Research report ICL-UT-1301
Multi-criteria checkpointing strategies: optimizing response-time versus resource utilization

Aurélien 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

February 25, 2013

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 in recent years, 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 resiliency. In the projections issued by the International Exascale Software Project (IESP) [10], 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 application life [13].

Many approaches have been investigated, over the years, to resolve the formidable threat that process failures pose to both productivity and efficiency of HPC applications. The most common approach has been the use of rollback recovery, based on periodic, complete application checkpoint, and complete restart upon failure. While this scheme has been successful in the past, it appears that coordinated 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 [12].

Other options are available to design fault tolerant applications, such as Algorithm Based Fault Tolerance [17], or Naturally Fault Tolerant Methods [8]. However, unlike checkpoint/restart, which can be provided automatically (without modifications to the application), these techniques are application-specific, because thet rely on intrinsic algorithmic properties. In addition, they often require excruciating software engineering efforts, resulting in generally low applicability to production codes.

In recent years, an alternative automatic rollback recovery technique, namely uncoordinated checkpoints with message logging, has received a lot of attention [4, 14]. 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 catchup 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 (typically submitted by different users). In addition to per-application completion time, which is strongly challenged by numerous failures, the goal of such a system is to complete as much 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, and overlaps idling time suffered by recovering applications by progress made on other applications loaded on the 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 [1] 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 tradeoffs 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. Section 3 presents the model and the scenarios used to analyze the behavior of the application-centric and platform-centric scenarios. The waste incurred for these scenarios is computed in Section 4. Section 5 is devoted to a comprehensive set of simulations on 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 Uncoordinated Checkpointing Strategy to Maximize Platform Efficiency

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. The closer the checkpoint date is from the failure date, the smaller the amount of lost computation to be re-executed. However, checkpointing is an expensive operation, and incurs overhead of its own, leading the best checkpoint frequency to be a compromise between minimizing the average lost computation per failure, and limiting the amount of computing power wasted on checkpoints. Young [21] and more recently Daly [9] proposed first-order formulas to compute the optimal checkpoint frequency based on the machine MTBF and checkpoint cost.

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; in particular, recovery lines that separate the emission and matching reception event of a message are problematic [7]. Two main families of rollback recovery techniques have been designed to resolve the issues posed by these messages crossing the recovery line: coordinated checkpoint, and uncoordinated checkpoint with message logging. In the coordinated checkpoint approach, a collection of checkpoints is constructed in a way that ensures that consistency threatening messages do not exist between checkpoints of the collection (using a coordination algorithm). As the checkpoint collection forms the only recovery line that is guaranteed to be correct, all processes have to rollback simultaneously, even if they are not faulty. As a result, the bulk amount of lost work is increased and 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 at random dates, 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 [11] store the necessary state elements during the execution of the application. When a process has to rollback 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 the original sender.

In the case of typical HPC applications, which are often tightly coupled, the ability of restarting only faulty process (hence limiting duplicate computation to a minimum) does not translate into great improvements of the application completion time [1]. 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 catchup with the general progress of the application, surviving process spend significant amount of time idling; altogether, the overall application completion time is only marginally improved. However, this conclusion is the result of focusing on the performance of a single application performance. 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 incurred by the whole platform.

In this paper, we propose a scheduling strategy that complements uncoordinated rollback recovery, in order to decrease the waste of computing power during recovery periods. When a failure occurs, a set of spare processes are used to execute the duplicate work of processes that have to rollback to a checkpoint. 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. When the processes reloaded from checkpoint 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 power endured by the platform. 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.

3 Model

In this section, we introduce all model parameters, and we detail the execution scenarios: Application-oriented uses the whole platform for a single application, while Platform-oriented uses a fraction of the resources as spare resources that recover and re-execute work upon failures, while the rest of the resources loads, executes and stores another application.

3.1 Model parameters

All relevant parameters are summarized in Table 1. We give a few words of explanation form each of them. More details can be found in [1]:

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

- μ_p is the MTBF of the platform, meaning that failures strike every μ_p seconds in average. We have $\mu_p = \frac{\mu_{ind}}{p_{total}}$, where μ_{ind} is the MTBF of individual processors, and p_{total} is the total number of processors. 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
- The platform is partitioned into processor groups. We have $G+1$ processor groups, each of size q (hence $(G+1)q = p_{total}$). One of these groups will be used as spare group in the PLATFORM-ORIENTED scenario, while all $G+1$ participate to execution in the APPLICATION-ORIENTED scenario (see Section 3.2 below). We use a hierarchical protocol with message-logging in both cases.
- Checkpoints are taken periodically, every T seconds. All groups checkpoint concurrently, in time C. 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: an 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 extra-communications induced by message logging. A typical value is $\lambda = 0.98$ [2, 14];
  - Altogether, the amount of work W that is executed every period of length T is

