HAL
open science

Assessing general-purpose algorithms to cope with fail-stop and silent errors

Anne Benoit, Aurélien Cavelan, Yves Robert, Hongyang Sun

To cite this version:

Anne Benoit, Aurélien Cavelan, Yves Robert, Hongyang Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. [Research Report] RR-8599, 2014. hal-01066664v1

HAL Id: hal-01066664

https://inria.hal.science/hal-01066664v1

Submitted on 23 Sep 2014 (v1), last revised 9 Feb 2016 (v5)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Assessing general-purpose algorithms to cope with fail-stop and silent errors

Anne Benoit, Aurélien Cavelan, Yves Robert, Hongyang Sun

RESEARCH REPORT
N° 8599
19 September 2014
Project-Teams ROMA

ISSN 0249-6399 ISRN INRIA/RR--8599--FR+ENG

Assessing general-purpose algorithms to cope with fail-stop and silent errors

Anne Benoit*, Aurélien Cavelan*, Yves Robert*†,
Hongyang Sun*

Project-Teams ROMA

Research Report n° 8599 — 19 September 2014 — 34 pages

Abstract: In this paper, we combine the traditional checkpointing and rollback recovery strategies with verification mechanisms to address both fail-stop and silent errors. The objective is to minimize either makespan or energy consumption. While DVFS is a popular approach for reducing the energy consumption, using lower speeds/voltages can increase the number of errors, thereby complicating the problem. We consider an application workflow whose dependence graph is a chain of tasks, and we study three execution scenarios: (i) a single speed is used during the whole execution; (ii) a second, possibly higher speed is used for any potential re-execution; (iii) different pairs of speeds can be used throughout the execution. For each scenario, we determine the optimal checkpointing and verification locations (and the optimal speeds for the third scenario) to minimize either objective. The different execution scenarios are then assessed and compared through an extensive set of experiments.

Key-words: No keywords

* Ecole Normale Superieure de Lyon, CNRS & INRIA, France
† University of Tennessee Knoxville, USA

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Evaluation d'algorithmes génériques tolérant pannes et erreurs silencieuses

Résumé : Dans cet article, nous combinons les techniques traditionnelles de prise de points de sauvegarde (checkpoint) avec des mécanismes de vérification afin de prendre en compte à la fois les pannes et les corruptions mémoire silencieuses. L'objectif est soit de minimiser le temps d'exécution, soit de minimiser la consommation d'énergie. DVFS est une approche populaire pour réduire la consommation d'énergie, mais utiliser des vitesses ou des fréquences trop basses peut accroître le nombre d'erreurs, et ainsi compliquer le problème. Nous considérons des applications dont le graphe de dépendance est une chaîne de tâches, et nous étudions trois scénarios: (i) une seule vitesse est utilisée pendant toute la durée de l'exécution; (ii) une seconde vitesse, pouvant être plus élevée, est utilisée pour toutes les ré-exécutions potentielles; (iii) différentes paires de vitesses peuvent être utilisées entre deux checkpoints pendant l'exécution. Pour chaque scénario, nous déterminons le placement optimal des checkpoints et des vérifications (et les vitesses optimales pour le troisième scénario) afin de minimiser l'un ou l'autre des objectifs. Les différents scénarios d'exécution sont ensuite testés et comparés à l'aide de simulations.

Mots-clés : HPC, tolérance aux erreurs, pannes, point de sauvegarde, checkpoint, corruption silencieuse de données, vérification, correction d'erreurs

1 Introduction

For HPC applications, scale is a major opportunity. Massive parallelism with 100,000+ nodes is the most viable path to achieving sustained petascale performance. Future platforms will enrol even more computing resources to enter the exascale era.

Unfortunately, scale is also a major threat. Resilience is the first challenge. Even if each node provides an individual MTBF (Mean Time Between Failures) of, say, one century, a machine with 100,000 such nodes will encounter a failure every 9 hours in average, which is larger than the execution time of many HPC applications. Furthermore, a one-century MTBF per node is an optimistic figure, given that each node is composed of several hundreds of cores. Worse, several types of errors need to be considered when computing at scale. In addition to classical fail-stop errors (such as hardware failures), silent errors (a.k.a silent data corruptions) constitute another threat that cannot be ignored any longer.

Another challenge is energy consumption. The power requirement of current petascale platforms is that of a small town, hence measures must be taken to reduce the energy consumption of future platforms. A widely-used strategy is to use DVFS techniques: modern processors can run at different speeds, and lower speeds induce big savings in energy consumption. In a nutshell, this is because the dynamic power consumed when computing at speed s is proportional to s^3, while execution time is proportional to $1/s$. As a result, computing energy (which is time times power) is proportional to s^2. However, static power must be accounted for, and this static power is paid throughout the duration of the execution, which calls for a shorter execution (at high speed). Overall there are trade-offs to be found, but in most practical settings, using lower speeds reduces global energy consumption.

To further complicate the picture, energy savings have an impact on resilience. Obviously, the longer the execution, the higher the expected number of errors, hence using a lower speed to save energy may well induce extra time and overhead to cope with more errors throughout execution. Even worse (again!), lower speeds are usually obtained via lower voltages, which themselves induce higher error rates and further increase the latter overhead.

In this paper, we introduce a model that addresses both challenges, resilience and energy-consumption. In addition, we address both fail-stop and silent errors, which, to the best of our knowledge, has never been achieved before. While checkpoint and roll-back recovery is the de-facto standard for dealing with fail-stop errors, there is no widely adopted general-purpose technique to cope with silent errors. The problem with silent errors is *detection latency*: contrarily to a fail-stop error whose detection is immediate, a silent error is identified only when the corrupted data is activated and/or leads to an unusual application behavior. However, checkpoint and rollback recovery assumes instantaneous error detection, and this raises a new difficulty: if the error stroke before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted, and cannot be used to restore the application. To solve this problem, one may envision to keep several checkpoints in memory, and to restore the application from the last *valid* checkpoint, thereby rolling back to the last *correct* state of the application [23]. This multiple-checkpoint approach has three major drawbacks. First, it is very demanding in terms of stable storage: each checkpoint typically represents a copy of the entire memory footprint of the application, which may well correspond to several terabytes. The second drawback is the possibility of fatal failures. Indeed, if we keep k checkpoints in memory, the approach assumes that the error that is currently detected did not strike before all the checkpoints still kept in memory, which would be fatal: in that latter case, all live checkpoints are corrupted, and one would have to re-execute the entire application from

scratch. The probability of a fatal failure is evaluated in [2] for various error distribution laws and values of k. The third drawback of the approach is the most serious, and applies even without memory constraints, i.e., if we could store an infinite number of checkpoints in storage. The critical question is to determine which checkpoint is the last valid one. We need this information to safely recover from that point on. However, because of the detection latency, we do not know when the silent error has indeed occurred, hence we cannot identify the last valid checkpoint, unless some verification system is enforced.

We introduce such a verification system in this paper. This approach is agnostic of the nature of this verification mechanism (checksum, error correcting code, coherence tests, etc.). This approach is also fully general-purpose, although application-specific information, if available, can always be used to decrease the cost of verification: see the overview of related work in Section 2 for examples.

In this context, the simplest protocol is to take only verified checkpoints. This corresponds to performing a verification just before taking each checkpoint. If the verification succeeds, then one can safely store the checkpoint. If the verification fails, then a silent error has struck since the last checkpoint, which was duly verified, and one can safely recover from that checkpoint to resume the execution of the application. Of course, if a fail-stop error strikes, we also safely recover from the last checkpoint, just as in the classical checkpoint and roll-back recovery method. This VC-ONLY protocol basically amounts to replacing the cost C of a checkpoint by the cost $V + C$ of a verification followed by a checkpoint. However, because we deal with two sources of errors, the one detected immediately and the other only when we reach the verification, the analysis of the optimal strategy is more involved. We extend both the classical bound by Young [33] or Daly [11], and the dynamic programming algorithm of Toueg and Babaoglu [31], to deal with these error sources.

While taking checkpoints without verifications seem a bad idea (because of the memory cost, and of the risk of saving corrupted data), taking a verification without checkpointing may be interesting. Indeed, if silent errors are frequent enough, it is worth verifying the data in between two (verified) checkpoints, so as to detect a possible silent error earlier in the execution, and thereby re-executing less work. We refer to VC+V as the protocol that allows for both verified checkpoints and isolated verifications.

The major objective of this paper is to study VC+V algorithms coupling verification and checkpointing, and to analytically determine the best balance of verifications between checkpoints so as to minimize either makespan (total execution time) or energy consumption. To achieve this ambitious goal, we restrict to a simplified, yet realistic, application framework. We consider application workflows that consist of a number of parallel tasks that execute on the platform, and that exchange data at the end of their execution. In other words the task graph is a linear chain, and each task (except maybe the first one and the last one) reads data from its predecessor and produces data for its successor. This scenario corresponds to a high-performance computing application whose workflow is partitioned into a succession of (typically large) tightly-coupled computational kernels, each of them being identified as a task by the model. At the end of each task, we have the opportunity either to perform a verification of the task output, or to perform a verification followed by a checkpoint.

