Multi-criteria Checkpointing Strategies: Response-Time versus Resource Utilization

Aurelien Bouteiller[1], Franck Cappello[2], Jack Dongarra[1], Amina Guermouche[3], Thomas Hérault[1], and Yves Robert[1,4]

[1] University of Tennessee Knoxville, USA
[2] University of Illinois at Urbana Champaign, USA & INRIA, France
[3] Univ. Versailles St Quentin, France
[4] Ecole Normale Supérieure de Lyon, France
{bouteill,dongarra,herault,yrobert1}@eecs.utk.edu,
cappello@illinois.edu, amina.guermouche@uvsq.fr

Abstract Failures are increasingly threatening the efficiency of HPC systems, and current projections of Exascale platforms indicate that rollback recovery, the most convenient method for providing fault tolerance to general-purpose applications, reaches its own limits at such scales. One of the reasons explaining this unnerving situation comes from the focus that has been given to per-application completion time, rather than to platform efficiency. In this paper, we discuss the case of uncoordinated rollback recovery where the idle time spent waiting recovering processors is used to progress a different, independent application from the system batch queue. We then propose an extended model of uncoordinated checkpointing that can discriminate between idle time and wasted computation. We instantiate this model in a simulator to demonstrate that, with this strategy, uncoordinated checkpointing per application completion time is unchanged, while it delivers near-perfect platform efficiency.

1 Introduction

The progress of many fields of research, in chemistry, biology, medicine, aerospace and general engineering, is heavily dependent on the availability of ever increasing computational capabilities. The High Performance Computing (HPC) community strives to fulfill these expectations, and for several decades, has embraced parallel systems to increase computational capabilities. Although there is no alternative technology in sight, the core logic of delivering more performance through ever larger systems bears its own issues, and most notably declining reliability. In the projections issued by the International Exascale Software Project (IESP) [1], even if individual components are expected to enjoy significant improvements in reliability, their number alone will drive the system Mean Time Between Failures (MTBF) to plummet, entering a regime where failures are not uncommon events, but a normal part of applications execution [2].

Coordinated rollback recovery, based on periodic, complete application checkpoint, and complete restart upon failure, has been the most successful and usually deployed failure mitigation strategy. Unfortunately, it appears that coordinated

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 420–431, 2013.

checkpoint/restart will suffer from unacceptable I/O overhead at the scale envisioned for future systems, leading to poor overall efficiency barely competing with replication [3]. In recent years, an alternative automatic rollback recovery technique, namely uncoordinated checkpointing with message logging [4], has received a lot of attention [5,6]. The key idea of this approach is to avoid the rollback of processes that have not been struck by failures, thereby reducing the amount of lost computation that has to be re-executed, and possibly permitting overlap between recovery and regular application progress. Unfortunately, under the reasonable hypothesis of tightly coupled applications (the most common type, whose complexity often compels automatic fault tolerance), processes that do not undergo rollback have to wait for restarted processes to catch up before they can resume their own progression, thereby spending as much time idling than they would have spent re-executing work in a coordinated approach.

In this paper, we propose to consider the realistic case of an HPC system with a queue of independent parallel jobs (from a single workflow, or even submitted by different users). Instead of solely focusing on per-application completion time, which is strongly challenged by numerous failures, the goal of such a system is to complete as many useful computations as possible (while still retaining reasonable per-application completion time). The proposed application deployment scheme addressed in this paper makes use of automatic, uncoordinated checkpoint/restart. It overlaps idling time suffered by recovering applications, by progress made on another application. This second application is loaded on available resources, meanwhile uncoordinated rollback recovery is taking place on the limited subset of the resources that needs to re-execute work after a failure. Based on this strategy, we extend the model proposed in [7] to make a distinction between wasted computation and processor idle time. The waste incurred by the individual application, and the total waste of the platform, are both expressed with the model, and we investigate the trade-offs between optimizing for application efficiency or for platform efficiency.