$$W = \lambda((T - C) + \alpha C) = \lambda(T - (1 - \alpha)C) \quad (1)$$

- In addition to the durations of the checkpoint C, we use D for the downtime and R for the time to restart from a checkpoint. We assume that $D \leq C$ to avoid clumsy expressions, and because it is always the case in practice. However, we can easily extend the analysis to the case where $D > C$.
- Message logging has both a positive impact and a negative impact on performance:
  - On the positive side, message logging reduces the re-execution time after a failure, because inter-group messages are stored in memory and directly accessible during the recovery. Our model accounts for this by introducing a speed-up factor ρ during the re-execution. Typical values for ρ lie in the interval $[1; 2]$, meaning that re-execution time can be reduced up to a half for some applications [5].
  - On the negative side (in addition to execution slow-down by the factor λ), message-logging increases the size of checkpoints. Because inter-group messages are logged, 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). Typical values of β are given in the examples of Section 5.
- 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 PLATFORM-ORIENTED scenario. As a consequence, when processor local storage is available, C_0 is increased by $\frac{G+1}{G}$ in PLATFORM-ORIENTED, compared to the APPLICATION-ORIENTED case.

3.2 Scenarios

In this section, we introduce some notations for both execution scenarios. The objective is to compare the overhead incurred for each of them. We define the *waste* as the fraction of time where resources are not used to perform useful work, and we aim at comparing the value of the minimum waste for each scenario. This minimum waste will be achieved for some optimal value of the period, which will likely differ for each scenario.

Scenario Application-oriented. One single application executes on the entire platform, using all $G+1$ groups. Resilience is provided through the standard uncoordinated hierarchical protocol. We let $\text{WASTE}_{app}(T)$ denote the waste induced by this scenario, and T_{app}^{opt} the value of T that minimizes it.

Scenario Platform-oriented. One main application executes on G groups and uses the spare group in case of failure. During the downtime, restart and re-execution, another application is loaded, if time permits. The utilization rate of the platform is higher this way, but at the price of using a spare group. We use the notations $\text{WASTE}_{plat}(T)$ for the waste, and T_{plat}^{opt} for the value of T that minimizes it.

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 tradeoff when using the optimal period of a scenario for the other one.

4 Computing the waste

In this section, we show how to compute the waste for both scenarios. We start with the generic approach to compute the waste, which we specialize later for each scenario.

4.1 Generic approach

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

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

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 $(1 - \frac{t_{lost}}{\mu_p})T_{final} = T_{ff}$ which we rewrite as

$$(1 - \text{WASTE}_{fail})T_{final} = T_{ff}, \textit{ where } \text{WASTE}_{fail} = \frac{t_{lost}}{\mu_p} \tag{5}$$

Here WASTE_{fail} denotes the waste due to failures. Combining Equations (4) and (5), we derive that

$$(1 - \text{WASTE}_{final})T_{final} = T_{base} \tag{6}$$

$$\text{WASTE}_{final} = \text{WASTE}_{ff} + \text{WASTE}_{fail} - \text{WASTE}_{ff}\text{WASTE}_{fail} \tag{7}$$

Here 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 WASTE_{final} for each scenario.

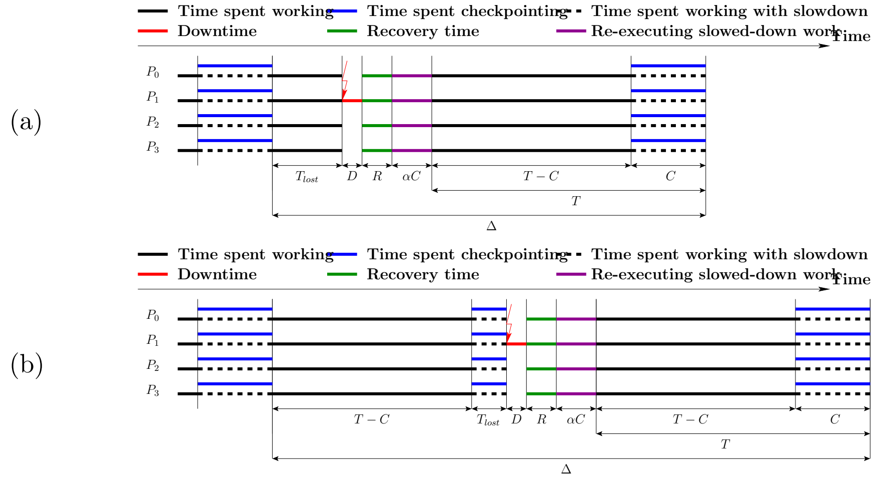

Figure 1: Uncoordinated checkpoint: illustrating the waste when a failure occurs (a) during the work phase; and (b) during the checkpoint phase.