In addition, we have to select a speed for each execution of each task. We envision three different scenarios. In the simple SINGLESPEED scenario, a unique speed s is available throughout execution. In the intermediate REEXECSPEED scenario, the same speed s is used for the first execution of each task, but another speed σ is available for re-execution after a fail-stop or silent error. Here the first speed s can be seen as the regular speed, while the second

speed σ corresponds to a higher speed (possibly obtained via over-clocking) to speed-up re-execution after an error. Finally, in the advanced MultiSpeed scenario, two different speeds s_i and σ_i can be used to execute the tasks in between two consecutive checkpoints (which we call a task segment). Each speed s_i or σ_i can be freely chosen among a set of K discrete speeds. Note that these speeds may well vary from one segment to another. Obviously, the ReExecSpeed scenario is a sub case of the MultiSpeed scenario, but dealing with the former first is useful to understand the optimal algorithms from the latter. For each scenario, we provide an optimal dynamic programming algorithm to determine the best repartition of checkpoints and verifications (and for the advanced scenario we provide the corresponding optimal pair of speeds for each segment).

The main contributions of this paper are the following:

- We introduce a general-purpose model to deal with both fail-stop and silent errors, combining periodic checkpoints with a verification mechanism
- We consider several execution scenarios, first with a single speed, and then with several discrete speeds that can freely change after each checkpoint, and also in case of re-execution
- For all scenarios, and for both the makespan and energy objectives, we consider two approaches, one using verified checkpoints only, and another using additional isolated verifications. We provide a dynamic-programming algorithm that determines the best repartition of checkpoints and of verifications across application tasks for each scenario/approach/objective combination.
- We provide an extensive set of simulations that fully supports the theory and enables us to assess the usefulness of each algorithm.

The rest of the paper is organised as follows. Section 2 provides an overview of related work. Section 3 is devoted to formally defining the framework and all model parameters. The next three sections deal with the main algorithmic contributions: we deal with the three execution scenarios, and design optimal algorithms for the VC-only approach, and then for the VC+V approach, targeting either time or energy minimization. Then in Section 7, we report on a comprehensive set of experiments to assess the impact of each scenario and approach. Finally, we outline main conclusions and directions for future work in Section 8.

2 Related work

2.1 Fail-stop errors

The de-facto general-purpose error recovery technique in high performance computing is checkpoint and rollback recovery [9, 16]. Such protocols employ checkpoints to periodically save the state of a parallel application, so that when an error strikes some process, the application can be restored into one of its former states. There are several families of checkpointing protocols, but they share a common feature: each checkpoint forms a consistent recovery line, i.e., when an error is detected, one can rollback to the last checkpoint and resume execution, after a downtime and a recovery time.

Many models are available to understand the behavior of checkpoint/restart [33, 11, 25, 7]. For a divisible-load application where checkpoints can be inserted at any point in execution for a nominal cost C, there exist well-known formulas due to Young [33] and Daly [11] to determine the optimal checkpointing period. For an application composed of a linear chain of tasks (which is also the subject of this paper), the problem of finding the optimal checkpoint

strategy, i.e., of determining which tasks to checkpoint, in order to minimize the expected execution time, has been solved by Toueg and Babaoglu [31], using a dynamic programming algorithm.

One major contribution of this paper is to extend both the Young/Daly formulas [33, 11] and the result of Toueg and Babaoglu [31] to deal with silent errors in addition to fail-stop errors, and with several discrete speeds instead of a single one.

2.2 Silent errors

Most traditional approaches maintain a single checkpoint. If the checkpoint file includes errors, the application faces an irrecoverable failure and must restart from scratch. This is because error detection latency is ignored in traditional rollback and recovery schemes. These schemes assume instantaneous error detection (therefore mainly targeting fail-stop failures) and are unable to accommodate silent errors. We focus in this section on related work about silent errors. A comprehensive list of techniques and references is provided by Lu, Zheng and Chien in [23].

Considerable efforts have been directed at error-checking to reveal silent errors. Error detection is usually very costly. Hardware mechanisms, such as ECC memory, can detect and even correct a fraction of errors, but in practice they are complemented with software techniques. The simplest technique is triple modular redundancy and voting [24], which induces a highly costly verification. For high-performance scientific applications, process replication (each process is equipped with a replica, and messages are quadruplicated) is proposed in the RedMPI library [18]. Elliot et al. [15] combine partial redundancy and checkpointing, and confirm the benefit of dual and triple redundancy. The drawback is that twice the number of processing resources is required (for dual redundancy).

Application-specific information can be very useful to enable ad-hoc solutions, that dramatically decrease the cost of detection. Many techniques have been advocated. They include memory scrubbing [22], but also ABFT techniques [21, 6, 30], such as coding for the sparse-matrix vector multiplication kernel [30], and coupling a higher-order with a lower-order scheme for PDEs [5]. These methods can only detect an error but do not correct it. Self-stabilizing corrections after error detection in the conjugate gradient method are investigated by Sao and Vuduc [28]. Also, Heroux and Hoemmen [19] design a fault-tolerant GMRES capable of converging despite silent errors, and Bronevetsky and de Supinski [8] provide a comparative study of detection costs for iterative methods.

A nice instantiation of the checkpoint and verification mechanism that we study in this paper is provided by Chen [10], who deals with sparse iterative solvers. Consider a simple method such as the PCG, the Preconditioned Conjugate Gradient method: Chen's approach performs a periodic verification every d iterations, and a periodic checkpoint every $d \times c$ iterations, which is a particular case of the VC+V approach with equidistant verifications. For PCG, the verification amounts to check the orthogonality of two vectors and to recompute and check the residual. The cost of the verification is small in front of the cost of an iteration, especially when the preconditioner requires much more flops than a sparse matrix-vector product.

As already mentioned, our work is agnostic of the underlying error-detection technique and takes the cost of verification as an input parameter to the model.

2.3 Energy model and error rate

Modern processors are equipped with *dynamic voltage and frequency scaling* (DVFS) capability. The total power consumption is the sum of the static/idle power and the dynamic power, which is proportional to the cube of the processing speed s [32, 4], i.e., $P(s) = P_{idle} + \beta \cdot s^3$, where $\beta > 0$. A widely used reliability model assumes that radiation-induced transient faults (soft errors) follow a Poisson process with an average arrival rate λ. The impact of DVFS on the error rate is, however, not completely clear.

On the one hand, lowering the voltage/frequency is believed to have an adverse effect on the system reliability [13, 35]. In particular, many papers (e.g., [35, 34, 3, 12]) have assumed the following exponential error rate model:

$$\lambda(s) = \lambda_0 \cdot 10^{\frac{d(s_{\max} - s)}{s_{\max} - s_{\min}}} , \tag{1}$$

where λ_0 denotes the average error rate at the maximum speed $s_{\max}$, $d > 0$ is a constant indicating the sensitivity of error rate to voltage/frequency scaling, and $s_{\min}$ is the minimum speed. This model suggests that the error rate increases exponentially with decreased processing speed, which is a result of decreasing the voltage/frequency and hence lowering the circuit's critical charge (i.e., the minimum charge required to cause an error in the circuit).

On the other hand, the failure rates of computing nodes have also been observed to increase with temperature [26, 17, 20, 29], which generally increases together with the processing speed (voltage/frequency). As a rule of thumb, Arrenhius' equation when applied to microelectronic devices suggests that the error rate doubles for every 10°C increase in the temperature [17]. In general, the mean time between failure (MTBF) of a processor, which is the reciprocal of failure rate, can be expressed as [29]:

$$MTBF = \frac{1}{\lambda} = A \cdot e^{-b \cdot T} , \tag{2}$$

where A and b are thermal constants, and T denotes the temperature of the processor. Under the reasonable assumption that higher operating voltage/frequency leads to higher temperature, this model suggests that the error rate increases with increased processing speed.

Clearly, the two models above draw contradictory conclusions on the impact of DVFS on error rates. In practice, the impact of the first model may be more evident, as the temperature dependency in some systems has been observed to be linear (or even not exist) instead of being exponential [14]. Generally speaking, the processing speed should have a composite effect on the average error rate by taking both voltage level and temperature into account. In the experimental section of this paper (see Section 7), we adopt a tradeoff model and modify Equation (1) to include the impact of temperature.

3 Framework

In this section we introduce all model parameters. For reference, main notations are summarized in Table 1. We start with a description of the application workflows. Then we present parameters related to energy consumption. Next we detail the resilient model to deal with fail-stop and silent errors. We conclude by presenting the various execution scenarios.