The rest of this paper is organized as follows: Section 2 gives an informal statement of the problem. The strategy that targets platform efficiency is described in Section 3. Section 4 presents the model and the scenarios used to analyze the behavior of the application-centric and platform-centric scenarios. Section 5 is devoted to a comprehensive set of simulations for relevant platform and application case studies. Section 6 provides an overview of related work. Finally we give some concluding remarks and hints for future work in Section 7.

2 Background and Problem Statement

Many approaches have been proposed to resolve the formidable threat that process failures pose to both productivity and efficiency of HPC applications. Algorithm Based Fault Tolerance [8], or Naturally Fault Tolerant Methods [9] promise to deliver exceptional performance despite failures, by tailoring recovery actions for each particular application. However, the use of intrinsic algorithmic properties is an application-specific solution that often entails excruciating software engineering efforts, which makes it difficult to apply to production codes.

In a sharp contrast, checkpoint/restart rollback recovery strategies can be provided automatically, without modifications to the supported application. Although the classical coordinated checkpoint approach seems to be reaching its limits [3], recent optimizations and experimental studies outline that compelling performance can be obtained from uncoordinated checkpointing [5,6]. Rollback recovery protocols employ checkpoints to periodically save the state of a parallel application, so that when a failure strikes some process, the application can be restored into one of its former states. In a parallel application, the recovery line is the state of the entire application after some processes have been restarted from a checkpoint. Unfortunately, not all recovery lines are consistent (*i.e.* result in a correct execution); in particular, recovery lines that separate the emission and matching reception event of a message are problematic. The two main families of rollback recovery protocols differ mostly in the way they handle these messages crossing the recovery line [4]. In the coordinated checkpoint approach, a collection of checkpoints is constructed so that consistency threatening messages do not exist between checkpoints of the collection (using a coordination algorithm). Since the checkpoint collection forms the only recovery line that is guaranteed to be correct, all processes have to roll back simultaneously, even if they are not faulty. As a result, the bulk amount of lost work is increased, and the strategy is not optimal for a given number of failures. The non-coordinated checkpoint approach avoids duplicating the work completed by non-faulty processes. Checkpoints are taken independently, and only failed processes endure rollback. Obviously, the resulting recovery line is not guaranteed to be correct without the addition of supplementary state elements to resolve the issues posed by crossing messages. Typically, message logging and event logging [4] store the necessary state elements during the execution of the application. When a process has to roll back to a checkpoint, it undergoes a managed, isolated re-execution, where all non-deterministic event outcomes are forced according to the event log, and messages from the past are served from the message log without rollback of original senders.

Problem Statement: For typical HPC applications, which are often tightly coupled, the ability of restarting only faulty processes (hence limiting duplicate computation to a minimum) does not translate into great improvements of the application completion time. This is illustrated in the instantiations of the model that we recently proposed [7], which captures the intricacies of advanced uncoordinated recovery techniques. Despite being spared the overhead of executing duplicate work, surviving processes quickly reach a synchronization point where further progress depends on input from rollback processes. Since the recovered processes have a significant amount of duplicate work to re-execute before they can catch up with the general progress of the application, surviving processes spend a significant amount of time idling; altogether, the overall application completion time is only marginally improved. Yet, it is clear that, given the availability of several independent jobs, idle time can be used to perform other useful computations, thereby diminishing the wasted time experienced by the platform as a whole.

3 Strategy to Improve Platform Efficiency

In this paper, we propose a scheduling strategy that complements uncoordinated rollback recovery, in order to decrease the waste of computing resources during recovery periods. When a failure occurs (represented as a lightning bolt in Figure 1), a set of spare processes is used to execute the duplicate work of processes that have to roll back to a checkpoint ($R + ReExec$). However, unlike regular uncoordinated checkpoint, instead of remaining active and idling, the remainder of the application is stopped, and flushed from memory to disk. The resulting free resources are used to progress an independent application *App*2. When the recovering processes have completed sufficient duplicate work, the supplementary application can be stopped (and its progress saved with a checkpoint); the initial application can then be reloaded and its execution resumes normally. In the next section, we propose an analytical model for this strategy, that permits to compute the supplementary execution time for the initial application, together with the total waste of computing resources endured by the platform. This model extends on previous work [7], which considered only the impact on application efficiency, and therefore let one of the key advantages of uncoordinated recovery unaccounted for, in the (reasonable) hypothesis of tightly coupled applications. We then use the model to investigate the appropriate checkpoint period, and to predict adequate strategies that deliver low platform waste while preserving application completion time.