Aurélien a promis de refaire cette figure 1 mais il peut peut être négocier avec Thomas pour avoir qq chose qui ressemble à la figure de l'autre scénario. En tout cas il faudrait voir un groupe re-executer et tous les autres attendre.

4.2 Scenario Application-oriented

We have $\text{WASTE}_{ff} = \frac{T-W}{T} = \frac{T-\lambda(T-(1-\alpha)C)}{T}$ for both scenarios but recall that we enroll $G+1$ groups in scenario APPLICATION-ORIENTED and only G groups in in scenario PLATFORM-ORIENTED, so that the value of C is not the same in

Equation (3). Next, we compute the value of WASTE_{fail} for the APPLICATION-ORIENTED scenario. We illustrate this computation with Figure 1:

$$\text{WASTE}_{fail} = \frac{1}{\mu_p}\left[D + R + \frac{T-C}{T} \times \text{ReExec}_1 + \frac{C}{T} \times \text{ReExec}_2\right] \quad (8)$$

where

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

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

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. Assume first that the failure strikes during work interval $(T - C)$, see Figure 1(a): we need to re-execute the work done in parallel with the last checkpoint (of duration C). This takes a time αC since no checkpoint activity is taking place during that replay. 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 expected value $\frac{T-C}{2}$. We derive the value of ReExec_1 by accounting for the speedup in re-execution (parameter ρ). This value is weighted by the probability $(T-C)/T$ of the failure striking within the work interval. We derive the value of ReExec_2 with a similar reasoning (see Figure 1(b)), and weight it by the probability C/T of the failure striking within the checkpoint interval. After simplification, we derive

$$\text{WASTE}_{fail} = \frac{1}{\mu_p}\left(D + R + \frac{1}{\rho}\left(\frac{T}{2} + \alpha C\right)\right) \quad (9)$$

4.3 Scenario Platform-oriented

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:

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

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

We now proceed to the computation of WASTE_{fail}, which is intricate. See Figure 2 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.

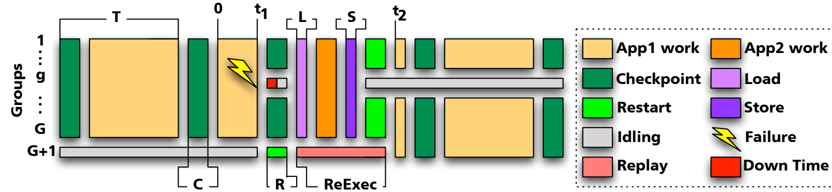

Figure 2: Illustrating the waste when a fault occurs in the PLATFORM-ORIENTED scenario.

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

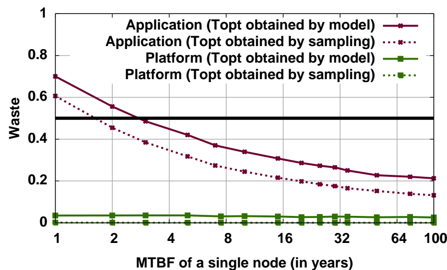

Figure 3: Waste as function of the compute node MTBF, considering a Matrix multiplication, on the K-Computer model, with checkpoints at current technology speed.

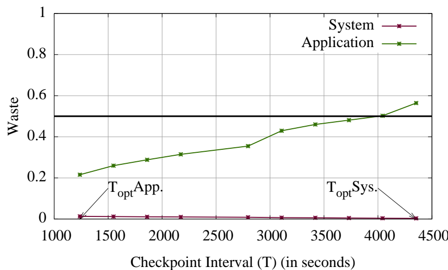

Figure 4: Waste as function of the checkpoint interval, considering a 2D-Stencil operation, on the Fat Exascale Computer model, with checkpoints at 10 times the speed of currently projected technology. (MTBF of a component of 20 years)

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

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

and is explained as follows:

- 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 Experiments

We instantiated the proposed waste model with different scenarios. We first detail two scenarios that illustrate the benefits of the proposed approach. Then

we provide comprehensive results for a wider set of experiments, whose results are similar.

The first scenario shown in Fig. 3, considers a model of the K-Computer ([18]), and a matrix-matrix multiply operation, assuming that the whole machine (less a group) is used and the problem size occupies the whole machine memory. Using the waste model, we computed the optimal checkpoint time, and simulated the waste as a function of the node MTBF using an in-house simulator adapted from [1], from the system perspective (green lines) and from the application perspective (red lines). We also sampled other checkpoint interval times, to take into account multiple failure scenarios: although mathematical limitations prevent us to compute a close formula for the waste in case of overlapping failures, the simulator is not limited, and take into account all scenarios. Note, however, that the model is still required, to provide a basis for the checkpoint interval, and that this interval provides a good approximation of the best, as is shown in the figure.