Tasks	
$\{T_1, T_2, \ldots, T_n\}$	Set of n tasks
w_i	Computational cost of task T_i
Speeds	
s	Regular speed
σ	Re-execution speed
$\{s_1, s_2, \ldots, s_K\}$	Set of K discrete computing speeds (DVFS)
Time	
$T_{i,j}(s)$	Time needed to execute tasks T_i to T_j at speed s
$V_i(s)$	Time needed to verify task T_i at speed s
C_i	Time needed to checkpoint task T_i
R_i	Time needed to recover from task T_i
Resilience	
$\lambda^F(s)$	Fail-stop error rate for a given speed s
$\lambda^S(s)$	Silent error rate for a given speed s
$p^F_{i,j}(s)$	Probability that a fail-stop error strikes between tasks T_i and T_j
$p^S_{i,j}(s)$	Probability that a silent error strikes between tasks T_i and T_j
Energy	
P_{idle}	Static power dissipated when the platform is switched on
$P_{cpu}(s)$	Dynamic power spent by operating the CPU at speed s
P_{io}	Dynamic power spent by I/O transfers (checkpoints and recoveries)
$E_{i,j}(s)$	Energy needed to execute tasks T_i to T_j
$E^{\mathrm{V}}_i(s)$	Energy needed to verify task T_i
E^{C}_i	Energy needed to checkpoint task T_i
E^{R}_i	Energy needed to recover from task T_i

Table 1: List of main notations.

3.1 Application workflows

We consider application workflows whose task graph is a linear chain $T_1 \to T_2 \cdots \to T_n$. Here n is the number of tasks, and each task T_i is weighted by its computational cost w_i. We target a platform with p identical processors. Each task is a parallel task that is executed on the whole platform. A fundamental characteristic of the application model is that it allows to view the platform as a single (albeit very powerful) *macro-processor*, thereby providing a tractable abstraction of the problem.

3.2 Energy consumption

When computing, we use DVFS capabilities to change the speed of the processors, and assume a set $S = \{s_1, s_2, \ldots, s_K\}$ of K discrete computing speeds. On the contrary when checkpointing, we assume a dedicated (constant) power consumption. Altogether, the total power consumption of the macro-processor is p times the power consumption of each individual resource. It is decomposed into three different components:

- P_{idle}, the static power dissipated when the platform is on (even idle)
- $P_{cpu}(s)$, the dynamic power spent by operating the CPU at speed s
- P_{io}, the dynamic power spent by I/O transfers (checkpoints and recoveries).

Assuming w.l.o.g. no overlap between CPU operations and I/O transfers. Then the total energy consumption $Energy$ during the execution of the application can be expressed as

$$Energy = P_{idle}(T_{cpu} + T_{io}) + \sum_{i=1}^{K} P_{cpu}(s_i)T_{cpu}(s_i) + P_{io}T_{io} \tag{3}$$

where $T_{cpu}(s_i)$ is the total time spent computing at speed s_i, $T_{cpu} = \sum_{i=1}^{K} T_{cpu}(s_i)$ is the total time spent computing, and T_{io} is the total time spent for I/O transfers.

The time to compute tasks T_i to T_j at speed s is $T_{i,j}(s) = \frac{1}{s}\sum_{k=i}^{j} w_i$ and the corresponding energy is $E_{i,j}(s) = T_{i,j}(s)(P_{idle} + P_{cpu}(s))$.

3.3 Resilience

We assume that errors only strike during computations, and not during I/O transfers (checkpoints and recoveries) nor verifications. We consider two types of errors, *fail-stop* and *silent*. To cope with fail-stop errors, we use checkpointing. The time to checkpoint after task T_i is C_i, and the time to recover from task T_i is R_i. Fail-stop errors follow an Exponential probability distribution of parameter $\lambda^F(s)$, where s is the current computing speed. This means that the probability that a fail-stop error would strike during an execution of length L at speed s is $p(\lambda^F(s), L)$ where we define $p(\lambda, L) = 1 - e^{-\lambda L}$.

To cope with silent errors, an additional verification mechanism is used. The time to verify (the output of) task T_i at speed s is $V_i(s)$. For the probability law followed by silent errors, we use an Exponential distribution of parameter $\lambda^F(s)$, where s is the current computing speed. As before, this means that the probability that a silent error would strike during an execution of length L at speed s is $p(\lambda^S(s), L)$. In the experimental evaluation (see Section 7), we investigate a wide range of values for $\lambda^F(s)$ and $\lambda^S(s)$.

Resilience has a cost in terms of energy consumption: the energy to checkpoint after task T_i is $E_i^{\mathrm{C}} = C_i(P_{idle} + P_{io})$, to recover from task T_i is $E_i^{\mathrm{R}} = R_i\ (P_{idle} + P_{io})$, and to verify task T_i at speed s is $E_i^{\mathrm{V}}(s) = V_i(s)(P_{idle} + P_{cpu}(s))$

3.4 Execution scenarios

We consider three different execution scenarios:

SingleSpeed A single speed s is used during the whole execution (hence $K = 1$).

ReExecSpeed There are two speeds, s for the first execution of each task, and σ for any potential re-execution (hence $K = 2$)

MultiSpeed We are given K discrete speeds, where K is arbitrary. The workflow chain is cut into subchains called segments and delimited by checkpoints. For each of these segments we can freely choose the speed of the first execution, and the (possibly different) speed for any ulterior execution, among the K speeds. Note that these speeds may well vary from one segment to another.

3.5 Optimization problems

For each execution scenario, we deal with four problems:
Time-VC Minimize total execution time (or makespan) using the VC-only approach
Time-VC+V Minimize total execution time (or makespan) using the VC+V approach
Energy-VC Minimize total energy consumption using the VC-only approach
Energy-VC+V Minimize total energy consumption using the VC+V approach

For the first two scenarios SingleSpeed and ReExecSpeed, we have to decide for the optimal location of the checkpoints (VC-only) and of the verifications (VC+V). For the last scenario MultiSpeed, we further have to select a pair of speeds (first execution and re-execution) for each segment.

4 SingleSpeed Scenario

In this scenario, we are given a single processing speed s. We investigate the VC-only and VC+V approaches, and for each of them we present an optimal polynomial-time dynamic programming algorithm.

4.1 VC-only: Using verified checkpoints only

For this approach, we only place verified checkpoints. We aimed at finding the best positions for checkpoints in order to minimize the total execution time (Time-VC) or the total energy consumption (Energy-VC).

4.1.1 Time-VC: Minimizing Makespan

Theorem 1. *For the* SingleSpeed *scenario, the* Time-VC *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^3)$.

Proof. We present a polynomial-time dynamic programming algorithm to compute the optimal execution time when we can only perform verified checkpoints. We define $Time_C^{rec}(j, k, s)$ as the optimal expectation of the time needed for successfully executing the tasks $T_1, \ldots, T_j$, where T_j has a verified checkpoint, and there are k additional verified checkpoints within tasks

$T_1, \ldots, T_{j-1}$. We always verify and checkpoint the last task T_n to save the result. Therefore, the goal is to compute

$$\min_{0 \le k < n} Time_C^{rec}(n, k, s)$$

To compute $Time_C^{rec}(j, k, s)$, we try all possible locations for the last checkpoint before T_j (see Figure 1):

$$Time_C^{rec}(j, k, s) = \min_{k \le i < j} \{Time_C^{rec}(i, k-1, s) + T_C(i+1, j, s)\}.$$

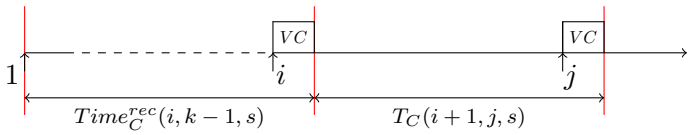


Figure 1: $Time_C^{rec}(j, k, s)$.

Here $T_C(i, j, s)$ is the expectation of the time needed for successfully executing all the tasks T_i to T_j, where T_{i-1} and T_j are both verified and checkpointed, while no other task in between is verified nor checkpointed. To ease notations, we assume that there is a special task T_0 that is always verified and checkpointed, with a recovery cost R_0. When computing $T_C(i, j, s)$, we can recover from T_{i-1} if a fail-stop error strikes before the completion of T_j or if a silent error is detected in the verification following T_j, and R_{i-1} is the time needed to recover from task T_{i-1}.

