrected in hardware lead to failures. In this case, rollback recovery is used to resume job execution from a previously saved fault-free execution state, or *checkpoint*. Rollback recovery implies frequent (usually periodic) *checkpointing* events at which the job state is saved to resilient storage. More frequent checkpoints lead to higher overhead during fault-free execution, but less frequent checkpoints lead to a larger loss when a failure occurs. The design of efficient *checkpointing strategies*, which specify when checkpoints should be taken, is thus key to high performance.

We study the problem of finding a checkpointing strategy that minimizes the expectation of a job's execution time, or *expected makespan*. In this context, our novel contributions are as follows. For sequential jobs, we provide the optimal solution for exponential failures and an accurate dynamic programming algorithm for general failures. The optimal solution for exponential failures, i.e., periodic checkpointing, is widely known in the "folklore" but, to the best of our knowledge, we provide the first rigorous proof. Our dynamic programming algorithm provides the first accurate solution of the expected makespan minimization problem with Weibull failures, which are representative of the behavior of real-world platforms [13, 26, 20]. For parallel jobs, we consider a variety of execution scenarios with different models of job parallelism (embarrassingly parallel jobs, generic parallel jobs, and typical numerical kernels such as matrix product or LU decomposition), and with different models of the overhead of checkpointing a parallel job (which may or may not depend on the total number of processors in use). In the case of Exponential failures we provide the optimal solution. In the case of general failures, since minimizing the expected makespan is computationally difficult, we instead provide a dynamic programming algorithm to maximize the amount of work successfully completed before the next failure. This approach turns out to provide a good heuristic solution to the expected makespan minimization problem. In particular, it significantly outperforms previously proposed solutions in the case of Weibull failures.

Sections 2 and 3 give theoretical results for sequential and parallel jobs, respectively. Section 4 presents our simulation methodology. Sections 5 and 6 discuss simulation results when using synthetic failure distributions and when using real-world failure data, respectively. Section 7 reviews related work. Finally, Section 8 concludes the paper with a summary of our findings and a discussion of future direc-

*Yves Robert is with the Institut Universitaire de France and with the University of Tennessee Knoxville. This work was supported in part by the ANR StochaGrid and RESCUE projects, and by the INRIA-Illinois Joint Laboratory for Petascale Computing.

SC11, November 12-18, 2011, Seattle, Washington, USA

tions. Due to lack of space, all technical proofs are omitted but available in a companion research report [5].

2. SEQUENTIAL JOBS

2.1 Problem statement

We consider an application, or *job*, that executes on one *processor*. We use the term processor to indicate any individually scheduled compute resource (a core, a multi-core processor, a cluster node) so that our work is agnostic to the granularity of the platform. The job must complete $\mathcal{W}$ units of (divisible) work, which can be split arbitrarily into separate *chunks*. The job state is checkpointed after the execution of every chunk. Defining the sequence of chunk sizes is therefore equivalent to defining the checkpointing dates. We use C to denote the time needed to perform a checkpoint. The processor is subject to *failures*, each causing a *downtime* period, of duration D, followed by a *recovery* period, of duration R. The downtime accounts for software rejuvenation (i.e., rebooting [16, 9]) or for the replacement of the failed processor by a spare. Regardless, we assume that after a downtime the processor is fault-free and begins a new lifetime at the beginning of the recovery period. This period corresponds to the time needed to restore the last checkpoint. Note that although C and R can fluctuate depending on cluster and network load, as in most works in the field we assume they are constant. We asssume coordinated checkpointing [30], meaning that no message logging/replay is needed when recovering from failures. Finally, we assume that failures can happen during recovery or checkpointing, but not during a downtime (otherwise, the downtime period could be considered part of the recovery period).

We study two optimization problems:

- MAKESPAN: Minimize the job's expected makespan;
- NEXTFAILURE: Maximize the expected amount of work completed before the next failure.

Solving MAKESPAN is our main goal. NEXTFAILURE amounts to optimizing the makespan on a "failure-by-failure" basis, selecting the next chunk size as if the next failure were to imply termination of the execution. Intuitively, solving NEXTFAILURE should lead to a good approximation of the solution to MAKESPAN, at least for large job sizes $\mathcal{W}$. Therefore, we use the solution of NEXTFAILURE in cases for which we are unable to solve MAKESPAN directly. We give formal definitions for both problems in the next section.

2.2 Formal problem definitions

We consider the processor from time t_0 onward. Failures occur at times $(t_n)_{n\geq 1}$, with $t_n = t_0 + \sum_{m=1}^{n} X_m$, where the random variables $(X_m)_{m\geq 1}$ are *iid* (independent and identically distributed). Given a current time $t > t_0$, we define $n(t) = \min\{n | t_n \geq t\}$, so that $X_{n(t)}$ corresponds to the inter-failure interval in which t falls. We use $P_{suc}(x|\tau)$ to denote the probability that the processor does not fail for the next x units of time, knowing that the last failure occurred τ units of time ago. In other words, if $X = X_{n(t)}$ denotes the current inter-arrival failure interval,

$$P_{suc}(x|\tau) = \mathbb{P}(X \geq \tau + x \mid X \geq \tau) \ .$$

Note that we do *not* assume that the failure stochastic process is memoryless.

For each problem stated in the previous section, a solution is fully defined by a function $f(\omega|\tau)$ that returns the size of the next chunk to execute given the amount of work ω that has not yet been executed successfully ($f(\omega|\tau) \leq \omega \leq \mathcal{W}$) and the amount of time τ elapsed since the last failure. f is invoked at each decision point, i.e., after each checkpoint or recovery. Our goal is to determine a function f that defines an optimal solution. Assuming a unit-speed processor without loss of generality, the time needed to execute a chunk of size ω is $\omega + C$ if no failure occurs.

Definition of MAKESPAN– For a given amount of work ω and a time elapsed since the last failure τ, we define $T(\omega|\tau)$ as the random variable that quantifies the time needed for executing ω units of work. Given a solution function f, let $\omega_1 = f(\omega|\tau)$ denote the size of the first chunk. We can write the following recursion:

$$\begin{aligned} &T(0|\tau) \ = 0 \\ &T(\omega|\tau) = \\ &\quad \begin{cases} \omega_1 + C + T(\omega - \omega_1 | \tau + \omega_1 + C) \\ \quad \text{if the processor does not fail during} \\ \quad \text{the next } \omega_1 + C \text{ units of time,} \\ T_{wasted}(\omega_1 + C|\tau) + T(\omega|R) \\ \quad \text{otherwise.} \end{cases} \end{aligned} \tag{1}$$

The two cases above are explained as follows:

- If the processor does not fail during the execution and checkpointing of the first chunk (i.e., for $\omega_1 + C$ time units), there remains to execute a work of size $\omega - \omega_1$ and the time since the last failure is $\tau + \omega_1 + C$;
- If the processor fails before successfully completing the first chunk and its checkpoint, then some additional delays are incurred, as captured by the variable $T_{wasted}(\omega_1 + C|\tau)$. The time wasted corresponds to the execution up to the failure, a downtime, and a recovery during which a failure may happen. We compute T_{wasted} in the next section. Regardless, once a successful recovery has been completed, there still remain ω units of work to execute, and the time since the last failure is simply R.

We define MAKESPAN formally as: find f that minimizes $\mathbb{E}(T(\mathcal{W}|\tau_0))$, where $\mathbb{E}(X)$ denotes the expectation of the random variable X, and τ_0 the time elapsed since the last failure before t_0.