This figure demonstrates the huge benefit, from the application perspective, of introducing a spare node: indeed, when 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 system waste itself is at significantly low levels.

The second scenario is illustrated in Figure 4: here, we fix the MTBF of a single component to 20 years, and study the impact of choosing the optimal checkpoint interval targeting the system efficiency, or the application efficiency. To do so, we varied that checkpoint interval between the Application optimal, and the System optimal, 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 (the "Fat" version, featuring heavy multicore nodes), and the modeled application is a 2D-Stencil application that fills the systems memory. As evaluated in [1], rollback/recovery protocols will be efficient in such a machine, only if there is a 10 fold increase in performance (or more) of checkpointing techniques, so we place ourself in this scenario. Figures 5, 6, 7, and 8present an exhaustive study on different sets of machines, applications and checkpoint performance models, and they conclude to the same general behavior: system's optimal checkpoint intervals are much higher than application's optimal checkpoint intervals of the same scenario, and both system and application exhibit a waste that increases when taking a checkpoint interval far away from their optimal. However, because the spare node is so much more beneficial to the general efficiency of the system than to the efficiency of the application, it is extremely beneficial to select the optimal application checkpoint interval: the performance of the system remains close to an efficiency of 1, while the waste of the application can be reduced significantly.

As a side note, one can also see that although replication (with a top efficiency of 50%) could be considered to improve the efficiency of the applications, in the scenarios where rollback-recovery can be less efficient than replication, a hierarchical checkpointing technique with dedicated spare node, as the one we propose here, is the only one that can provide a waste for the system close to 0.

As already mentioned, results with other platforms, applications and failure distributions exhibit similar behavior, as shown in Figures 5, 6, 7, and 8. In these figures, *Platform/X* corresponds to experiments on the corresponding *Platform*, where the checkpoint time C_0 is divided by X, where $X = \{1, 10, 100\}$. Note that C_0 is the time to write the memory footprint of one application group onto stable storage; detailed information on the platform parameters are available in [1]. Dividing by the factor X allows up to investigate whether, and up to what extent, faster checkpointing can prove useful, or necessary, at very large scale.

6 Related work

Fault tolerance and rollback recovery has been a very active field of research [11]. Recent optimizations and experimental studies outline that compelling performance can be obtained from uncoordinated checkpointing [4, 14], and have characterized with more precision the typical range of values for fixed overheads (such as the slowdown imposed by checkpointing, message logging, and message log growth rate). In some recent work, we have proposed a general model capturing the intricacies of these advanced rollback recovery techniques [1]. However, the model 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.

Another interesting development designed to take advantage of the idling time left on surviving processors with uncoordinated rollback recovery is parallel re-execution of the lost workload. Upon restart, the workload initially executed by the failed processes is split and dispatched across all computing resources [6]. In this work, the workload can be easily divided as the program is written with Charm++. In practice, however, the deployment of such techniques for production MPI codes written in legacy Fortran/C is difficult, as it requires a full rewrite of the application to account for the different data and computation distribution during recovery periods (some works propose to partially automate this with compilation techniques [20]). Even when such splitting is practical, the resulting scalability is challenged, as the workload to be re-executed after a partial rollback is (hopefully) orders of magnitude smaller than the initial application workload, which, in accordance with Gustafson law [15], typically results in poor parallel efficiency at scale.

Overlapping downtime of programs blocked on I/O or memory accesses is an idea that has been investigated in many contexts, and has resulted in a variety of hardware and software techniques to improve throughput (Hyperthreads [19], massive oversubscription in task based systems [16], etc.) . Interestingly, checkpoint-restart can be used as a tool designed to improve overlap of computation and computation with co-scheduling [3]. However, it has seldom been considered to overcome the cost of rollback recovery itself, and modeling tools to assess the effectiveness of compensation techniques have not been available yet, to the best of our knowledge. The model proposed here permits to

characterize the difference in terms of platform efficiency when multiple independent applications must be completed.

7 Conclusion

In this paper, 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. We recall that 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 term of application runtime and in term of 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.

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

References

[1] Georges Bosilca, Aurelien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. Research report RR-7950, INRIA, 2012.

[2] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

[3] Aurelien Bouteiller, Hinde-Lilia Bouziane, Thomas Herault, Pierre Lemarinier, and Franck Cappello. Hybrid preemptive scheduling of message passing interface applications on grids. *International Journal of High Performance Computing Applications*, 20(1):77–90, 2006.