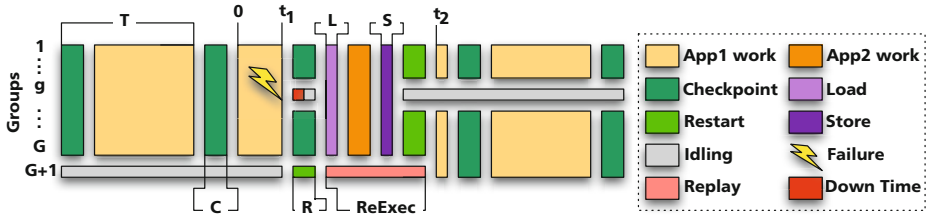

Fig. 1. The S_{Plat} scenario: a deployment strategy that improves the efficiency of resource usage in the presence of failures

4 Model

In this section, we introduce the model used to investigate the performance behavior of the proposed deployment scheme. We detail two execution scenarios: a regular uncoordinated checkpoint deployment that uses the whole platform for a single application; and the aforementioned approach that sacrifices a group of processors as a spare dedicated to recovery of failed processors, but can use the remainder of the platform to progress another application during a recovery. The goal is to compare the *waste* – the fraction of time where resources are not used to perform useful work. The minimum waste will be achieved for some optimal checkpointing period, which will likely differ for each scenario.

Table 1. Key model parameters

μ_p	Platform MTBF
G or $G+1$	Number of groups
T	Length of period
W	Work done every period
C	Checkpoint time
D	Downtime
R	Restart (from checkpoint) time
α	Slow-down execution factor when checkpointing
λ	Slow-down execution factor due to message logging
β	Increase rate of checkpoint size per work unit

Model Parameters and Notations. The model parameters and notations are summarized in Table 1.

- The system employs a hierarchical rollback recovery protocol with message-logging for protection against failures. The platform is therefore partitioned into $G+1$ processor[1] groupsthat can recover independently. In the S_{Plat} scenario, one of these groups is used as a spare, while all $G+1$ participate to the execution in the S_{App} scenario. We let $\text{WASTE}_{app}(T)$ denote the waste induced by the S_{App} scenario with a checkpointing period of T, and T_{app}^{opt} the value of T that minimizes it. Similarly, We define $\text{WASTE}_{plat}(T)$, T_{plat}^{opt} for the S_{Plat} scenario.
- Checkpoints are taken periodically, every T seconds. Hence, every period of length T, we perform some useful work W and take a checkpoint of duration C. Without loss of generality, we express W and T with the same unit: a unit of work executed at full speed takes one second. However, there are two factors that slow-down execution:
  - During checkpointing, which lasts C seconds, we account for a slow-down due to I/O operations, and only αC units of work are executed, where $0 \leq \alpha \leq 1$. The case $\alpha = 0$ corresponds to a fully blocking checkpoint, while $\alpha = 1$ corresponds to a fully overlapped checkpoint, and all intermediate situations can be represented;
  - Throughout the period, we account for a slow-down factor λ due to the communication overhead induced by message logging. A typical value is $\lambda = 0.98$ [5,6];
  - Altogether, the amount of work W that is executed every period of length T is

$$W = \lambda((T - C) + \alpha C) = \lambda(T - (1 - \alpha)C) \tag{1}$$

- We use D for the downtime and R for the time to restart from a checkpoint, after a failure has struck. We assume that $D \leq C$ to avoid clumsy expressions, and because it is always the case in practice. However, one can easily extend the analysis to the case where $D > C$.
- Message logging has both a positive and a negative impact on performance:

[1] Our approach is agnostic of the granularity of the processor, which can be either a single CPU, or a multi-core processor, or any relevant computing entity.