To initialize the recurrence, we define $Time_C^{rec}(j, 0, s) = T_C(1, j, s)$. Note that we never call $Time_C^{rec}(j, k, s)$ with $k \ge j$; that would not be feasible (not enough tasks compared to the number of checkpoints to be placed). We now show how to compute $T_C(i, j, s)$

In order to express $T_C(i, j, s)$, we start by considering *silent errors only* and use the notation $T_C^S(i, j, s)$ to that purpose. Silent errors can occur at any time during the computation but we can only detect them after all the tasks have been executed. Thus we always have to pay $T_{i,j}(s) + V_j(s)$, the time needed to execute tasks T_i to T_j at seeped s, and then verify T_j at speed s too. If the verification succeeds, which happens with probability $1 - p_{i,j}^S(s)$, we still have to checkpoint after T_j. Otherwise, with probability $p_{i,j}^S(s)$, a silent error has occurred, and we have to recover from T_{i-1} and start anew. We derive that

$$T_C^S(i, j, s) = T_{i,j}(s) + V_j(s) + p_{i,j}^S(s) \left(R_{i-1} + T_C^S(i, j, s)\right) + \left(1 - p_{i,j}^S(s)\right) C_j,$$

which simplifies into:

$$T_C^S(i, j, s) = e^{\lambda^S(s) T_{i,j}(s)} \left(T_{i,j}(s) + V_j(s)\right) + (e^{\lambda^S(s) T_{i,j}(s)} - 1) R_{i-1} + C_j.$$

When accounting for *fail-stop errors only*, things are different, because if a fail-stop error strikes during the execution of the tasks $T_1, \ldots, T_j$ then the application stops immediately,

in the middle of the computation. Let $T_{lost_{i,j}}(s)$ be the expectation of the time lost when a fail-stop error strikes:

$$T_{lost_{i,j}}(s) = \int_0^{\infty} x\mathbb{P}(X = x|X < T_{i,j}(s))dx = \frac{1}{\mathbb{P}(X < T_{i,j}(s))} \int_0^{T_{i,j}(s)} x\lambda^F(s)e^{-\lambda^F(s)x}dx$$

and $\mathbb{P}(X < T_{i,j}(s)) = 1 - e^{-\lambda^F(s)T_{i,j}(s)}$. Integrating by parts, we derive that

$$T_{lost_{i,j}}(s) = \frac{1}{\lambda^F(s)} - \frac{T_{i,j}(s)}{e^{\lambda^F(s)T_{i,j}(s)} - 1}.$$

Therefore, using the notation $T_C^F(i,j,s)$ when restricting to fail-stop errors, we have:

$$T_C^F(i,j,s) = p_{i,j}^F(s)\left(T_{lost_{i,j}}(s) + R_{i-1} + T_C^F(i,j,s)\right) + \left(1 - p_{i,j}^F(s)\right)\left(T_{i,j}(s) + C_j\right),$$

which simplifies to:

$$T_C^F(i,j,s) = (e^{\lambda^F(s)T_{i,j}(s)} - 1)\left(\frac{1}{\lambda^F(s)} + R_{i-1}\right) + C_j.$$

We now account for *both fail-stop and silent errors*, and use the notation $T_C^{SF}(i,j,s)$ to that purpose. We look at fail-stop errors first. If the application stops, then we do not perform the verification and do not need to check for silent errors since we must do a recovery. If no fail-stop error stroke during the execution, then we can do the verification and check for silent errors. Therefore,

$$\begin{aligned} T_C^{SF}(i,j,s) &= p_{i,j}^F(s)\left(T_{lost_{i,j}}(s) + R_{i-1} + T_C^{SF}(i,j,s)\right) \\ &\quad + \left(1 - p_{i,j}^F(s)\right)\left(T_{i,j}(s) + V_j(s) + p_{i,j}^S(s)\left(R_{i-1} + T_C^{SF}(i,j,s)\right) + \left(1 - p_{i,j}^S(s)\right)C_j\right). \end{aligned}$$

When plugging $p_{i,j}^F$, $p_{i,j}^S$ and $T_{lost_{i,j}}$ back into the equation, this simplifies to

$$T_C^{SF}(i,j,s) = e^{\lambda^S(s)T_{i,j}(s)}\left(\frac{e^{\lambda^F(s)T_{i,j}(s)} - 1}{\lambda^F(s)} + V_j(s)\right) + \left(e^{(\lambda^F(s)+\lambda^S(s))W_{i,j}} - 1\right)R_{i-1} + C_j.$$

Finally, we have computed $T_C(i,j,s) = T_C^{SF}(i,j,s)$, which completes the computation of $Time_C^{rec}(j,k,s)$. The complexity is $O(n^3)$ because there are three nested loops in the algorithm, and this concludes the proof. □

Theorem 1 nicely extends the result of Toueg and Babaoglu [31] to a linear chain of tasks subject to both fail-stop and silent errors. For the sake of comparing with the case of a divisible load application, we extend Young/Daly's formula as follows:

Proposition 1. *For a divisible load application subject to both fail-stop errors and silent errors, a first-order approximation of the optimal checkpointing period is*

$$T_{opt}(s) = \sqrt{\frac{2(V + C)}{\lambda^F(s) + 2\lambda^S(s)}} \tag{4}$$

where C is the time to checkpoint, $\lambda^F(s)$ is the rate of fail-stop errors and $\lambda^S(s)$ is the rate of silent errors.

Proof. In the presence of fail-stop errors only, Young/Daly's formula $T_{opt}(s) = \sqrt{\frac{2C}{\lambda^F(s)}}$ is obtained by computing the *waste*, which is the fraction of time where the platform is not doing useful work. Let T be the checkpointing period. The waste Waste comes from two sources: the waste Waste_{ef} due to the resilience method itself in an error-free execution, and the waste Waste_{fail} due to failures. Because there is a checkpoint of length C every T time-units, we have $\text{Waste}_{ef} = \frac{C}{T}$. Fail-stop errors strike every $\frac{1}{\lambda^F(s)}$ time-units in average, and the expected time-lost is $R + \frac{T}{2}$, where R is the time for recovery (this is because the average time-lost is half the period $\frac{T}{2}$), which gives $\text{Waste}_{fail} = \lambda^F(s)(R + \frac{T}{2}$. When $\frac{1}{\lambda^F(s)}$ is large in front of the resilience parameters C and R, the two sources of waste add up (as a first-order approximation), hence

$$\text{Waste} = \text{Waste}_{ef} + \text{Waste}_{fail} = \frac{C}{T} + \lambda^F(s)(R + \frac{T}{2}).$$

Differentiating, we find that $T_{opt}(s) = \sqrt{\frac{2C}{\lambda^F(s)}}$ minimizes the waste Waste, which is Young/Daly's formula.

Now with both fail-stop and silent errors, the error-free waste becomes $\text{Waste}_{ef} = \frac{V+C}{T}$, because we lose $V + C$ time-units every period of length T. We waste due to failures becomes $\text{Waste}_{fail} = \lambda^F(s)(R + \frac{T}{2}) + \lambda^S(s)(R + T)$. Note that the entire period is always lost when a silent error is detected at the end of the period, while a fail-stop period leads to losing half the period in average. Finally,

$$\text{Waste} = \text{Waste}_{ef} + \text{Waste}_{fail} = \frac{V + C}{T} + \lambda^F(s)(R + \frac{T}{2}) + \lambda^S(s)(R + T).$$

Differentiating, we find that $T_{opt}(s) = \sqrt{\frac{2(V+C)}{\lambda^F(s)+2\lambda^S(s)}}$ minimizes the waste Waste. Again, we stress that this result is a first-order approximation, which is valid only if all resilience parameters C,R and V are small in front of both MTBF values, namely $1/\lambda^F(s)$ for fail-stop errors and and $1/\lambda^S(s)$ for silent errors. □

We use this formula as a lower bound on the overhead of the VC-ONLY algorithm in Section 7 (see Figure 5). With a chain of tasks, we have less flexibility for checkpointing than with a divisible load application, and it is interesting to numerically evaluate the difference due to the application framework.

4.1.2 Energy-VC: Minimizing Energy

Proposition 2. *For the* SINGLESPEED *scenario, the* ENERGY-VC *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^3)$.

Proof. The problem is very similar to makespan minimization (Section 4.1.1). We replace $Time_C^{rec}$ with $Energy_C^{rec}(j, k, s)$, which is the optimal expectation of the total energy needed for successfully executing tasks T_1 to T_j with k verified checkpoints in between. Instead of T_C^{SF} we use $E_C^{SF}(i, j, s)$, which is the total energy needed for successfully executing all the tasks from T_i to T_j with no checkpoint and no verification in between. Note that T_j is checkpointed and verified. The goal is to compute:

$$\min_{0 \le k < n} Energy_C^{rec}(n, k, s)$$

where $Energy_C^{rec}(j,k)$, $Energy_C^{rec}(i,j,k_v,s)$ can be expressed as follows:

$$Energy_{VC}^{rec}(j,k,s) = \min_{k \le i < j} \left\{ Energy_C^{rec}(i,k-1,s) + E_C^{SF}(i+1,j,s) \right\}.$$

Previously we used $T_{lost_{i,j}}(s)$ to compute the time lost when fail-stop errors occur, now we use $E_{lost_{i,j}}(s) = T_{lost_{i,j}}(s)\,(P_{cpu}(s) + P_{idle})$, which is the expectation of the energy lost executing the tasks from T_i to T_j depending on when the fail-stop error stroke. Therefore we can express $E_C^{SF}(i,j,s)$ as follows:

$$\begin{aligned} E_C^{SF}(i,j,s) &= p_{i,j}^F(s)\left(E_{lost_{i,j}}(s) + E_{i-1}^{\mathrm{R}} + E_C^{SF}(i,j,s)\right) \\ &\quad + \left(1 - p_{i,j}^F(s)\right)\left(E_{i,j}(s) + V_j(s) + p_{i,j}^S(s)\left(E_{i-1}^{\mathrm{R}} + E_C^{SF}(i,j,s)\right) + \left(1 - p_{i,j}^S(s)\right)E_j^{\mathrm{C}}\right). \end{aligned}$$

which simplifies to:

$$\begin{aligned} E_C^{SF}(i,j,s) &= \left(P_{idle} + P_{cpu}(s)\right) e^{\lambda^S(s)T_{i,j}(s)} \left(\frac{e^{\lambda^F(s)T_{i,j}(s)} - 1}{\lambda^F(s)} + V_j(s) \right) \\ &\quad + \left(e^{(\lambda^F(s)+\lambda^S(s))T_{i,j}} - 1 \right) E_{i-1}^{\mathrm{R}} + E_j^{\mathrm{C}}. \end{aligned}$$

Clearly, the complexity is the same as for makepsan minimization (Theorem 1). □

4.2 VC+V: Using verified checkpoints and single verifications

For the VC+V approach, we can add extra verifications between two checkpoints. Basically, this allows us to detect an error without having to wait for the next checkpoint. The sooner the error is detected, the sooner we can recover and the lesser the re-execution time. We aim at finding the best positions for checkpoints and verifications in order to minimize the total execution time (Time-VC+V) or the total energy consumption (Energy-VC+V). For both objectives, adding extra verifications between two checkpoints adds an extra step in the algorithm, which results in a higher complexity.