[4] Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack J. Dongarra. Correlated set coordination in fault tolerant message logging protocols. In *Proc. of Euro-Par'11 (II)*, volume 6853 of *LNCS*, pages 51–64. Springer, 2011.

[5] Aurelien Bouteiller, Thomas Herault, Geraud Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V: a multiprotocol fault tolerant MPI. *IJHPCA*, 20(3):319–333, 2006.

[6] Sayantan Chakravorty and L.V. Kale. A fault tolerance protocol with fast fault recovery. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –10, march 2007.

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

[8] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proc. 10th ACM SIGPLAN symp. on Principles and practice of parallel programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005. ACM.

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

[10] Jack Dongarra, Pete Beckman, Patrick Aerts, Frank Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Terry Moore, Rick Stevens, Anne Trefethen, and Mateo 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.

[11] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Survey*, 34:375–408, 2002.

[12] K. Ferreira, J. Stearley, J. H. III Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proceedings of the 2011 ACM/IEEE Conf. on Supercomputing*, 2011.

[13] G. Gibson. Failure tolerance in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022, 2007.

[14] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. Hydee: Failure containment without event logging for large scale send-deterministic mpi applications. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1216 –1227, may 2012.

[15] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[16] Chao Huang, Gengbin Zheng, Laxmikant Kalé, and Sameer Kumar. Performance evaluation of adaptive mpi. In *Proc. 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 12–21, New York, NY, USA, 2006. ACM.

[17] K.H. Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 100(6):518–528, 1984.

[18] Hiroyuki Miyazaki, Yoshihiro Kusano, Hiroshi Okano, Tatsumi Nakada, Ken Seki, Toshiyuki Shimizu, Naoki Shinjo, Fumiyoshi Shoji, Atsuya Uno, and Motoyoshi Kurokawa. K computer: 8.162 petaflops massively parallel scalar supercomputer built with over 548k cores. In *ISSCC*, pages 192–194. IEEE, 2012.

[19] Radhika Thekkath and Susan J. Eggers. The effectiveness of multiple hardware contexts. In *Proc. 6th int. conf. on Architectural support for programming languages and operating systems*, ASPLOS VI, pages 328–337, New York, NY, USA, 1994. ACM.

[20] Xuejun Yang, Yunfei Du, Panfeng Wang, Hongyi Fu, and Jia Jia. Ftpa: Supporting fault-tolerant parallel computing through parallel recomputing. *Parallel and Distributed Systems, IEEE Transactions on*, 20(10):1471 – 1486, oct. 2009.