Definition of NEXTFAILURE– For a given amount of work ω and a time elapsed since the last failure τ, we define $W(\omega|\tau)$ as the random variable that quantifies the amount of work successfully executed before the next failure. Given a solution function f, let $\omega_1 = f(\omega|\tau)$ denote the size of the first chunk. We can write the following recursion:

$$\begin{aligned} &W(0|\tau) \ = 0 \\ &W(\omega|\tau) = \\ &\quad \begin{cases} \omega_1 + W(\omega - \omega_1 | \tau + \omega_1 + C) \\ \quad \text{if the processor does not fail during} \\ \quad \text{the next } \omega_1 + C \text{ units of time,} \\ 0 \quad \text{otherwise.} \end{cases} \end{aligned} \tag{2}$$

This recursion is simpler than the one for MAKESPAN because a failure during the computation of the first chunk means that no work (i.e., no fraction of ω) will have been successfully executed before the next failure. We define NEXTFAILURE formally as: find f that maximizes $\mathbb{E}(W(\mathcal{W}|\tau_0))$.

2.3 Solving MAKESPAN

A challenge for solving MAKESPAN is the computation of $T_{wasted}(\omega_1 + C|\tau)$. We rely on the following decomposition:

$$T_{wasted}(\omega_1 + C|\tau) = T_{lost}(\omega_1 + C|\tau) + T_{rec}\ , \qquad \text{where}$$

- $T_{lost}(x|\tau)$ is the amount of time spent computing before a failure, knowing that the next failure occurs within the next x units of time, and that the last failure has occurred τ units of time ago.
- T_{rec} is the amount of time needed by the system to recover from the failure (accounting for the fact that other failures may occur during recovery).

PROPOSITION 1. *The* MAKESPAN *problem is equivalent to finding a function f minimizing the following quantity:*

$$\begin{aligned}&\mathbb{E}(T(\mathcal{W}|\tau)) = \\ &P_{suc}(\omega_1 + C|\tau)\Big(\omega_1 + C + \mathbb{E}(T(\mathcal{W} - \omega_1|\tau + \omega_1 + C))\Big) \\ &+(1 - P_{suc}(\omega_1 + C|\tau))\Big(\mathbb{E}(T_{lost}(\omega_1 + C|\tau)) \\ &\qquad + \mathbb{E}(T_{rec}) + \mathbb{E}(T(\mathcal{W}|R))\Big)\end{aligned} \tag{3}$$

where $\omega_1 = f(\mathcal{W}|\tau)$ and where $\mathbb{E}(T_{rec})$ is given by

$$\mathbb{E}(T_{rec}) = D + R + \frac{1 - P_{suc}(R|0)}{P_{suc}(R|0)}(D + \mathbb{E}(T_{lost}(R|0))).$$

2.3.1 Results for the Exponential distribution

In this section we assume that the failure inter-arrival times follow an Exponential distribution with parameter λ, i.e., each $X_n = X$ has probability density $f_X(t) = \lambda e^{-\lambda t} dt$ and cumulative distribution $F_X(t) = 1 - e^{-\lambda t}$ for all $t \geq 0$. The advantage of the Exponential distribution, exploited time and again in the literature, is its "memoryless" property: the time at which the next failure occurs does not depend on the time elapsed since the last failure occurred. Therefore, in this section, we simply write $T(\omega)$, $T_{lost}(\omega)$, and $P_{suc}(\omega)$ instead of $T(\omega|\tau)$, $T_{lost}(\omega|\tau)$, and $P_{suc}(\omega|\tau)$.

LEMMA 1. *With the Exponential distribution:*

$$\mathbb{E}(T_{lost}(\omega)) = \frac{1}{\lambda} - \frac{\omega}{e^{\lambda\omega} - 1} \qquad \text{and}$$
$$\mathbb{E}(T_{rec}) = D + R + \frac{1 - e^{-\lambda R}}{e^{-\lambda R}}(D + \mathbb{E}(T_{lost}(R))).$$

The memoryless property makes it possible to solve the MAKESPAN problem analytically:

THEOREM 1. *Let $\mathcal{W}$ be the amount of work to execute on a processor with failure inter-arrival times that follow an Exponential distribution with parameter λ. Let $K_0 = \frac{\lambda\mathcal{W}}{1 + \mathbb{L}(-e^{-\lambda C - 1})}$ where $\mathbb{L}$, the Lambert function, is defined as $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$. Then the optimal strategy to minimize the expected makespan is to split $\mathcal{W}$ into $K^* = \max(1, \lfloor K_0 \rfloor)$ or $K^* = \lceil K_0 \rceil$ same-size chunks, whichever leads to the smaller value. The optimal expectation of the makespan is:*

$$\mathbb{E}(T^*(\mathcal{W})) = K^* \left(e^{\lambda R}\left(\frac{1}{\lambda} + D\right)\right)\left(e^{\lambda(\frac{\mathcal{W}}{K^*} + C)} - 1\right).$$

Although periodic checkpoints have been widely used in the literature, Theorem 1 is, to the best of our knowledge, the first proof that the optimal deterministic strategy uses a finite number of chunks and is periodic. The proof (fully detailed in [5]) is technical and proceeds along the following steps:

Algorithm 1: DPMAKESPAN (x,b,y,τ_0)

```
if x = 0 then
    return 0
if solution[x][b][y] = unknown then
    best ← ∞;        τ ← bτ_0 + yu
    for i = 1 to x do
        exp_succ ← first(DPMAKESPAN(x − i, b, y + i + C/u, τ_0))
        exp_fail ← first(DPMAKESPAN(x, 0, R/u, τ_0))
        cur ← P_suc(iu + C|τ)(iu + C + exp_succ)
              +(1 − P_suc(iu + C|τ))( E(T_lost(iu + C, τ))
                                      + E(T_rec) + exp_fail )
        if cur < best then
            best ← cur;        chunksize ← i
    solution[x][b][y] ← (best, chunksize)
return solution[x][b][y]
```

- all possible executions for any given f use the same sequence of chunk sizes;
- the optimal strategy uses only a finite number of chunk sizes;
- by a convexity argument, the expected makespan is minimized when all these chunk sizes are equal; and
- the optimization problem is solved by differentiating the objective function.

Note that the checkpointing strategy in Theorem 1 can be shown to be optimal among all deterministic and non-deterministic strategies, as a consequence of Proposition 4.4.3 in [24].

2.3.2 Results for arbitrary distributions

Solving the MAKESPAN problem for arbitrary distributions is difficult because, unlike in the memoryless case, there is no reason for the optimal solution to use a single chunk size [27]. In fact, the optimal solution is very likely to use chunk sizes that depend on additional information that becomes available during the execution (i.e., failure occurrences to date). Using Proposition 1, we can write

$$\begin{aligned}&\mathbb{E}(T^*(\mathcal{W}|\tau)) = \\ &\min_{0 < \omega_1 \leq \mathcal{W}} \left(\begin{array}{l} P_{suc}(\omega_1 + C|\tau)\Big(\omega_1 + C + \mathbb{E}(T^*(\mathcal{W} - \omega_1|\tau + \omega_1 + C))\Big) \\ +(1 - P_{suc}(\omega_1 + C|\tau)) \times \\ \qquad (\mathbb{E}(T_{lost}(\omega_1 + C|\tau)) + \mathbb{E}(T_{rec}) + \mathbb{E}(T^*(\mathcal{W}|R)) \end{array} \right)\end{aligned}$$

which can be solved via dynamic programming. We introduce a time quantum u, meaning that all chunk sizes ω_i are integer multiples of u. This restricts the search for an optimal execution to a finite set of possible executions. The trade-off is that a smaller value of u leads to a more accurate solution, but also to a higher number of states in the algorithm, hence to a higher compute time.