4.2.1 Time-VC+V: Minimizing Makespan

Theorem 2. *For the* SingleSpeed *scenario, the* Time-VC+V *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^5)$.

Proof. In Section 4.1, we were only placing verified checkpoints. Here we add verifications that are not associated with a checkpoint. The main idea is to replace the expression T_C by another recursive expression, $Time_V^{rec}(i,j,k_v,s)$, which returns the optimal expectation of the time needed for successfully executing the tasks (and verifications) from T_i to T_j, where T_{i-1} has a verified checkpoint, and there are k_v additional verifications within tasks $T_i, \ldots, T_{j-1}$.

Instead of $Time_C^{rec}(j,k,s)$, we now express $Time_{VC}^{rec}(j,k,s)$ (see Figure 2), which is the optimal expectation of the time needed for successfully executing the first j tasks with k verified checkpoints and other single verifications. The goal is to compute:

$$\min_{0 \le k < n} Time_{VC}^{rec}(n,k,s),$$

where $Time_{VC}^{rec}(j,k,s)$ can be expressed as follows:

$$Time_{VC}^{rec}(j,k,s) = \min_{k \le i < j} \left\{ Time_{VC}^{rec}(i,k-1,s) + \min_{0 \le k_v < j-i} \{Time_{V}^{rec}(i+1,j,k_v,s)\} + C_j \right\}.$$

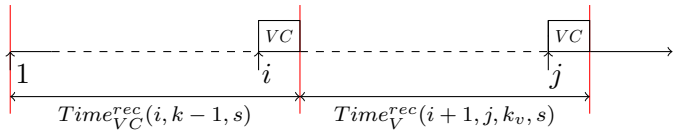


Figure 2: $Time_{VC}^{rec}(j,k,s)$.

Indeed, we try all possible values of k_v, the number of single verifications between T_{i+1} and T_{j-1}, and we account for the checkpointing time C_j, which is not included in the computation of $Time_{V}^{rec}$.

We initialize the recurrence with $Time_{VC}^{rec}(j,0,s) = \min_{0 \le k_v < j} \{Time_{V}^{rec}(1,j,k_v,s) + C_j\}$. Indeed, there are no checkpoints, and we only decide if some extra verifications should be placed. Note that we always have $j > k$ when calling $Time_{VC}^{rec}(j,k,s)$, therefore there are always enough tasks compared to the number of checkpoints to be placed.

We are now ready to express $Time_{V}^{rec}(i,j,k_v,s)$ (see Figure 3):

$$Time_{V}^{rec}(i,j,k_v,s) = \min_{i+k_v-1 \le l < j} \{Time_{V}^{rec}(i,l,k_v-1,s) + T_V(l+1,j,i-1,k_v-1,s)\}, \quad (5)$$

where $T_V(i,j,l_c,k_v,s)$ is the expectation of the time needed for successfully executing all the non-verified and non-checkpointed tasks from i to j, knowing that if an error is detected after T_j, we can recover from T_{l_c}, the task followed by the last checkpoint, with a cost R_{l_c}. We keep also k_v as a parameter; it is the number of single verifications between tasks T_{l_c+1} and T_{i-2}. Indeed, when computing $T_V(i,j,l_c,k_v,s)$, in case of failure, we need to recompute $Time_{V}^{rec}(l_c+1,i-1,k_v,s)$. Note that in Equation (5), $T_V(l+1,j,i-1,k_v-1,s)$ reuses the value $Time_{V}^{rec}(i,l,k_v-1,s)$, hence $Time_{V}^{rec}(i,j,k_v,s)$ can be expressed as a function of $Time_{V}^{rec}(i,l,k_v-1,s)$ only, with $l < j$.

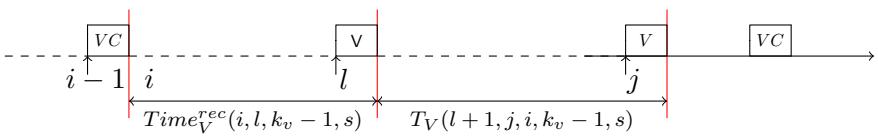


Figure 3: $Time_{V}^{rec}(i,j,k_v,s)$.

We initialize the recurrence with $Time_{V}^{rec}(i,j,0,s) = T_V(i,j,i-1,0,s)$ if $j \ge i$ (i.e., there are no verifications between T_i and T_j, and the last checkpoint is after T_{i-1}), and $Time_{V}^{rec}(i,j,k_v,s) = 0$ if $j < i$ (there is no task to execute in this case).

We now explain how to compute T_V. With *silent errors only* (notation T_V^S), we always execute tasks T_i to T_j and we pay the verification cost for task T_j, and in case of failure, we recover from T_{l_c} and redo the work, knowing that there is a verification after task T_{i-1}, and there are k_v other single verifications between tasks T_{l_c+1} and T_{i-2}:

$$T_V^S(i,j,l_c,k_v,s) = T_{i,j}(s) + V_j(s) + p_{i,j}^S(s)\left(R_{l_c} + Time_V^{rec}(l_c+1,i-1,k_v,s) + T_V^S(i,j,l_c,k_v,s)\right),$$

which simplifies to:

$$T_V^S(i,j,l_c,k_v,s) = e^{\lambda^S(s)T_{i,j}(s)}\left(T_{i,j}(s) + V_j(s)\right) + \left(e^{\lambda^S(s)T_{i,j}(s)} - 1\right)\left(R_{l_c} + Time_V^{rec}(l_c+1,i-1,k_v,s)\right).$$

Note that in the initial case, $T_V(i,j,i-1,0,s)$ calls $Time_V^{rec}(i,i-1,k_v,s)$, whose cost is 0, hence the recurrence is properly initialized.

When accounting for *fail-stop errors only* (notation T_V^F), we do not perform any verification, and therefore we can simply reuse the results from Section 4.1.1 to compute T_V^F, because there are no isolated verifications.

Finally, when accounting for *both silent and fail-stop errors* (notation $T_V = T_V^{SF}$), we use the same method as in the previous section, using $T_{lost_{i,j}}$. If a fail-stop error strikes between two verifications, we must do a recovery, otherwise we check for silent errors:

$$\begin{aligned}T_V^{SF}(i,j,l_c,k_v,s) &= p_{i,j}^F(s)\left(T_{lost_{i,j}}(s) + R_{l_c} + Time_V^{rec}(l_c+1,i-1,k_v,s) + T_V^{SF}(i,j,l_c,k_v,s)\right)\\ &+(1-p_{i,j}^F(s))\left(T_{i,j}(s) + V_j(s) + p_{i,j}^S(s)\left(R_{l_c} + Time_V^{rec}(l_c+1,i-1,k_v,s) + T_V^{SF}(i,j,l_c,k_v,s)\right)\right).\end{aligned}$$

When plugging $p_{i,j}^F$, $p_{i,j}^S$ and $T_{lost_{i,j}}$ into the equation, this simplifies to:

$$\begin{aligned}T_V^{SF}(i,j,l_c,k_v,s) &= e^{\lambda^S(s)T_{i,j}(s)}\left(\frac{e^{\lambda^F(s)T_{i,j}(s)} - 1}{\lambda^F(s)} + V_j(s)\right)\\ &+ \left(e^{(\lambda^F(s)+\lambda^S(s))T_{i,j}(s)} - 1\right)\left(R_{l_c} + Time_V^{rec}(l_c+1,i-1,k_v,s)\right).\end{aligned}$$

Finally, there are five nested loops in the algorithm, hence the complexity is $O(n^5)$. □

4.3 Energy-VC+V: Minimizing Energy

Proposition 3. *For the* SingleSpeed *scenario, the* Energy-VC+V *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^5)$.