- During the recovery, inter-group messages entail no communication as they are available from the message log, in local memory. This results in a speed-up of the re-execution (up to twice as fast for some applications [10]), which is captured in the model by the ρ factor.
- Every inter-group message that has been logged since the last checkpoint must be included in the current checkpoint. Consequently, the size of the checkpoint increases with the amount of work per unit. To account for this increase, we write the equation

$$C = C_0(1 + \beta W) \tag{2}$$

The parameter C_0 is the time needed to write this application footprint onto stable storage, without message-logging. The parameter β quantifies the increase in the checkpoint time resulting from the increase of the log size per work unit (which is itself strongly tied to the communication to computation ratio of the application).

– Combining Equations (1) and (2), we derive the final value of the checkpoint time

$$C = \frac{C_0(1 + \beta\lambda T)}{1 + C_0\beta\lambda(1 - \alpha)} \tag{3}$$

We point out that the same application is deployed on G groups instead of $G + 1$ in the S_{Plat} scenario. As a consequence, when processor local storage is available, C_0 is increased by $\frac{G+1}{G}$ in S_{Plat}, compared to the S_{App} case.

Computing the Waste. The major objective of this paper is to compare the minimum waste resulting from each scenario. Intuitively, the period T_{app}^{opt} (single application) will be smaller than the period T_{plat}^{opt} (platform-oriented) because the loss due to a failure is higher in the former scenario. In the latter scenario, we lose a constant amount of time (due to switching applications) instead of losing an average of half the checkpointing period in the first scenario. We then aim at comparing the four values $\text{WASTE}_{app}(T_{app}^{opt})$, $\text{WASTE}_{app}(T_{plat}^{opt})$, $\text{WASTE}_{plat}(T_{plat}^{opt})$, and $\text{WASTE}_{plat}(T_{app}^{opt})$, the later two values characterizing the trade-off when using the optimal period of a scenario for the other one.

Let T_{base} be the parallel execution time without any overhead (no checkpoint, failure-free execution). The first source of overhead comes the rollback-and-recovery protocol. Every period of length T, we perform some useful work W (whose value is given by Equation (1)) and take a checkpoint. Checkpointing induces an overhead, even if there is no failure, because not all the time is spent computing: the fraction of *useful* time is $\frac{W}{T} \leq 1$. The failure-free execution time T_{ff} is thus given by the equation $\frac{W}{T}T_{ff} = T_{base}$, which we rewrite as

$$(1 - \text{WASTE}_{ff})T_{ff} = T_{base}, \text{ where } \text{WASTE}_{ff} = \frac{T - W}{T} \tag{4}$$

Here WASTE_{ff} denotes the waste due to checkpointing and message logging in a failure-free environment. Now, we compute the overhead due to failures.

Failures strike every μ_p units of time in average, and for each of them, we lose an amount of time t_{lost}. The final execution time T_{final} is thus given by the equation $(1 - \frac{\mathrm{t}_{lost}}{\mu_p})T_{final} = T_{ff}$ which we rewrite as

$$(1 - \textsc{Waste}_{fail})T_{final} = T_{ff}, \text{ where } \textsc{Waste}_{fail} = \frac{\mathrm{t}_{lost}}{\mu_p} \tag{5}$$

Here $\textsc{Waste}_{fail}$ denotes the waste due to failures. Combining Equations (4) and (5), we derive that

$$(1 - \textsc{Waste}_{final})T_{final} = T_{base} \tag{6}$$

$$\textsc{Waste}_{final} = \textsc{Waste}_{ff} + \textsc{Waste}_{fail} - \textsc{Waste}_{ff}\textsc{Waste}_{fail} \tag{7}$$

Here $\textsc{Waste}_{final}$ denotes the total waste during the execution, which we aim at minimizing by finding the optimal value of the checkpointing period T. In the following, we compute the values of $\textsc{Waste}_{final}$ for each scenario. The analysis restricts to (at most) a single failure per checkpointing period. Simulation results in Section 5 discuss the impact of this hypothesis (which, to that best of our knowledge, is mandatory for a tractable mathematical analysis).