PROPOSITION 2. *Using a time quantum u, and for any failure inter-arrival time distribution,* DPMAKESPAN *(Algorithm 1), called with parameters $(\mathcal{W}/u, 1, 0, \tau_0)$, computes an optimal solution to* MAKESPAN *in time $\mathcal{O}(\frac{\mathcal{W}}{u}^3(1 + \frac{C}{u})a)$, where a is an upper bound on the time needed to compute $E(T_{lost}(\omega|t))$, for any ω and t.*

Algorithm 1 provides an approximation of the optimal solution to the MAKESPAN problem.We evaluate this approximation experimentally in Section 5, including a direct comparison with the optimal solution in the case of Exponential failures (in which case the optimal can be computed via Theorem 1).

Algorithm 2: DPNextFailure (x,n,τ_0)

```
if x = 0 then
  return 0
if solution[x][n] = unknown then
  best ← ∞
  τ ← τ0 + (W − xu) + nC
  for i = 1 to x do
    work = first(DPNextFailure(x − i, n + 1, τ0))
    cur ← P_suc(iu + C|τ) × (iu + work)
    if cur < best then
      best ← cur;      chunksize ← i
  solution[x][n] ← (best, chunksize)
return solution[x][n]
```

2.4 Solving NextFailure

Weighting the two cases in Equation 2 by their probabilities of occurrence, we obtain the expected amount of work successfully computed before the next failure:

$$\mathbb{E}(W(\omega|\tau)) = P_{suc}(\omega_1 + C|\tau)(\omega_1 + \mathbb{E}(W(\omega - \omega_1|\tau + \omega_1 + C))).$$

Here, unlike for Makespan, the objective function to be maximized can easily be written as a closed form, even for arbitrary distributions. Developing the expression above leads to the following result:

Proposition 3. *The* NextFailure *problem is equivalent to maximizing the following quantity:*

$$\mathbb{E}(W(\mathcal{W}|\tau_0)) = \sum_{i=1}^{K} \omega_i \times \prod_{j=1}^{i} P_{suc}(\omega_j + C|t_j) \ ,$$

where $t_j = \tau_0 + \sum_{\ell=1}^{j-1}(\omega_\ell + C)$ is the total time elapsed (without failure) before the start of the execution of chunk ω_j, and K is the (unknown) target number of chunks.

Unfortunately, there does not seem to be an exact solution to this problem. However, just as for the Makespan problem, the recursive definition of $\mathbb{E}(W(\mathcal{W}|\tau))$ lends itself naturally to a dynamic programming algorithm. The dynamic programming scheme is simpler because the size of the i-th chunk is only needed when no failure has occurred during the execution of the first $i-1$ chunks, regardless of the value of the τ parameter. More formally:

Proposition 4. *Using a time quantum u, and for any failure inter-arrival time distribution,* DPNextFailure *(Algorithm 2), called with parameters $(\mathcal{W}/u, 0, \tau_0)$, computes an optimal solution to* NextFailure *in time $\mathcal{O}(\frac{\mathcal{W}}{u}^3)$.*

3. PARALLEL JOBS

3.1 Problem statement

We now turn to parallel jobs that can execute on any number of processors, p. We consider the following relevant scenarios for checkpointing/recovery overheads and for parallel execution times.

Checkpointing/recovery overheads – Checkpoints are synchronized over all processors. We use $C(p)$ and $R(p)$ to denote the time for saving a checkpoint and for recovering from a checkpoint on p processors, respectively (we assume that the downtime D does not depend on p). Assuming that the application's memory footprint is V bytes, with each processor holding V/p bytes, we consider two scenarios:

- Proportional overhead: $C(p) = R(p) = \alpha V/p$ for some constant α. This is representative of cases in which the bandwidth of the network card/link at each processor is the I/O bottleneck.
- Constant overhead: $C(p) = R(p) = \alpha V$, which is representative of cases in which the bandwidth to/from the resilient storage system is the I/O bottleneck.

Parallel work – Let $\mathcal{W}(p)$ be the time required for a failure-free execution on p processors. We use three models:

- Embarrassingly parallel jobs: $\mathcal{W}(p) = \mathcal{W}/p$.
- Generic parallel jobs: $\mathcal{W}(p) = \mathcal{W}/p + \gamma\mathcal{W}$. As in Amdahl's law [1], $\gamma < 1$ is the fraction of the work that is inherently sequential.
- Numerical kernels: $\mathcal{W}(p) = \mathcal{W}/p + \gamma\mathcal{W}^{2/3}/\sqrt{p}$. This is representative of a matrix product or a LU/QR factorization of size N on a 2D-processor grid, where $\mathcal{W} = O(N^3)$. In the algorithm in [3], $p = q^2$ and each processor receives $2q$ blocks of size N^2/q^2. Here γ is the communication-to-computation ratio of the platform.

We assume that the parallel job is tightly coupled, meaning that all p processors operate synchronously throughout the job execution. These processors execute the same amount of work $\mathcal{W}(p)$ in parallel, chunk by chunk. The total time (on one processor) to execute a chunk of size ω, and then checkpointing it, is $\omega + C(p)$. For the Makespan and NextFailure problems, we aim at computing a function f such that $f(\omega|\tau_1, \ldots, \tau_p)$ is the size of the next chunk that should be executed on every processor given a remaining amount of work $\omega \leq \mathcal{W}(p)$ and given a system state $(\tau_1, \ldots, \tau_p)$, where τ_i denotes the time elapsed since the last failure of the i-th processor. We assume that failure arrivals at all processors are *iid*.

An important remark on rejuvenation.

Two options are possible for recovering after a failure. Assume that a processor, say P_1, fails at time t. A first option found in the literature [6, 28] is to rejuvenate *all* processors together with P_1, from time t to $t + D$ (e.g., via rebooting in case of software failure). Then all processors are available at time $t + D$ at which point they start executing the recovery simultaneously. In the second option, *only* P_1 is rejuvenated and the other processors are kept idle from time t to $t + D$. With this option any processor other than P_1 may fail between t and $t + D$ and thus may be itself in the process of being rejuvenated at time $t + D$.

Let us consider a platform with p processors that experience *iid* failures according to a Weibull distribution with scale parameter λ and shape parameter k, i.e., with cumulative distribution $F(t) = 1 - e^{-\frac{t^k}{\lambda^k}}$, and mean $\mu = \lambda\Gamma(1 + \frac{1}{k})$. Define a *platform* failure as the occurrence of a failure at any of the processors. When rejuvenating all processors after each failure, platform failures are distributed according to a Weibull distribution with scale parameter $\frac{\lambda}{p^{1/k}}$ and shape parameter k. The MTBF for the platform is thus $D + \frac{\mu}{p^{1/k}}$ (note that the processor-level MTBF is $D + \mu$). When rejuvenating only the processor that failed, the platform MTBF is simply $\frac{D+\mu}{p}$. If $k = 1$, which corresponds to an Exponential distribution, rejuvenating all processors leads to a higher platform MTBF and is beneficial. However, if $k < 1$, rejuvenating all processors leads to a lower platform MTBF than rejuvenating only the processor that failed because $D \ll \frac{\mu}{p}$