Proof. The problem is very similar to the minimization of the maskepan for the VC+V approach (see Section 4.2.1). We replace $Time_{VC}^{rec}$ with $Energy_{VC}^{rec}(j,k,s)$, which is the optimal expectation of the energy needed for successfully executing the tasks T_1 to T_j with k verified checkpoints in between. Instead of $Time_V^{rec}$ we use $Energy_V^{rec}(i,j,k_v,s)$ and we replace T_V^{SF} with $E_V^{SF}(i,j,l_c,k_v,s)$. The goal is to compute:

$$\min_{0\le k<n} Energy_{VC}^{rec}(n,k,s)$$

where $Energy_{VC}^{rec}(j,k)$ and $Energy_V^{rec}(i,j,k_v,s)$ can be expressed as follows:

$$Energy_{VC}^{rec}(j,k) = \min_{k\ge i<j}\left\{Energy_{VC}^{rec}(i,k-1,s) + \min_{0\le k_v<j-i}\left\{Energy_V^{rec}(i+1,j,k_v,s) + E_j^{\mathrm{C}}\right\}\right\},$$

$$Energy_V^{rec}(i,j,k_v,s) = \min_{i+k_v-1 \le l < j} \left\{ Energy_V^{rec}(i,l,k_v-1,s) + E_V^{SF}(l+1,j,i,k_v-1,s) \right\}$$

$E_V^{SF}(i,j,l_c,k_v,s)$ is very similar in its construction to $T_V^{SF}(i,j,l_c,k_v,s)$. E_V^{SF} is used to compute the total energy consumption for executing the tasks T_i to T_j knowing that the last checkpoint is at T_{l_c}. In case of failure, and as for T_V^{SF}, we have to call both $Energy_V^{rec}$ and E_V^{SF} after the recovery to re-execute all the tasks that have been lost. This expression can be expressed as follows:

$$\begin{aligned} E_V^{SF}(i,j,l_c,k_v,s) &= p_{i,j}^F(s) \left(E_{lost_{i,j}}(s) + E_{l_c}^{\mathrm{R}} + Energy_V^{rec}(l_c,i-1,k_v,s) + E_V^{SF}(i,j,l_c,k_v,s) \right) \\ &+ (1-p_{i,j}^F(s)) \left(E_{i,j}(s) + E_j^{\mathrm{V}}(s) + p_{i,j}^S(s) \left(E_{l_c}^{\mathrm{R}} + Energy_V^{rec}(l_c,i-1,k_v,s) + E_V^{SF}(i,j,l_c,k_v,s) \right) \right) \end{aligned}$$

Which simplifies to:

$$\begin{aligned} E_V^{SF}(i,j,l_c,k_v,s) &= (P_{idle} + P_{cpu}(s))\, e^{\lambda^S(s) T_{i,j}(s)} \left(\frac{e^{\lambda^F T_{i,j}(s)} - 1}{\lambda^F} + V_j(s) \right) \\ &+ \left(e^{(\lambda^F + \lambda^S(s)) T_{i,j}(s)} - 1 \right) \left(E_{l_c}^{\mathrm{R}} + Energy_V^{rec}(l_c,i-1,k_v,s) \right) \end{aligned}$$

Clearly, the complexity is the same as for makepsan minimization (Theorem 2). □

5 ReExecSpeed Scenario

In the ReExecSpeed scenario, we are given two CPU speeds s and σ. s is the regular speed and σ is a higher speed (possibly obtained by overclocking). The regular speed s is used for the first execution of the tasks, while σ is used to speed up any potential re-executions. We always account for both *silent and fail-stop errors*.

In this scenario, and for both the VC-only and VC+V approaches, we end up solving two independent problems. The first problem is to minimize the makespan (or total energy consumption) for the first execution of the tasks with the first speed, until the first error is encountered. It means that once the first error strikes, we recover from the last checkpoint and we start re-executing the tasks with the second speed until we reach the next checkpoint. This is the second problem: minimizing the time (or energy) spent re-executing the tasks with the second speed. Eventually, this amounts to solving the problem for the SingleSpeed scenario, but at speed σ.

5.1 VC-only: Using verified checkpoints only

Following the VC-only approach, we aim at finding the best positions for verified checkpoints in order to minimize the makespan (Time-VC) or the total energy consumption (Energy-VC).

5.1.1 Time-VC: Minimizing Makespan

Theorem 3. *For the* ReExecSpeed *scenario, the* Time-VC *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^3)$.

Proof. The approach is similar to that of Section 4.1.1. First, we replace $Time_C^{rec}$ with $Time_{C_{re}}^{rec}(j,k,s,\sigma)$, which now accounts for the two speeds. We introduce $T_{C_{re}}(i,j,s,\sigma)$ as the expectation of the time needed for successfully executing all the tasks from T_i to T_j, where both T_{i-1} and T_j are checkpointed. Note that the cost of the checkpoint after T_j is included in $Time_{C_{re}}^{rec}$. The goal is to compute:

$$\min_{0\le k<n} Time_{C_{re}}^{rec}(n,k,s,\sigma),$$

where $Time_{C_{re}}^{rec}(j,k,s,\sigma)$ can be expressed as follows:

$$Time_{C_{re}}^{rec}(j,k,s,\sigma) = \min_{k\le i<j} \left\{ Time_{C_{re}}^{rec}(i,k-1,s,\sigma) + T_{C_{re}}(i+1,j,s,\sigma) + C_j \right\}.$$

We initialize the recurrence with $Time_{C_{re}}^{rec}(j,0,s,\sigma) = T_{C_{re}}(i+1,j,s,\sigma) + C_j$. To compute the time needed for successfully executing the tasks between two checkpoints, we need to make the difference between the first execution (before the first error) and all the potential re-executions (after at least one error). It is done in two steps denoted by $T_{C_{first}}^{SF}(i,j,s)$ and $T_C^{SF}(i,j,\sigma)$. $T_{C_{first}}^{SF}(i,j,s)$ is the expectation of the time spent executing the tasks T_i to T_j the very first time, before the first error is encountered, while T_C^{SF} is the expectation of the time needed for successfully executing all the tasks T_i to T_j, as in the previous section (see Section 4.1.1), but using speed σ. Note that both T_{i-1} and T_j are checkpointed.

Let $p_{i,j}^{\mathrm{Err}}(s)$ be the probability that at least one error is detected during the execution of the tasks from T_i to T_j at a speed s. We account for both *silent and fail-stop errors* and we can only detect silent errors if no fail stop error has occurred, thus $p_{i,j}^{\mathrm{Err}}(s) = p_{i,j}^F(s) + (1-p_{i,j}^F(s))p_{i,j}^S(s)$. If no error strikes, then the time to compute the tasks from T_i to T_j is exactly $T_{C_{first}}^{SF}$ and $p_{i,j}^{\mathrm{Err}}(s) = 0$, which means that all the tasks have been executed successfully with the first speed. If at least one error occurs, then $T_{C_{first}}^{SF}$ is the time lost trying to execute the tasks with the first speed and we need to recover from the last checkpoint with a probability $p_{i,j}^{\mathrm{Err}}(s) > 0$. Then we use $T_C^{SF}(i,j,\sigma)$ to re-execute all the tasks from T_i to T_j with the second speed until we pass the next checkpoint. Therefore,

$$T_{C_{re}}(i,j,s,\sigma) = T_{C_{first}}^{SF}(i,j,s) + p_{i,j}^{\mathrm{Err}}(s)\left(R_{i-1} + T_C^{SF}(i,j,\sigma)\right), \tag{6}$$

$T_{C_{first}}^{SF}(i,j,s)$ returns the expectation of the time needed to execute the tasks T_i to T_j, where T_{i-1} and T_j are verified and checkpointed. During the execution of the tasks T_i to T_j there are two possible scenarios: either a fail-stop error has occurred and we have to compute $T_{lost_{i,j}}$, or there was no fail-stop error and we have to check wether a silent error has occurred or not. Recall that we do not account for the re-executions, as this is already handled by T_C^{SF} separately (and with the second speed). Therefore,

$$T_{C_{first}}^{SF}(i,j,s) = (1-p_{i,j}^F(s))T_{i,j}(s) + p_{i,j}^F(s)\left(\frac{1}{\lambda^F(s)} - \frac{T_{i,j}}{(e^{\lambda^F(s)T_{i,j}}-1)}\right).$$

Despite the two steps needed to compute $T_{C_{re}}$, the complexity is the same as for the SingleSpeed scenario (Theorem 2). There are three nested loops in the algorithm, hence the complexity is $O(n^3)$, and this concludes the proof. □

5.1.2 Energy-VC: Minimizing Energy

Proposition 4. *For the* ReExecSpeed *scenario, the* Energy-VC *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^3)$.