Scenario S_{App}. We have $\textsc{Waste}_{ff} = \frac{T-W}{T} = \frac{T-\lambda(T-(1-\alpha)C)}{T}$ for both scenarios but recall that we enroll $G+1$ groups in scenario S_{App} and only G groups in in scenario S_{Plat}, so that the value of C is not the same in Equation (3). Next, we compute the value of $\textsc{Waste}_{fail}$ for the S_{App} scenario.

$$\textsc{Waste}_{fail} = \frac{1}{\mu_p}\left[D + R + \frac{T-C}{T} \times \mathrm{ReExec}_1 + \frac{C}{T} \times \mathrm{ReExec}_2\right] \tag{8}$$

$$\text{where} \qquad \mathrm{ReExec}_1 = \frac{1}{\rho}\left(\alpha C + \frac{T-C}{2}\right), \qquad \mathrm{ReExec}_2 = \frac{1}{\rho}\left(\alpha C + T - C + \frac{C}{2}\right)$$

First, $D + R$ is the duration of the downtime and restart. Then we add the time needed to re-execute the work that had already completed during the period, and that has been lost due to the failure. The time spent re-executing lost work is split into two terms, depending whether the failure strikes when a checkpoint is taking place or not. ReExec_1 is the term when no checkpoint was taking place at the time of the failure; it is therefore weighted by $(T-C)/T$, the probability of the failure striking within such a $T-C$ timeframe. ReExec_1 first includes the re-execution of the work done in parallel with the last checkpoint (of initial duration C), but no checkpoint activity happens during re-execution, so it takes only αC time units. Then we re-execute the work done in the work-only area. On average, the failure happens in the middle of the interval of length $T-C$, hence the time lost has an expected value of $\frac{T-C}{2}$. We finally account for the communication speedup during re-execution by introducing the ρ factor. We derive the value of ReExec_2 with a similar reasoning, and weight it by the probability C/T of the failure striking within a checkpoint interval. After simplification, we derive

$$\textsc{Waste}_{fail} = \frac{1}{\mu_p}\left(D + R + \frac{1}{\rho}\left(\frac{T}{2} + \alpha C\right)\right) \tag{9}$$

Scenario S_{Plat}. In this scenario, the first G groups are computing for the current application and are called *regular* groups. The last group is the *spare* group. As already pointed out, this leads to modifying the value of C, and hence the value of WASTE_{ff}. In addition, we also have to modify the value of T_{base}, which becomes $\frac{G+1}{G}T_{base}$, to account for the fact that it takes more time to produce the same work with fewer processors. We need to recompute WASTE_{final} accordingly so that Equation (6) still holds and we derive:

$$(1 - \text{WASTE}_{final})T_{final} = \frac{G+1}{G}T_{base} \tag{10}$$

$$\text{WASTE}_{final} = \frac{1}{G+1} + \frac{G}{G+1}\left(\text{WASTE}_{ff} + \text{WASTE}_{fail} - \text{WASTE}_{ff}\text{WASTE}_{fail}\right) \tag{11}$$

We now proceed to the computation of WASTE_{fail}, which is intricate. See Figure 1 for an illustration:

- Assume that a fault occurs within group g. Let t_1 be the time elapsed since the completion of the last checkpoint. At that point, the amount of work that is lost and should be re-executed is $W_1 = \alpha C + t_1$. Then:
  1. The faulty group (number g) is down during D seconds;
  2. The spare group (number $G+1$) takes over for the faulty group and does the recovery from the previous checkpoint at time t_1. It starts re-executing the work until time $t_2 = t_1 + R + \text{ReExec}$, when it has reached the point of execution where the fault took place. Here ReExec denotes the time needed to re-execute the work, and we have $\text{ReExec} = \frac{W_1}{\rho}$;
  3. The remaining $G-1$ groups checkpoint their current state while the faulty group g takes its downtime (recall that $D \leq C$);
  4. At time $t_1 + C$, the now free G groups load another application from its last checkpoint, which takes L seconds, perform some computations for this second application, and store their state to stable storage, which takes S seconds. The amount of work for the second application is computed so that the store operation completes exactly at time $t_2 - R$. Note that it is possible to perform useful work for the second application only if $t_2 - t_1 = R + \text{ReExec} \geq C + L + S + R$. Note that we did not assume that $L = C$, nor that $S = R$, because the amount of data written and read to stable storage may well vary from one application to another;
  5. At time $t_2 - R$, the G groups excluding the faulty group start the recovery for the first application, and at time t_2 they are ready to resume the execution of this first application together with the spare group: there remains $W - W_1$ units of work to execute to finish up the period. From time t_2 on, the faulty group becomes the spare group.