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

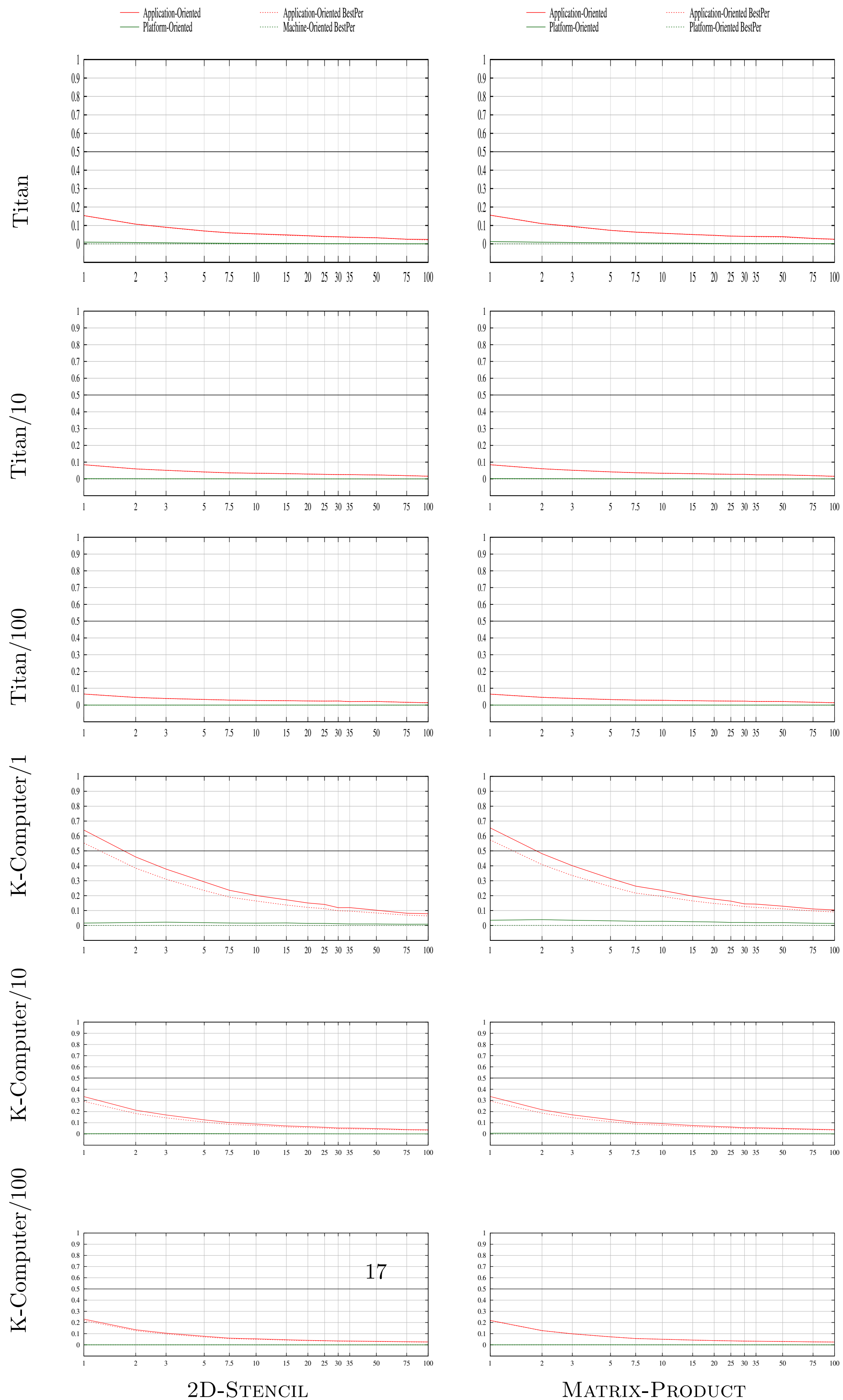

Figure 5: Waste as a function of processor MTBF μ, for an Exponential Distribution, Current platforms