Proof. Minimizing the energy for the VC-only approach is done the same way as minimizing the makespan (see Section 5.1.1). We replace $Time^{rec}_{VC_{re}}(n,k,s,\sigma)$ with $Energy^{rec}_{VC_{re}}(n,k,s,\sigma)$ which accounts for the both speeds. The goal is to compute:

$$\min_{0\le k<n} Energy^{rec}_{C_{re}}(n,k,s,\sigma),$$

where $Energy^{rec}_{C_{re}}(j,k,s,\sigma)$ can be expressed as follows:

$$Energy^{rec}_{C_{re}}(j,k,s,\sigma) = \min_{k\le i<j} \left\{ Energy^{rec}_{C_{re}}(i,k-1,s,\sigma) + E_{C_{re}}(i+1,j,s,\sigma) + E^{\mathrm{C}}_j \right\}.$$

Similarly to $T_{C_{re}}$ we introduce $E_{C_{re}}$, which is also computed in two steps, one for the first execution with the first speed and a second one for all potential re-executions with the second speed. Note that as $T_{C_{re}}$, $E_{C_{re}}$ doesn't include any recovery operation since we only account for the energy spent executing the task until the first is encountered. As a result, the associated energy is not related to P_{io} and is directly proportional to the time with respect to s and $P_{cpu}(s)$. The time spent executing the tasks during the first execution is already computed by $T^{SF}_{C_{first}}$ (see Section 5.1.1). We can express $E_{C_{re}}(i,j,s,\sigma)$ as follows:

$$E_{C_{re}}(i,j,s,\sigma) = (P_{idle} + P_{cpu}(s))\, T^{SF}_{C_{first}}(i,j,s) + p^{\mathrm{Err}}_{i,j}(s)\left(E^{\mathrm{R}}_{i-1} + E^{SF}_{C}(i,j,\sigma)\right),$$

Clearly, the complexity is the same as for makepsan minimization (Theorem 3). □

5.2 VC+V: Using verified checkpoints and single verifications

Following the VC+V approach, we solve the Time-VC+V and Energy-VC+V problem with a another polynomial-time dynamic programming algorithm, whose complexity remains he same as for the SingleSpeed scenario.

5.2.1 Time-VC+V: Minimizing Makespan

Theorem 4. *For the* ReExecSpeed *scenario, the* Time-VC+V *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^5)$.

Proof. The VC+V approach follows the same reasoning as the VC-only approach (see Section 5.1.1). We replace $Time^{rec}_{C_{re}}$ with $Time^{rec}_{VC_{re}}(j,k,s,\sigma)$, which accounts for both speeds. Instead of $T_{C_{re}}(i,j,s,\sigma)$ we use $T_{VC_{re}}(i,j,s,\sigma)$. Note that $T_{VC_{re}}$ uses the new re-execution model but and it also includes the cost of additional verifications between T_i and T_{j-1}. Again, the cost of the checkpoint after T_j is included in $Time^{rec}_{VC_{re}}$. The goal is to compute:

$$\min_{0\le k<n} Time^{rec}_{VC_{re}}(n,k,s,\sigma),$$

where $Time^{rec}_{VC_{re}}(j,k,s,\sigma)$ can be expressed as follows:

$$Time^{rec}_{VC_{re}}(j,k,s,\sigma) = \min_{k\le i<j} \left\{ Time^{rec}_{VC_{re}}(i,k-1,s,\sigma) + T_{VC_{re}}(i+1,j,s,\sigma) + C_j \right\}.$$

We initialize the recurrence with $Time^{rec}_{VC_{re}}(j,0,s,\sigma) = T_{VC_{re}}(1,j,s,\sigma) + C_j$. As before, $T_{VC_{re}}$ is computed in two steps denoted by $T_{VC_{first}}(i,j,s)$ and $T_{VC_{sec}}(i,j,\sigma)$. $T_{VC_{first}}(i,j,s)$ is the expectation of the time spent executing the tasks T_i to T_j the very first time, before the first error is encountered, and $T_{VC_{sec}}$ is the expectation of the time needed for successfully executing all the tasks T_i to T_j the same way it was done in the SingleSpeed scenario. Note that both T_{i-1} and T_j are checkpointed, and that while $T_{VC_{first}}$ uses s for the first execution, $T_{VC_{sec}}$ uses σ for all the potential re-executions. Both of them also include the cost of additional verifications between T_i and T_{j-1}, and as a result we get two sets of optimal places for verifications (one for each speed).

If no error strikes, then the time to compute the tasks from T_i to T_j is exactly $T_{VC_{first}}$ and $p^{\text{Err}}_{i,j}(s) = 0$, which means that all the tasks have been executed successfully with the first speed. If at least one error occurs, then $T_{VC_{first}}$ is the time lost trying to execute the tasks with the first speed and we need to recover from the last checkpoint with a probability $p^{\text{Err}}_{i,j}(s) > 0$. Then we use $T_{VC_{sec}}(i,j,\sigma)$ to re-execute all the tasks from T_i to T_j with the second speed until we pass the next checkpoint, which amounts to use $Time^{rec}_V$ from the SingleSpeed model with the second speed σ (see Equation (5)). Therefore,

$$T_{VC_{re}}(i,j,s,\sigma) = T_{VC_{first}}(i,j,s) + p^{\text{Err}}_{i,j}(s)\left(R_{i-1} + T_{VC_{sec}}(i,j,\sigma)\right), \tag{7}$$

where $T_{VC_{first}}(i,j,s)$ and $T_{VC_{sec}}(i,j,\sigma)$ can be expressed as follows:

$$T_{VC_{first}}(i,j,s) = \min_{0\le k_v<j-i}\left\{Time^{rec}_{V_{first}}(i,j,k_v,s)\right\}.$$
$$T_{VC_{sec}}(i,j,\sigma) = \min_{0\le k_v<j-i}\left\{Time^{rec}_{V}(i,j,k_v,\sigma)\right\}.$$

$Time^{rec}_{V_{first}}(i,j,k_v,s)$ and $T^{SF}_{V_{first}}(i,j,s)$ are similar to $Time^{rec}_V$ and T^{SF}_V, but they only account for the first execution of the tasks from T_i to T_j, and therefore they do not account for recovery operations nor re-executions. $Time^{rec}_{V_{first}}$ finds the best positions for the verifications between T_i and T_{j-1} in order to minimize the expectation of the time spent executing the tasks T_i to T_j at speed s until the first error strikes. $T^{SF}_{V_{first}}(i,j,s)$ computes the expectation of the time to execute the tasks T_i to T_j, where T_{i-1} and T_j are verified. We have to make sure that no error has happened before T_i, because otherwise we have to recover and re-execute the tasks with the second speed, which is done by $T_{VC_{sec}}$. Therefore,

$$Time^{rec}_{V_{first}}(i,j,k_v,s) = \min_{i+k_v-1\le l<j}\left\{Time^{rec}_{V_{first}}(i,l,k_v-1,s) + (1-p^{\text{Err}}_{i,l})(T^{SF}_{V_{first}}(l+1,j,i,s))\right\}.$$

$T^{SF}_{V_{first}}(i,j,s)$ returns the expectation of the time needed to execute the tasks T_i to T_j, where T_{i-1} and T_j are verified. During the execution of the tasks T_i to T_j there are two possible scenarios: either a fail-stop error has occurred and we have to compute $T_{lost_{i,j}}$, or there was no fail-stop error and we have to check wether a silent error has occurred or not. Recall that we do not account for the re-executions, as this is already handled by $T_{VC_{sec}}$ separately (and with the second speed). Therefore,

$$T^{SF}_{V_{first}}(i,j,s) = (1-p^F_{i,j}(s))\left(T_{i,j}(s) + V_j(s)\right) + p^F_{i,j}(s)\left(\frac{1}{\lambda^F(s)} - \frac{W_{i,j}}{(e^{\lambda^F(s)W_{i,j}}-1)}\right).$$

We initialize $Time^{rec}_{V_{first}}$ with $Time^{rec}_{V_{first}}(i,j,0,s) = (1-p^{\text{Err}}_{i,l})(T^{SF}_{V_{first}}(i,j,i,s))$.

Desptite the two steps needed to compute the expected makespan between two checkpoints, the complexity is the same as for the SingleSpeed model (Theorem 2). There are five nested loop in the algorithm, hence the complexity is $O(n^5)$. □

5.2.2 Energy-VC+V: Minimizing Energy

Proposition 5. *For the* ReExecSpeed *scenario, the* Energy-VC+V *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^5)$.

Proof. Again, the VC+V approach for this scenario is very similar to the makespan minimization (see Section 5.2.1). We replace $Time^{rec}_{VC_{re}}(n,k,s,\sigma)$ with $Energy^{rec}_{VC_{re}}(n,k,s,\sigma)$ which accounts for the two speeds. Instead of $T_{VC_{re}}(i,j,s,\sigma)$, we use $E_{VC}(i,j,s,\sigma)$ which returns the optimal exptectation of the energy consumed between two checkpoints (both T_i and T_j are checkpointed). E_{VC} also accounts for the re-execution model the same way it was done for the makespan. The goal is to compute:

$$\min_{0\le k<n} Energy^{rec}_{VC_{re}}(n,k,s,\sigma),$$

where, $Energy^{rec}_{VC_{re}}(j,k,s,\sigma)$ can b expressed as follows:

$$Energy^{rec}_{VC_{re}}(j,k,s,\sigma) = \min_{k\le i<j} \left\{ Energy^{rec}_{VC_{re}}(i,k-1,s,\sigma) + E_{VC}(i+1,j,s,\sigma) + E^{\mathrm{C}}_j \right\}.$$