To simplify notations, let $X = C + L + S + R$ and $Y = X - R$. We rewrite the condition $t_2 - t_1 = R + \text{ReExec} \geq X$ as $\text{ReExec} \geq Y$, i.e., $\frac{\alpha C + t_1}{\rho} \geq Y$. This is equivalent to $t_1 \geq Z$, where $Z = \rho Y - \alpha C$. So if $t_1 \geq Z$, the first G groups lose X seconds, and otherwise they lose $R + \text{ReExec}$ seconds. Since t_1 is uniformly

distributed over the period T, the first case happens with probability $\frac{T-Z}{T}$ and the second case with probability $\frac{Z}{T}$. As for the second case, the expectation of t_1 conditioned to $t_1 \leq Z$ is $E[t_1|t_1 \leq Z] = \frac{Z}{2}$, hence the expectation of the time lost is $E[R + \text{ReExec}|t_1 \leq Z] = R + \frac{Y}{2} + \frac{\alpha C}{2\rho}$. Altogether the formula for WASTE_{fail} is:

$$\text{WASTE}_{fail} = \frac{1}{\mu_p}\left(\frac{T-Z}{T} \times X + \frac{Z}{T} \times (R + \frac{Y}{2} + \frac{\alpha C}{2\rho})\right) \tag{12}$$

- if the failure strikes during the first Z units of the period, which happens with probability $\frac{Z}{T}$, there is not enough time to load the second application, and the regular groups all waste $E[R + \text{ReExec}|t_1 \leq Z]$ seconds in average
- if the failure strikes during the last $T - Z$ units of the period, which happens with probability $\frac{T-Z}{T}$, then the regular groups all waste X units of time, and they perform some useful computation for the second application in the remaining time that they have before the spare group catches up.

5 Results

We instantiated the proposed waste model with different scenarios. Due to lack of space, we present here only two representative scenarios that illustrate the proposed approach. Parameters are set in accordance to target system specifications, and using experimentally observed values for message logging cost. Details for all parameters, as well as supplementary scenarios, consistent with the examples presented here, are available in the companion technical report [11]. The first scenario shown in Fig. 2, instantiates the model with features of the K-Computer ([12]); the checkpoint growth rate (β) is set according to a matrix-matrix multiply operation. The results present the waste from the platform perspective (green lines) and from the application perspective (red lines). The optimal checkpoint

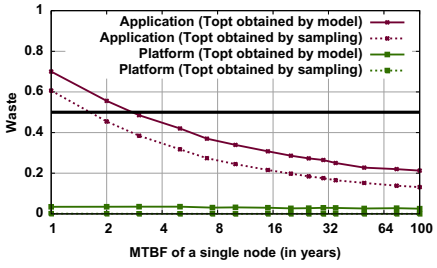

Fig. 2. Waste as function of the compute node MTBF, considering a matrix multiplication, on the K-Computer model

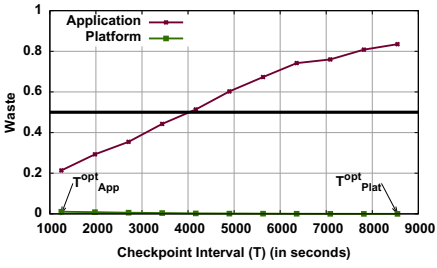

Fig. 3. Waste as function of the checkpoint period, considering a 2D-stencil operation, on the Fat Exascale Computer model (20 years individual processor MTBF)

period is computed by minimizing the model-computed waste. Because it is impossible to account for simultaneous failures in a closed-form formula, we then simulate an execution according to the model, except that simultaneous failures are possible; we verify that the model prediction is accurate nonetheless by comparing it with the best performing period obtained by sampling input periods in the simulator (dashed lines). Comparing the waste obtained when using the model-predicted optimal checkpoint period versus the brute force obtained period in the simulator reveals that the model slightly overestimate the waste, due to optimizing for the case where no simultaneous failures happen. However, general trends are respected, and the difference is under 7%.