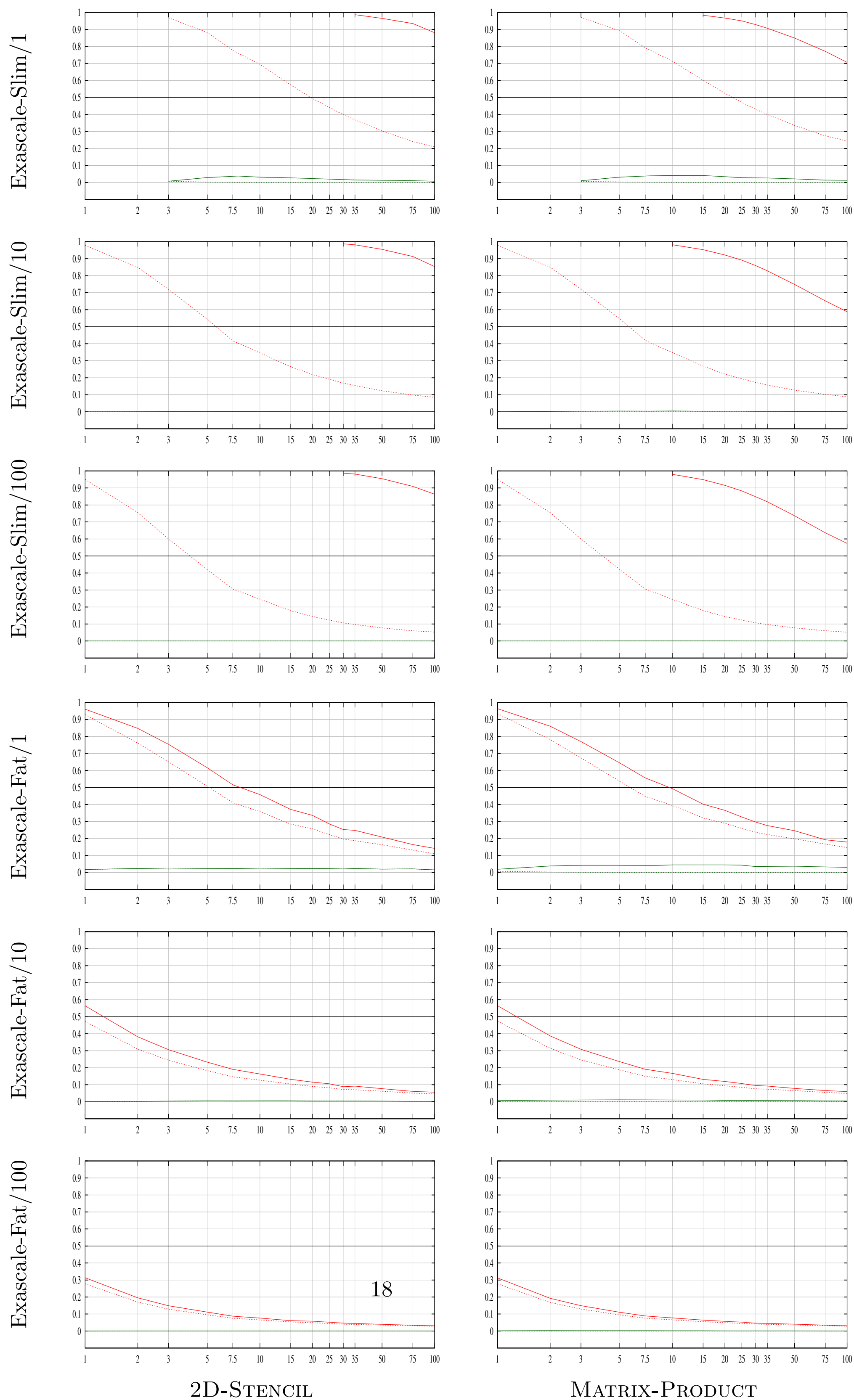

Figure 6: Waste as a function of processor MTBF μ, for an Exponential Distribution, Exascale platforms

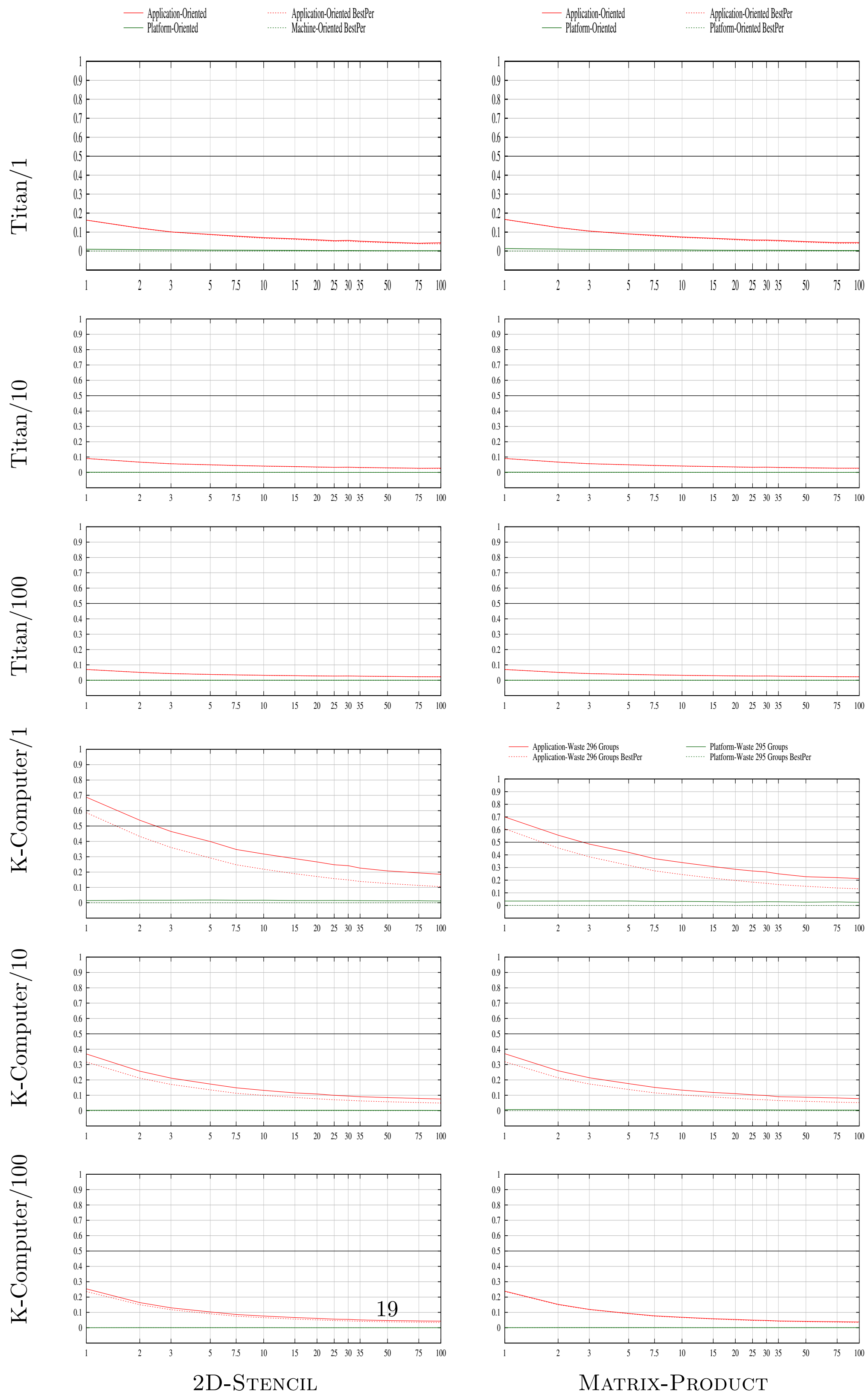

Figure 7: Waste as a function of processor MTBF μ, for a Weibull Distribution k=0.7, Current platforms