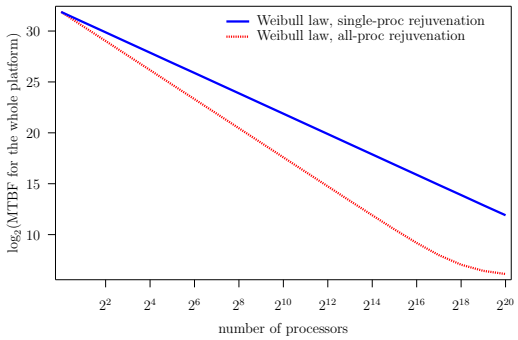

Figure 1: Impact of the two rejuvenation options on the platform MTBF for a Weibull distribution with shape parameter 0.70, a processor-level MTBF of 125 years, and a downtime of 60 seconds.

in practical settings. This is shown on an example in Figure 1, which plots the platform MTBF vs. the number of processors. This behavior is easily explained: for a Weibull distribution with shape parameter $k < 1$, the probability $\mathbb{P}(X > t + x|X > t)$ strictly increases with t. In other words, a processor is less likely to fail the longer it remains in a fault-free state. It turns out that failure inter-arrival times for real-life systems have been modeled well by Weibull distributions whose shape parameter are strictly lower than 1 (either 0.7 or 0.78 in [13], 0.50944 in [20], and between 0.33 and 0.49 in [26]). The overall conclusion is then that rejuvenating all processors after a failure, albeit commonly used in the literature, is likely not appropriate for large-scale platforms. Furthermore, even for Exponential distributions, rejuvenating all processors is not meaningful for hardware failures. Therefore, in the rest of this paper we assume that after a failure only the failed processor is rejuvenated[1].

3.2 Solving Makespan

In the case of the Exponential distribution, due to the memoryless property, the p processors used for a job can be conceptually aggregated into a virtual "macro-processor" with the following characteristics:

- Failure inter-arrival times follow an Exponential distribution of parameter $\lambda' = p\lambda$;
- The checkpoint and recovery overheads are $C(p)$ and $R(p)$, respectively.

A direct application of Theorem 1 yields the optimal solution of the Makespan problem for parallel jobs:

Proposition 5. *Let $\mathcal{W}(p)$ be the amount of work to execute on p processors whose failure inter-arrival times follow iid Exponential distributions with parameter λ. Let $K_0 = \frac{p\lambda\mathcal{W}(p)}{1+\mathbb{L}(-e^{-p\lambda C(p)-1})}$. Then the optimal strategy to minimize the expected makespan time is to split $\mathcal{W}(p)$ into $K^* = \max(1, \lfloor K_0 \rfloor)$ or $K^* = \lceil K_0 \rceil$ same-size chunks, whichever minimizes $\psi(K^*) = K^*(e^{p\lambda(\frac{\mathcal{W}(p)}{K^*}+C(p))} - 1)$.*

Interestingly, although we know the optimal solution with p processors, we are not able to compute the optimal expected makespan analytically. Indeed, $\mathbb{E}(T_{rec})$, for which we had a closed form in the case of sequential jobs, becomes quite intricate in the case of parallel jobs. This is because during the downtime of a given processor another processor may fail. During the downtime of that processor, yet another processor may fail, and so on. We would need to compute the expected duration of these "cascading" failures until all processors are simultaneously available.

For arbitrary distributions, i.e., distributions without the memoryless property, we cannot tractably extend the dynamic programming algorithm DPMakespan. This is because one would have to memorize the evolution of the time elapsed since the last failure for all possible failure scenarios for each processor, leading to a number of states exponential in p. Fortunately, the dynamic programming approach for solving NextFailure can be extended to the case of a parallel job, as seen in Section 3.3. This was our motivation for studying NextFailure in the first place, and in the case of non-exponential failures, we use the solution to NextFailure as a heuristic solution for Makespan.

3.3 Solving NextFailure

For solving NextFailure using dynamic programming, there is no need to keep for each processor the time elapsed since its last failure as parameter of the recursive calls. This is because the τ variables of all processors evolve identically: recursive calls only correspond to cases in which no failure has occurred. Formally, the goal is to find a function $f(\omega|\tau) = \omega_1$ that maximizes $\mathbb{E}(W(\omega|\tau_1, \ldots, \tau_p))$, where $\mathbb{E}(W(0|\tau_1, \ldots, \tau_p)) = 0$ and

$$\mathbb{E}(W(\omega|\tau_1, \ldots, \tau_p)) = \begin{cases} \omega_1 + \mathbb{E}(W(\omega - \omega_1|\tau_1 + \omega_1 + C(p), \ldots, \tau_p + \omega_1 + C(p)) \\ \quad \text{if no processor fails during the next } \omega_1 + C(p) \\ \quad \text{units of time} \\ 0 \quad \text{otherwise.} \end{cases}$$

Using a straightforward adaptation of DPNextFailure, which computes the probability of success

$$P_{suc}(x|\tau_1, \ldots, \tau_p) = \prod_{i=1}^{p} \mathbb{P}(X \geq x + \tau_i | X \geq \tau_i),$$

we obtain:

Proposition 6. *Using a time quantum u, for any failure inter-arrival time distribution,* DPNextFailure *computes an optimal solution to* NextFailure *with p processors in time $O(p\frac{\mathcal{W}}{u}^3)$.*

Even if a linear dependency in p, due to the computation of P_{suc}, seems a small price to pay, the above computational complexity is not tractable. Typical platforms in the scope of this paper (Jaguar [4], Exascale platforms) consist of tens of thousands of processors. The DPNextFailure algorithm is thus unusable, especially since it must be invoked after each failure. In what follows we propose a method to reduce its computational complexity.

Rather than working with the set of all p τ_i values, we approximate this set. With distributions such as the Weibull distribution, the smallest τ_i's have the highest impact on the overall probability of success. Therefore, we keep in the set the exact n_{exact} smallest τ_i values. Then we approximate the $p - n_{\text{exact}}$ remaining τ_i values using only n_{approx} "reference" values $\tau_1^{\approx}$, ..., $\tau_{n_{\text{approx}}}^{\approx}$. To each processor P_i whose τ_i value is not one of the n_{exact} smallest τ_i values, we associate one of the reference values. We can then simply keep track of how many processors are associated to each

[1]For the sake of completeness, we consider both rejuvenation options for Exponential failures in the companion research report [5]. We observe similar results for both options.

reference value, thereby vastly reducing computational complexity. We pick the reference values as follows. $\tau_1^{\approx}$ is the smallest of the remaining $p - n_{\text{exact}}$ exact τ_i values, while $\tau_{n_{\text{approx}}}^{\approx}$ is the largest. (Note that if processor P_i has never failed to date then $\tau_i = \tau_{n_{\text{approx}}}^{\approx}$.) The remaining $n_{\text{approx}} - 2$ reference values are chosen based on the distribution of the (*iid*) failure inter-arrival times. Assuming that X is a random variable distributed according to this distribution, then, for $i \in [2, n_{\text{approx}} - 1]$, we compute $\tau_i^{\approx}$ as

$$\tau_i^{\approx} = \text{quantile}\left(X, \frac{n_{\text{approx}} - i}{n_{\text{approx}} - 1}\,\mathbb{P}(X \geq \tau_1^{\approx}) + \frac{i-1}{n_{\text{approx}} - 1}\,\mathbb{P}(X \geq \tau_{n_{\text{approx}}}^{\approx})\right).$$

We have implemented DPNextFailure with $n_{\text{exact}} = 10$ and $n_{\text{approx}} = 100$. For the simulation scenario detailed in Section 5.2.2, we have studied the precision of this approximation by evaluating the relative error incurred when computing the probability using the approximated state rather than the exact one, for chunks of size 2^{-i} times the MTBF of the platform, with $i \in \{0..6\}$ and failure inter-arrival times following a Weibull distribution. It turns out that the larger the chunk size, the less accurate the approximation. Over the whole execution of a job in the settings of Section 5.2.2 (i.e., for 45,208 processors), the worst relative error is lower than 0.2% for a chunk of duration equal to the MTBF of the platform. In practice, the chunks used by DPNextFailure are far smaller, and the approximation of their probabilities of success is thus far more accurate.