When comparing the waste incurred on the application versus the waste of platform resources, this figure demonstrates the huge benefit of introducing a spare node (and loading a second application) on platform efficiency. Indeed, while the application waste, due to I/O congestion at checkpoint time, starts from a relatively high level when the component MTBF is very low (and thus when the machine usability is low), the platform waste itself is almost negligible.

Figure 3 fixes the MTBF of a single component to 20 years, and study the impact of choosing the optimal checkpoint interval so as to target either platform efficiency, or application efficiency. To do so, we varied the checkpoint period between the Application optimal value, and the Platform optimal value, as given by the model. To illustrate the diversity of experiments we conducted, the modeled system is one of the envisioned machines for Exascale systems [1] (the "Fat" version, featuring heavy multicore nodes), and the modeled application is a 2D-stencil application that fills up the system memory. Platform-optimal checkpoint periods are much longer than application-optimal checkpoint periods on the same machine, and both experiments exhibit a waste that increases when using a checkpoint period far away from their optimal. However, because the spare node is so much more beneficial to the general efficiency of the platform than to the efficiency of the application, it is extremely beneficial to select the optimal application checkpoint interval: the performance of the platform remains close to an efficiency of 1, while the waste of the application can be reduced significantly.

6 Related Work

An alternative approach to accelerate uncoordinated rollback recovery, in the Charm++ language, is to split and rebalance the re-execution [13]. For production MPI codes, written in Fortran/C, accounting for the different data and computation distribution during recovery periods entails an in depth rewrite (which may be partially automated by compilation techniques [14]). Even when such splitting is practical, the recovery workload is a small section of the application that is stretched on all resources, which, in accordance with Gustafson law [15], typically results in lower parallel efficiency at scale.

Overlapping downtime of programs blocked on I/O or memory accesses is achieved by a wide range of hardware and software techniques that improve

computational throughput (Hyper-threads [16], massive oversubscription in task based systems [17], etc.) Interestingly, co-scheduling [18] can leverage checkpoint-restart to improve communication/computation overlap. However, these have seldom been considered to overcome the cost of rollback recovery itself. Furthermore, checkpoint-restart modeling tools to assess the effectiveness of compensation techniques have not been available yet; the work proposed here supersedes previous models [19,7] in characterizing the difference in terms of platform efficiency when multiple, independent applications must be completed.

7 Conclusion

We have proposed a deployment strategy that permits to overlap the idle time created by recovery periods in uncoordinated rollback recovery with useful work from another application. This opportunity is unique to uncoordinated rollback recovery, since coordinated checkpointing requires the rollback of all processors, hence generates a similar re-execution time, but without idle time. We designed an accurate analytical model that captures the waste resulting from failures and protection actions, both in terms of application runtime and resource usage. The model results are compatible with experimentally observed behavior, and simplifications to express the model as a closed formula introduce only a minimal imprecision, that we have quantified through simulations.

The model has been used to investigate the effective benefit of the uncoordinated checkpointing strategy to improve platform efficiency, even in the most stringent assumptions of tightly coupled applications. Indeed, the efficiency of the platform can be greatly improved, even when using the checkpointing period that is the most amenable to minimizing application runtime. Finally, although replication (with a top efficiency of 50%) sometime delivers better per-application efficiency, we point out that a hierarchical checkpointing technique with dedicated spare nodes, as the one proposed in this paper, is the only approach that can provide a global platform waste close to zero.