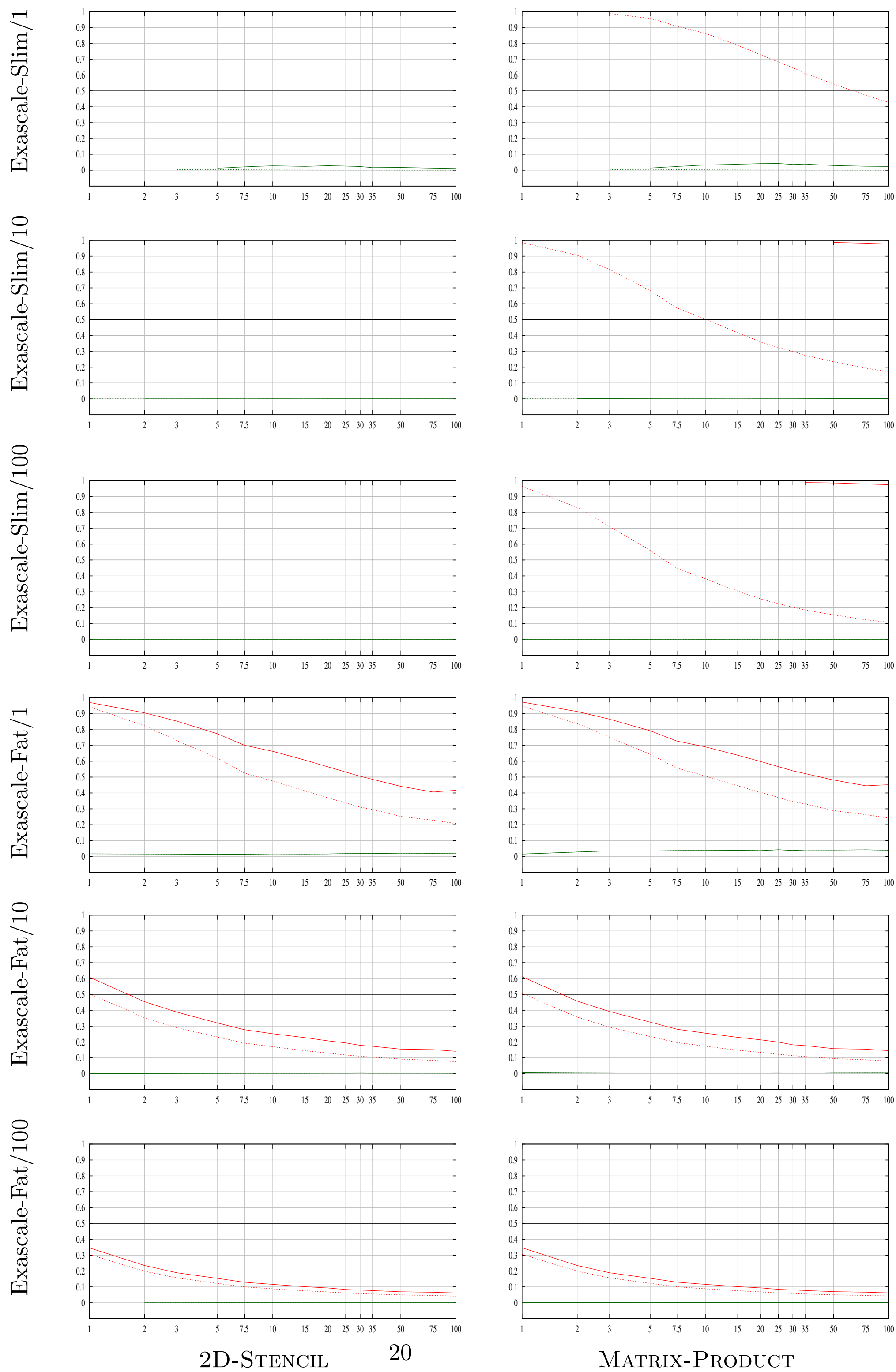

Figure 8: Waste as a function of processor MTBF μ, for a Weibull Distribution k=0.7, Exascale platforms