The running time of DPNextFailure is proportional to the work size $\mathcal{W}$. If $\mathcal{W}$ is significantly larger than the platform MTBF, which is very likely in practice, then with high probability a failure occurs before the last chunks of the solution produced by DPNextFailure are even considered for execution. In other words, a significant portion of the solution produced by DPNextFailure is unused, and can thus be discarded without a significant impact on the quality of the end result. In order to further boost the execution time of DPNextFailure, rather than invoking it on the size of the remaining work ω, we invoke it for a work size equal to $\min(\omega, 2 \times MTBF/p)$, where $MTBF$ is the processor-level mean time between failures ($MTBF/p$ is thus the platform mean time between failures). We use only the first half of the chunks in the solution produced by DPNextFailure so as to avoid any side effects due the truncated ω.

With all these optimizations, DPNextFailure runs in a few seconds even for the largest platforms in our experiments. In all the application execution times reported in Sections 5 and 6 the execution time of DPNextFailure is taken into account.

4. SIMULATION FRAMEWORK

In this section we detail our simulation methodology. We use both synthetic and real-world failure distributions. The source code and all simulation results are publicly available at: `http://graal.ens-lyon.fr/~fvivien/checkpoint`.

4.1 Heuristics

Our simulator implements the following eight checkpointing policies (recall that $MTBF/p$ is the mean time between failures of the whole platform):

- Young is the periodic checkpointing policy with period $\sqrt{2 \times C(p) \times \frac{MTBF}{p}}$ given in [31].
- DalyLow is the first order approximation given in [10]. This is a periodic policy with period: $\sqrt{2 \times C(p) \times (\frac{MTBF}{p} + D + R(p))}$.
- DalyHigh is the periodic policy (high order approximation) given in [10].
- Bouguerra is the periodic policy given in [6].
- Liu is the non-periodic policy given in [20].
- OptExp is the periodic policy whose period is given in Proposition 5.
- DPNextFailure is the dynamic programming algorithm that maximizes the expectation of the amount of work completed before the next failure occurs.
- DPMakespan is the dynamic programming algorithm that minimizes the expected makespan. For parallel jobs, DPMakespan makes the false assumption that all processors are rejuvenated after each failure (without this assumption this heuristic cannot be used).

Our simulator also implements LowerBound, an omniscient algorithm that knows when the next failure will happen and checkpoints just in time, i.e., $C(p)$ time units before the failure. The makespan of LowerBound is thus an absolute lower bound on the makespan achievable by any policy, and is unattainable in practice. Along the same line, the simulator implements PeriodLB, which implements a numerical search for the optimal period by evaluating each candidate period on 1,000 randomly generated scenarios (which would have a prohibitive computational cost in practice). To build the candidate periods, the period computed by OptExp is multiplied and divided by $1 + 0.05 \times i$ with $i \in \{1, ..., 180\}$, and by 1.1^j with $j \in \{1, ..., 60\}$. PeriodLB corresponds to the periodic policy that uses the best period found by the search.

We point out that DalyLow, DalyHigh, and OptExp compute the checkpointing period based solely on the MTBF, implicitly assuming that failures are exponentially distributed. For the sake of completeness we nevertheless include them in all our simulations, simply using the MTBF value even when failures are not exponentially distributed.

Performance evaluation – We compare heuristics using average makespan degradation, defined as follows. Given an experimental scenario (i.e., parameter values for failure distribution and platform configuration), we generate a set $\{tr_1, \ldots, tr_{250}\}$ of 250 traces. For each trace tr_i and each heuristic $heur_j$, we compute the achieved makespan, $res_{(i,j)}$. The makespan degradation for heuristic $heur_j$ on trace tr_i is defined as $v_{(i,j)} = res_{(i,j)}/min_{j \neq 0}\{res_{(i,j)}\}$ (where $heur_0$ is LowerBound). We compute the average degradation for heuristic $heur_j$ as $\sum_{i=1}^{250} v_{(i,j)}/250$. Standard deviations are small and thus not plotted on figures (see the companion research report [5] for standard deviations values).

4.2 Platforms

We target three types of platforms: One-processor, Petascale, and Exascale. For Petascale we choose as reference the Jaguar supercomputer [4], which contains $p_{total} = 45,208$ processors. We consider jobs that use between 1,024 and 45,208 processors. We then corroborate the Petascale results by running simulations of Exascale platforms with $p_{total} = 2^{20}$ processors. For all three platform types, we determine the job size $\mathcal{W}$ so that a job using the whole platform would

	p_{total}	D	C,R	$MTBF$	$\mathcal{W}$
1-proc	1	60 s	600 s	1 h, 1 d, 1 w	20 d
Peta	45,208	60 s	600 s	125 y, 500 y	1,000 y
Exa	2^{20}	60 s	600 s	1250 y	10,000 y

Table 1: Parameters used in the simulations (C, R and D chosen according to [14, 8]). The first line corresponds to one-processor platforms, the second to Petascale platforms, and the third to Exascale platforms.

use it for a significant amount of time in the absence of failures, namely ≈ 20 days for the One-processor platform, ≈ 8 days for Petascale platforms, and ≈ 3.5 days for Exascale platforms. All relevant parameters are listed in Table 1.

4.3 Generation of failure scenarios

Synthetic failure distributions – To choose failure distribution parameters that are representative of realistic systems, we use failure statistics from the Jaguar platform. Jaguar is said to experience on the order of 1 failure per day [21, 2]. Assuming a 1-day platform MTBF gives us a processor MTBF equal to $\frac{p_{total}}{365} \approx 125$ years, where $p_{total} = 45,208$ is the number of processors of the Jaguar platform. To verify that our results are consistent over a range of processor MTBF values, we also consider a processor MTBF of 500 years. We then compute the parameters of Exponential and Weibull distributions so that they lead to this MTBF value (recall that $MTBF = \mu + D \approx \mu$, where μ is the mean of the underlying distribution). Namely, for the Exponential distribution we set $\lambda = \frac{1}{MTBF}$ and for the Weibull distribution, which requires two parameters k and λ, we set $\lambda = MTBF/\Gamma(1 + 1/k)$. We first fix $k = 0.7$ based on the results of [26], and then vary it between 0.15 and 1.0.

Log-based failure distributions – We also consider failure distributions based on failure logs from production clusters. We used logs for the largest clusters among the preprocessed logs in the *Failure trace archive* [17], i.e., for clusters at the Los Alamos National Laboratory [26]. In these logs, each failure is tagged by the node —and not just the processor— on which the failure occurred. Among the 26 possible clusters, we opted for the logs of the only two clusters with more than 1,000 nodes. The motivation is that we need a sample history sufficiently large to simulate platforms with more than ten thousand nodes. The two chosen logs are for clusters 18 and 19 in the archive (referred to as 7 and 8 in [26]). For each log, we record the set $\mathcal{S}$ of availability intervals. The discrete failure distribution for the simulation is generated as follows: the conditional probability $\mathbb{P}(X \geq t \mid X \geq \tau)$ that a node stays up for a duration t, knowing that it has been up for a duration τ, is set to the ratio of the number of availability durations in $\mathcal{S}$ greater than or equal to t, over the number of availability durations in $\mathcal{S}$ greater than or equal to τ.