For future works, we notice that the spare group is left unused outside of recovery period. Since the next activity on this group is a restart, it could be employed to aggressively prefetch checkpoints. It could also execute compute intensive yet accurate failure prediction models, to adapt checkpoint frequency and prefetching according to sensed runtime hardware conditions.

Acknowledgments. Y. Robert is with the Institut Universitaire de France. This work was supported in part by NSF, DOE, and the ANR Rescue project.

References

1. Dongarra, J., Beckman, P., Aerts, P., Cappello, F., Lippert, T., Matsuoka, S., Messina, P., Moore, T., Stevens, R., Trefethen, A., Valero, M.: The international exascale software project: a call to cooperative action by the global high-performance community. IJHPCA 23(4), 309–322 (2009)

2. Gibson, G.: Failure tolerance in petascale computers. Journal of Physics: Conference Series 78, 012022 (2007)
3. Ferreira, K., Stearley, J., Laros, J.H.I., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the Viability of Process Replication Reliability for Exascale Systems. In: Proc. of SC 2011. ACM/IEEE (2011)
4. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Survey 34, 375–408 (2002)
5. Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.J.: Correlated set coordination in fault tolerant message logging protocols. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 51–64. Springer, Heidelberg (2011)
6. Guermouche, A., Ropars, T., Snir, M., Cappello, F.: HydEE: Failure containment without event logging for large scale send-deterministic MPI applications. In: Proc. 26th IPDPS, pp. 1216–1227. IEEE (May 2012)
7. Bosilca, G., Bouteiller, A., Brunet, E., Cappello, F., Dongarra, J., Guermouche, A., Herault, T., Robert, Y., Vivien, F., Zaidouni, D.: Unified model for assessing checkpointing protocols at extreme-scale. Research report RR-7950, INRIA (2012)
8. Huang, K., Abraham, J.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers 100(6), 518–528 (1984)
9. Chen, Z., Fagg, G.E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., Dongarra, J.: Fault tolerant high performance computing by a coding approach. In: Proc. 10th ACM SIGPLAN PPoPP, pp. 213–223. ACM (2005)
10. Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., Cappello, F.: MPICH-V: a multiprotocol fault tolerant MPI. IJHPCA 20(3), 319–333 (2006)
11. Bouteiller, A., Cappello, F., Dongarra, J., Guermouche, A., Herault, T., Robert, Y.: Multi-criteria checkpointing strategies: Optimizing response-time versus resource utilization. Research report ICL-UT-1301, University of Tennessee (February 2013)
12. Miyazaki, H., Kusano, Y., Okano, H., Nakada, T., Seki, K., Shimizu, T., Shinjo, N., Shoji, F., Uno, A., Kurokawa, M.: K computer: 8.162 petaflops massively parallel scalar supercomputer built with over 548k cores. In: ISSCC, pp. 192–194. IEEE (2012)
13. Chakravorty, S., Kale, L.: A fault tolerance protocol with fast fault recovery. In: Proc. 21st IPDPS, pp. 1–10. IEEE (March 2007)
14. Yang, X., Du, Y., Wang, P., Fu, H., Jia, J.: FTPA: Supporting fault-tolerant parallel computing through parallel recomputing. IEEE Transactions on Parallel and Distributed Systems 20(10), 1471–1486 (2009)
15. Gustafson, J.L.: Reevaluating Amdahl's law. Communications of the ACM 31, 532–533 (1988)
16. Thekkath, R., Eggers, S.J.: The effectiveness of multiple hardware contexts. In: Proc. of the 6th ASPLOS, pp. 328–337. ACM (1994)
17. Huang, C., Zheng, G., Kalé, L., Kumar, S.: Performance evaluation of Adaptive MPI. In: Proc. 11th ACM SIGPLAN PPoPP, pp. 12–21. ACM (2006)
18. Bouteiller, A., Bouziane, H.L., Herault, T., Lemarinier, P., Cappello, F.: Hybrid preemptive scheduling of message passing interface applications on grids. IJHPCA 20(1), 77–90 (2006)
19. Daly, J.T.: A higher order estimate of the optimum checkpoint interval for restart dumps. FGCS 22(3), 303–312 (2004)