Scenario generation – Given a p-processor job, a failure trace is a set of failure dates for each processor over a fixed time horizon h. In the one-processor case, h is set to 1 year. In all the other cases, h is set to 11 years and the job start time, t_0, is assumed to be one-year to avoid side-effects related to the synchronous initialization of all nodes/processors. Given the distribution of inter-arrival times at a processor, for each processor we generate a trace via independent sampling until the target time horizon is reached. Finally, for simulations where the only varying parameter is the number of processors $a \leq p \leq b$, we first generate traces for b processors. For experiments with p processors we then simply select the first p traces. This ensures that simulation results are coherent when varying p.

The two clusters used for computing our log-based failure distributions consist of 4-processor nodes. Hence, to simulate a 45,208-processor platform we generate 11,302 failure traces, one for each four-processor node.

5. SIMULATIONS WITH SYNTHETIC FAILURES

5.1 Single processor jobs

For a single processor, we cannot use a 125-year MTBF, as a job would have to run for centuries in order to need a few checkpoints. Hence we study scenarios with smaller values of the MTBF, from one hour to one week. This study, while unrealistic, allows us to compare the performance of DPNextFailure with that of DPMakespan.

5.1.1 Exponential failures

Table 2 shows the average makespan degradation for the eight heuristics and the two lower bounds, in the case of exponentially distributed failure inter-arrival times. Unsurprisingly, LowerBound is significantly better than all heuristics, especially for a small MTBF. It may seem surprising that PeriodLB achieves results close to but not equal to 1. This is because although the expected optimal solution is periodic, checkpointing with the optimal period is not always the best strategy for a given random scenario.

A first interesting observation is that the performance by the well-known Young, DalyLow, and DalyHigh heuristics is indeed close to optimal. While this result seems widely accepted, we are not aware of previously published simulation studies that have demonstrated it. Looking more closely at the results [5] we find that, in a large neighborhood of the optimal period, the performance of periodic policies is almost independent of the period. This explains that the Young, DalyLow, and DalyHigh heuristics have near optimal performance even if their periods differ.

In Section 2.4, we claimed that DPNextFailure should provide a reasonable solution to the Makespan problem. We observe that, at least in the one-processor case, DPNextFailure does lead to solutions that are close to those computed by DPMakespan and to the optimal.

5.1.2 Weibull failures

Table 3 shows results when failure inter-arrival times follow a Weibull distribution (note that the Liu heuristic was specifically designed to handle Weibull distributions). Unlike in the exponential case, the optimal checkpoint policy may be non-periodic [27]. Results in the table show that all the heuristics lead to results that are close to the optimal. The implication is that, in the one-processor case, one can safely use Young, DalyLow, and DalyHigh, which only require the failure MTBF, even for Weibull failures. In Section 5.2.2 we see that this result does not hold for multiprocessor platforms. Note that, just like in the Exponential case, DPNextFailure leads to solutions that are close to those computed by DPMakespan.

Heuristics	MTBF 1 hour	1 day	1 week
LowerBound	0.62865	0.90714	0.979151
PeriodLB	1.00705	1.01588	1.02298
Young	1.01635	1.01590	1.02332
DalyLow	1.02711	1.01611	1.02338
DalyHigh	1.00700	1.01592	1.02373
Liu	1.01607	1.01655	1.02333
Bouguerra	1.02562	1.02329	1.02685
OptExp	1.00705	1.01611	1.02298
DPNextFailure	1.00785	1.01699	1.02851
DPMakespan	1.00737	1.01655	1.03467

Table 2: Degradation from best for a single processor with Exponential failures.

Heuristics	MTBF 1 hour	1 day	1 week
LowerBound	0.66417	0.90714	0.97915
PeriodLB	1.00960	1.01588	1.02298
Young	1.00965	1.01590	1.02332
DalyLow	1.01155	1.01611	1.02338
DalyHigh	1.01785	1.01592	1.02373
Liu	1.00914	1.01655	1.02333
Bouguerra	1.02936	1.02329	1.02685
OptExp	1.01788	1.01611	1.02298
DPNextFailure	1.01408	1.01699	1.02851
DPMakespan	1.00731	1.01655	1.03467

Table 3: Degradation from best for a single processor with Weibull failures.

5.2 Parallel jobs

Section 3.1 defines 3×2 combinations of parallelism and checkpointing overhead models. For our experiments we have instantiated these models as follows: $\mathcal{W}(p)$ is equal to either $\frac{\mathcal{W}}{p}$, $\frac{\mathcal{W}}{p} + \gamma\mathcal{W}$ with $\gamma \in \{10^{-4}, 10^{-6}\}$, or $\frac{\mathcal{W}}{p} + \gamma\frac{\mathcal{W}^{2/3}}{\sqrt{p}}$ with $\gamma \in \{0.1, 1, 10\}$; and $C(p) = R(p) = 600$ seconds or $C(p) = R(p) = 600 \times p_{total}/p$ seconds. Due to lack of space, in this paper we only report results for the embarrassingly parallel applications ($\mathcal{W}(p) = \mathcal{W}/p$) with constant checkpoint overhead ($C(p) = R(p) = 600$ seconds). Results for all other cases lead to the same conclusions regarding the relative performance of the various checkpointing strategies. We refer the reader to [5], which contains the comprehensive set of results for all combinations of parallelism and checkpointing overhead models.

5.2.1 Exponential failures

Petascale platforms – Figure 2 shows results for Petascale platforms. The main observation is that, regardless of the number of processors p, the Young, DalyLow, DalyHigh and Liu heuristics compute an almost optimal solution (i.e., with degradation below 1.023) indistinguishable from that of OptExp and PeriodLB. By contrast, the degradation of Bouguerra is only slightly higher. We see that DPNextFailure behaves satisfactorily: its degradation is less than 4.8‰ worse than that of OptExp for $p \geq 2^{13}$, and less than 1.85% worse overall. We also observe that DPNextFailure always performs better than DPMakespan. This is likely due to the false assumption in DPMakespan that all processors are rejuvenated after each failure. The same conclusions are reached when the MTBF per processor is 500 years instead of 125 years (see [5]).

Exascale platforms – Results for Exascale platforms, shown in Figure 3, corroborates the results obtained for Petascale platforms.

5.2.2 Weibull failures

Petascale platforms – A key contribution of this paper is the comparison between DPNextFailure and all previously proposed heuristics for the Makespan problem on platforms whose failure inter-arrival times follow a Weibull distribution. Existing heuristics provide good solutions for sequential jobs (see Section 5.1). Figure 4 shows that this is no longer the case beyond $p = 1,024$ processors as demonstrated by growing gaps between heuristics and PeriodLB as p increases. For large platforms, only DPNextFailure is able to bridge this gap. For example with $45,208$ processors, Young, DalyLow, and DalyHigh are at least 4.3% worse than DPNextFailure, the latter being only 0.76% worse than PeriodLB. These poor results of previously proposed heuristics are partly due to the fact that the optimal solution is not periodic. For instance, throughout a complete execution with $45,208$ processors, DPNextFailure changes the size of inter-checkpoint intervals from $2,984$ seconds up to $6,108$ seconds. Bouguerra is supposed to handle Weibull failures but has poor performance because it relies on the assumption that all processors are rejuvenated after each failure. Liu, which is specifically designed to handle Weibull failures, also provides bad results for large platforms[2]. We conclude that our dynamic programming approach provides significant improvements over all previously proposed approaches for solving the Makespan problem in the case of large platforms. The same conclusions are reached when the MTBF per processor is 500 years instead of 125 years (see [5]).

Number of spare processors necessary – In our simulations, for a job running around 10.5 days on a 45,208 processor platform, when using DPNextFailure, on average, 38.0 failures occur during a job execution, with a maximum of 66 failures. This provides some guidance regarding the number of spare processors necessary so as not to experience any job interruption, in this case circa 1‰.

Impact of the shape parameter k – We report results from experiments in which we vary the shape parameter k of the Weibull distribution in a view to assessing the sensitivity of each heuristic to this parameter. Figure 5 shows average makespan degradation vs. k. We see that, with small values of k, the degradation is small for DPNextFailure (below 1.040 for $k \geq 0.15$) , while it is dramatically larger for all other heuristics. DPNextFailure achieves the best performance over all heuristics for the range of k values seen in practice as reported in the literature (between 0.33 and 0.78 [13, 20, 26]). Bouguerra leads to very poor solutions because it assumes that all processors are rejuvenated after

[2]On most figures the curve for Liu is incomplete. Liu computes the *dates* at which the application should be checkpointed. In several cases the interval between two consecutive dates is smaller than the checkpoint duration, C, which is nonsensical. In such cases we do not report any result for Liu and speculate that there may be an error in [20].

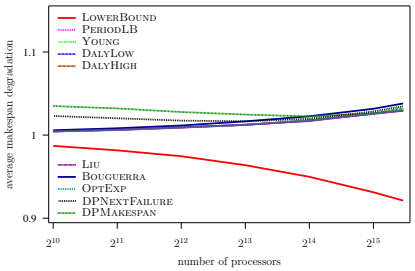

Figure 2: Evaluation of the different heuristics on a Petascale platform with Exponential failures.

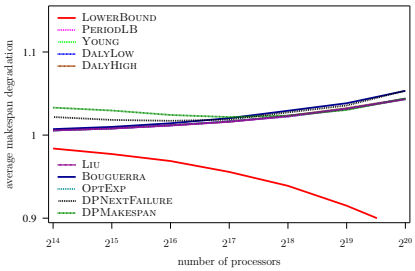

Figure 3: Evaluation of the different heuristics on an Exascale platform with Exponential failures.

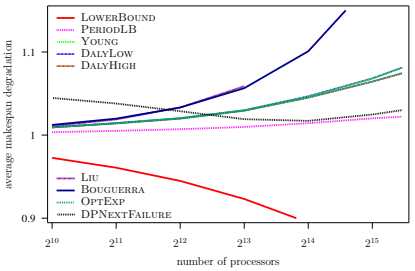

Figure 4: Evaluation of the different heuristics on a Petascale platform with Weibull failures.

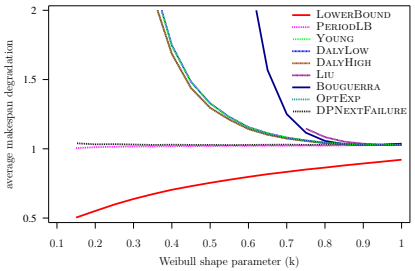

Figure 5: Varying the shape parameter k of the Weibull distribution for a Jaguar-like platform with $45,208$ processors.

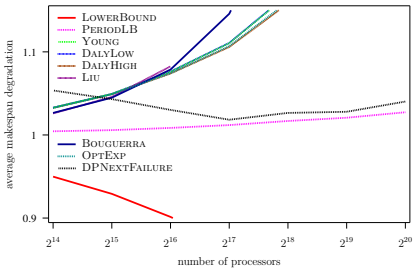

Figure 6: Evaluation of the different heuristics on an Exascale platform with Weibull failures.

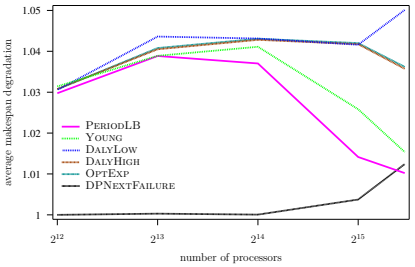

Figure 7: Evaluation of the different heuristics on a Petascale platform with failures based on the failure log of LANL cluster 19.

a failure, which is only acceptable for k close to 1 (i.e., for nearly Exponential failures) but becomes increasingly harmful as k becomes smaller.

Exascale platforms – Figure 6 presents results for Exascale platforms. The advantage of DPNextFailure over the other heuristics is even more pronounced than for Petascale platforms. The average degradation from best of DPNextFailure for platforms with between 2^{16} and 2^{20} processors is less than 1.028, the reference being defined by the inaccessible performance of PeriodLB.

6. SIMULATIONS WITH LOG-BASED FAILURES

To fully assess the performance of DPNextFailure, we also perform simulations using the failure logs of two production clusters, following the methodology explained in Section 4.3. We compare DPNextFailure to the Young, DalyLow, DalyHigh, and OptExp heuristics, adapting them by pretending that the underlying failure distribution is Exponential with the same MTBF at the empirical MTBF computed from the log. The same adaptation cannot be done for Liu, Bouguerra, and DPMakespan, which are thus not considered in this section.

Simulation results corresponding to one of the production cluster (LANL cluster 19, see Section 4.3) are shown in Figure 7. For the sake of readability, we do not display LowerBound as it leads to low values ranging from 0.80 to 0.56 as p increases (which underlines the intrinsic difficulty of the problem). As before, DalyHigh and OptExp achieve similar performance. But their performance, alongside that of DalyLow, is significantly worse than that of Young, especially for large p. The performance of all these heuristics is closer to the performance of PeriodLB than in the case of Weibull failures. The main difference with results for synthetic failures is that the performance of DPNextFailure is even better than that of PeriodLB. This is because, for these real-world failure distributions, periodic heuristics are inherently suboptimal. By contrast, DPNextFailure keeps adapting the size of the chunks that it attempts to execute. On a 45,208 processor platform, the processing time (or size) of the attempted chunks range from as (surprisingly) low as 60 seconds up to 2280 seconds. These values may seem extremely low, but the platform MTBF in this case is only 1,297 seconds (while R+C=1,200 seconds). This is thus a very difficult problem instance, but DPNextFailure solves it satisfactorily. More concretely, DPNextFailure saves more than 18,000 processor hours when using 45,208 processors, and more than 262,000 processor hours using 32,768 processors, compared to PeriodLB.

Simulation results based on the failure log of the other cluster (cluster 18, see Section 4.3) are similar, and even more in favor of DPNextFailure (see [5]).

7. RELATED WORK

In [10], Daly studies periodic checkpointing policies on platforms where failures inter-arrival times are exponentially distributed. That study accounts for checkpointing and recovery overheads (but not for downtimes), and allows failures to happen during recoveries. Two estimates of the optimal period are proposed. The lower order estimate is a generalization of Young's approximation [31], which takes recovery overheads into account. The higher order estimate is ill-formed as it relies on an equation that sums up non-

independent probabilities (Equation (13) in [10]). That work was later extended in [15], which studies the impact of sub-optimal periods on application performance.

In [6], Bouguerra et al. study the optimal checkpointing policy when failures can occur during checkpointing and recovery, with checkpointing and recovery overheads depending upon the application progress. They show that the optimal checkpointing policy is periodic when checkpointing and recovery overheads are constant, and when failure inter-arrival times follow either an Exponential or a Weibull distribution. They also give formulas to compute the optimal period in both cases. Their results, however, rely on the unstated assumption that all processors are rejuvenated after each failure and after each checkpoint. The dynamic programming approach in [28] suffers from the same issue.

In [29], the authors claim to use an "optimal checkpoint restart model [for] Weibull's and Exponential distributions" that they have designed in another paper (referenced as [1] in [29]). However, this latter paper is not available, and we were unable to compare our work to their solution. However, as explained in [29] the "optimal" solution in [1] is found using the assumption that checkpoint is periodic (even for Weibull failures). In addition, the authors of [29] partially address the question of the optimal number of processors for parallel jobs, presenting experiments for four MPI applications, using a non-optimal policy, and for up to 35 processors. Our approach is radically different since we target large-scale platforms with up to tens of thousands of processors and rely on generic application models for deriving optimal solutions.

In this work, we solve the NextFailure problem to obtain heuristic solutions to the Makespan problem in the case of parallel jobs. The NextFailure problem has been studied by many authors in the literature, often for single-processor jobs. Maximizing the expected work successfully completed before the first failure is equivalent to minimizing the expected wasted time before the first failure, which is itself a classical problem. Some authors propose analytical resolution using a "checkpointing frequency function", for both infinite (see [19, 20]) and finite time horizons (see [23]). However, these works use approximations, e.g., assuming that the expected failure occurrence is exactly halfway between two checkpointing events, which do not hold for general failure distributions. Approaches that do not rely on a checkpointing frequency function are used in [27, 18], but only for infinite time horizons. Finally, the dynamic programing approach proposed in [7] can handle variable checkpoint costs, but assumes that processors are rejuvenated after each failure and after each checkpoint.

8. CONCLUSION

We have studied the problem of scheduling checkpoints for minimizing the makespan of sequential and parallel jobs on large-scale and failure-prone platforms, which we have called Makespan. An auxiliary problem, NextFailure, was introduced as an approximation of Makespan. Both problems are defined rigorously in general settings. For exponential distributions, we have provided a complete analytical solution of Makespan together with an assessment of the quality of the NextFailure approximation. We have also designed dynamic programming solutions for both problems, that can be applied for any failure distribution.

We have obtained a number of key results via simulation experiments. For Exponential failures, our approach allows us to determine the optimal checkpointing policy. For Weibull failures, we have demonstrated the importance of using the "single processor rejuvenation" model. With this model, we have shown that our dynamic programming algorithm leads to significantly more efficient executions than all previously proposed algorithms with an average decrease in the application makespan of at least 4.16% for our largest simulated Petascale platforms, and of at least 23.9% for our largest simulated Exascale platforms. We have also considered failures from empirical failure distributions extracted from failure logs of two production clusters. In this settings, once again our dynamic programming algorithm leads to significantly more efficient executions than all previously proposed algorithms. Given that our results also hold across our various application and checkpoint scenarios, we claim that our dynamic programming approach provides a key step for the effective use of next-generation large-scale platforms. Furthermore, our dynamic programming approach can be easily extended to settings in which the checkpoint and restart costs are not constants but depends on the progress of the application execution.

There are several promising avenues for future work. Interesting questions relate to computing the optimal number of processors for executing a parallel job. On a fault-free machine, we have assumed that the execution time of the job decreases with the number of enrolled resources, and hence is minimal when the whole platform is used. In the presence of failures, this is no longer true (see the companion report [5] for examples), and the expected makespan may be smaller when using fewer than p_{total} processors. This leads to the idea of replicating the execution of a given job on say, both halves of the platform, i.e., with $p_{total}/2$ processors each. This could be done independently, or better, by synchronizing the execution after each checkpoint. The question of which is the optimal strategy is open. Another research direction comes from the fact that the (expected) makespan is not the only worthwhile or relevant objective. Because of the enormous energy cost incurred by large-scale platforms, along with environmental concerns, a crucial direction for future work is the design of checkpointing strategies that can trade off a longer execution time for a reduced energy consumption. Finally, it would be interesting to study how our approach could be adapted to non-coordinated checkpointing [12] and to multi-level checkpointing [22].

It is reasonable to expect that parallel jobs will be deployed successfully on exascale platforms only by using multiple techniques together (checkpointing, migration, replication, self-tolerant algorithms). While checkpointing is only part of the solution, it is an important part. This paper has shown the intrinsic difficulty of designing efficient checkpointing strategies, but it has also given promising results.

Acknowledgments. The authors would like to thank Slim Bouguerra, Franck Cappello, Bruno Gaujal, and Denis Trystram for many fruitful conversations.

9. REFERENCES

[1] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.

[2] L. Bautista Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka. Transparent low-overhead checkpoint for GPU-accelerated clusters. `https://wiki.ncsa.illinois.edu/download/attachments/17630761/INRIA-UIUC-WS4-lbautista.pdf?version=1&modificationDate=1290470402000`.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[4] A. Bland, R. Kendall, D. Kothe, J. Rogers, and G. Shipman. Jaguar: The World's Most Powerful Computer. In *GUC'2009*, 2009.

[5] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. Research Report 7520, INRIA, France, Jan. 2011. Available at `http://graal.ens-lyon.fr/~fvivien/`.

[6] M.-S. Bouguerra, T. Gautier, D. Trystram, and J.-M. Vincent. A flexible checkpoint/restart model in distributed systems. In *PPAM*, volume 6067 of *LNCS*, pages 206–215, 2010.

[7] M. S. Bouguerra, D. Trystram, and F. Wagner. An optimal algorithm for scheduling checkpoints with variable costs. Technical report, INRIA, Oct. 2010.

[8] F. Cappello, H. Casanova, and Y. Robert. Checkpointing vs. migration for post-petascale supercomputers. In *ICPP'2010*. IEEE Computer Society Press, 2010.

[9] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging. *IBM J. Res. Dev.*, 45(2):311–332, 2001.

[10] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2004.

[11] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero. The international exascale software project: a call to cooperative action by the global high-performance community. *Int. J. High Perform. Comput. Appl.*, 23(4):309–322, 2009.

[12] 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*, 34:375–408, 2002.

[13] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. *SIGMETRICS Perf. Eval. Rev.*, 30(1):217–227, 2002.

[14] J. Ho, C. Wang, and F. Lau. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[15] W. Jones, J. Daly, and N. DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *HPDC'10*, pages 276–279. ACM, 2010.

[16] N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS '95*, page 381, Washington, DC, USA, 1995. IEEE CS.

[17] 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*, 0:398–407, 2010.

[18] P. L'Ecuyer and J. Malenfant. Computing optimal checkpointing strategies for rollback and recovery systems. *IEEE Transactions on computers*, 37(4):491–496, 2002.

[19] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Transactions on computers*, pages 699–708, 2001.

[20] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *IPDPS 2008*, pages 1–9. IEEE, 2008.

[21] E. Meneses. Clustering Parallel Applications to Enhance Message Logging Protocols. `https://wiki.ncsa.illinois.edu/download/attachments/17630761/INRIA-UIUC-WS4-emenese.pdf?version=1&modificationDate=1290466786000`.

[22] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the ACM/IEEE SC Conference*, pages 1–11, 2010.

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

[24] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 2005.

[25] V. Sarkar and others. Exascale software study: Software challenges in extreme scale systems, 2009. White paper available at: `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`.

[26] 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.

[27] A. Tantawi and M. Ruschitzka. Performance analysis of checkpointing strategies. *ACM TOCS*, 2(2):123–144, 1984.

[28] S. Toueg and O. Babaoglu. On the optimum checkpoint selection problem. *SIAM J. Computing*, 13(3):630–649, 1984.

[29] K. Venkatesh. Analysis of Dependencies of Checkpoint Cost and Checkpoint Interval of Fault Tolerant MPI Applications. *Analysis*, 2(08):2690–2697, 2010.

[30] L. Wang, P. Karthik, Z. Kalbarczyk, R. Iyer, L. Votta, C. Vick, and A. Wood. Modeling Coordinated Checkpointing for Large-Scale Supercomputers. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 812–821, June 2005.

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