We express E_{VC} as we did for the minimization of the makespan (see Equation (7)), by breaking E_{VC} in two steps $E_{VC_{first}}$ and $E_{VC_{sec}}$. The first step is to compute the energy consumed when executing the tasks for the very first time, before the first error is encountered. It is done by $E_{VC_{first}}$ similarly to $T_{VC_{first}}$. It uses the first speed s and it also includes the cost of extra verifications between T_i and T_j. In case at least one error occurs during the first step we have to rollback to the last checkpoint at T_{i-1} and we have to re-execute the tasks T_i to T_j. This is done by $E_{VC_{sec}}(i,j,\sigma)$, which minimizes the energy consumed between T_i and T_j using the second speed σ, similarly to $T_{VC_{sec}}$. We express $E_{VC}(i,j,s,\sigma)$ as follows:

$$E_{VC}(i,j,s,\sigma) = E_{VC_{first}}(i,j,s) + p^{\mathrm{Err}}_{i,j}(s)\left(E^{\mathrm{R}}_{i-1} + E_{VC_{sec}}(i,j,\sigma)\right)$$

$E_{VC_{first}}(i,j,s)$ and $E_{VC_{sec}}(i,j,\sigma)$ are very similar to $T_{VC_{first}}$ and $T_{VC_{sec}}$. Recall that $T_{VC_{first}}$ returns the optimal expected makespan for executing the tasks the very first time, before the first error. It only uses verifications in order to detect the first error and it doesn't include checkpoints, nor recoveries. Therefore, the associated energy is not related to P_{io}. As a result, the expected energy consumed during the first step is directly proportional to the expected time with respect to s and $P_{cpu}(s)$. We can express $E_{VC_{first}}$ and $E_{VC_{sec}}$ as follows:

$$E_{VC_{first}}(i,j,s) = (P_{idle} + P_{cpu}(s))T_{VC_{first}}(i,j,s)$$
$$E_{VC_{sec}}(i,j,\sigma) = \min_{0\le k_v<j-i} \{Energy^{rec}_V(i,j,k_v,\sigma)\}$$

Clearly, the complexity is the same as for makepsan minimization (Theorem 4). □

6 MultiSpeed Scenario

In this section, we investigate the most flexible model, MultiSpeed, to get even more control over the execution time and the energy consumption, but at the cost of a higher complexity. This model is built over the ReExecSpeed model. Instead of having two fixed speeds, we are given a set S of K discrete speeds. We call a segment of the chain a group of tasks between two checkpoints, and we allow each of these segments to use one speed for the first execution, and a second speed for all potential re-executions, where both speeds belong to S. Each of these two speeds can be different for each segment.

6.1 VC-only: Using verified checkpoints only

Following the VC-ONLY approach, we aim at finding the best positions for the checkpoints, as well as the best speeds for each segment, in order to minimize the expected makespan (TIME-VC) and the total energy consumption (ENERGY-VC). This section is based on the work done for the REEXECSPEED model under the VC-ONLY approach.

6.1.1 Time-VC: Minimizing Makespan

Theorem 5. *For the* MULTISPEED *scenario, the* TIME-VC *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^3 + n^2K^2)$.

Proof. Recall that $T_{C_{re}}$ from the REEXECSPEED model already accounts for the two speeds (see Equation (6)). We try all the possible pairs of speeds for each segment, computed by T_C^{SF}. The goal is to compute:

$$\min_{0 \le k < n} Time^{rec}_{C_{mul}}(n, k),$$

where

$$Time^{rec}_{C_{mul}}(j, k) = \min_{k \le i < j} \left\{ Time^{rec}_{C_{mul}}(i, k-1) + \min_{s \in S, \sigma \in S} \left\{ T_C^{SF}(i, j, s, \sigma) \right\} + C_j \right\}.$$

To initialize the recurrence, we define $Time^{rec}_{C_{mul}}(j, 0, s, \sigma) = \min_{s \in S, \sigma \in S} \left\{ T_C^{SF}(i, j, s, \sigma) \right\} + C_j$.

There are n^2 possible segments in the chain, and for each segment we try all possible pairs of speeds among the K available speeds. In addition, there are three nested loops in the algorithm, which leads to the complexity $O(n^3 + n^2K^2)$. □

6.1.2 Energy-VC: Minimizing Energy

Proposition 6. *For the* MULTISPEED *scenario, the* ENERGY-VC *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^3 + n^2K^2)$.

Proof. The algorithm for minimzing the energy with the VC-ONLY approach is similar to the minimization of the makespan (see Section 6.1.1). We replace $Time^{rec}_{C_{mul}}(n, k)$ by $Energy^{rec}_{C_{mul}}(n, k)$ and $T_{C_{re}}$ by $E_{C_{re}}$. The goal is to compute:

$$\min_{0 \le k < n} Energy^{rec}_{C_{mul}}(n, k),$$

where $Energy^{rec}_{C_{mul}}(j, k)$ can be expressed as follows:

$$Energy^{rec}_{C_{mul}}(j, k) = \min_{k \le i < j} \left\{ Energy^{rec}_{C_{mul}}(i, k-1) + \min_{s \in S, \sigma \in S} \left\{ E_{C_{re}}(i, j, s, \sigma) \right\} + E_j^{\mathrm{C}} \right\}.$$

Clearly, the complexity is the same as for makepsan minimization (Theorem 5). □

6.2 VC+V: Using verified checkpoints and single verifications

Following the VC+V approach, we aim at finding the best positions for checkpoints and verifications, as well as the best speeds for each segment, in order to minimize the expected makespan (TIME-VC) and the total energy consumption (ENERGY-VC). This section is based on the work done for the REEXECSPEED model under the VC+V approach.

6.2.1 Time-VC+V: Minimizing Makespan

Theorem 6. *For the* MultiSpeed *scenario, the* Time-VC+V *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^5K^2)$.

Proof. Recall that $T_{VC_{re}}$ from the ReExecSpeed model already accounts for the two speeds (see Equation (7)). It looks for the best places for verifications and for one segment of the chain. Instead of fixing the two speeds in advance as it was done in the previous section, we replace $Time^{rec}_{VC_{re}}$ with $Time^{rec}_{VC_{mul}}$, which tries all the possible pairs of speeds for each segment. The goal is to compute:

$$\min_{0\le k<n} Time^{rec}_{VC_{mul}}(n,k),$$

where

$$Time^{rec}_{VC_{mul}}(j,k) = \min_{k\le i<j}\left\{Time^{rec}_{VC_{mul}}(i,k-1) + \min_{s\in S,\sigma\in S}\{T_{VC_{re}}(i,j,s,\sigma)\} + C_j\right\}.$$

We initialize the recurrence with $Time^{rec}_{VC_{mul}}(j,0,s,\sigma) = \min_{s\in S,\sigma\in S}\{T_{VC_{re}}(i,j,s,\sigma)\} + C_j$.

There are n^2 possible segments in the chain, and for each segment, we try all possible pairs of speeds among the K available speeds. In addition, three nested loops are necessary to compute the expected makespan of one segment. There are n steps in the computation, hence the complexity is $O(n^5K^2)$. □

6.2.2 Energy-VC+V: Minimizing Energy

Proposition 7. *For the* MultiSpeed *scenario, the* Energy-VC+V *problem can be solved by a dynamic programming algorithm whose complexity is* $O(n^5K^2)$.

Proof. This is the same idea for the VC+V approach. We follow the algorithm used to minimize the makespan (see Section 6.2.1). We replace $Time^{rec}_{VC_{mul}}(n,k)$ with $Energy^{rec}_{VC_{mul}}(n,k)$. The goal is to compute:

$$\min_{0\le k<n} Energy^{rec}_{VC_{mul}}(n,k),$$

where $Energy^{rec}_{VC_{mul}}(j,k)$ can be expressed as follows:

$$Energy^{rec}_{VC_{mul}}(j,k) = \min_{k\le i<j}\left\{Energy^{rec}_{VC_{mul}}(i,k-1) + \min_{s\in S,\sigma\in S}\{E_{VC}(i,j,s,\sigma)\} + E^{\mathrm{C}}_j\right\}.$$

Clearly, the complexity is the same as for makespan minimization (Theorem 6). □

7 Experiments

We conduct simulations to evaluate the performance of the dynamic programming algorithms under different execution scenarios and parameter settings. We instantiate the model parameters with realistic values taken from the literature, and we point out that the code for all algorithms and simulations is publicly available at `http://graal.ens-lyon.fr/~yrobert/failstop-silent`, so that interested readers can build relevant scenarios of their choice.

7.1 Simulation settings

We generate linear chains with different number n of tasks while keeping the total computational cost at $W = 5 \times 10^4$ seconds $\approx$ 14 hours. The total amount of computation is distributed among the tasks in three different patterns: (1) *Uniform*, all tasks share the same cost W/n, as in matrix multiplication or in some iterative stencil kernels; (2) *Decrease*, task T_i has cost $\alpha \cdot (n+1-i)^2$, where $\alpha \approx 3W/n^3$. This quadratically decreasing function resembles some dense matrix solvers, e.g., using LU or QR factorization. (3) *HighLow*, a set of identical tasks with large cost is followed by tasks with small cost. This distribution is created to distinguish the performance of different execution scenarios. In this case, we fix the number of large tasks to be 10% of the total number n of tasks while varying the computational cost dedicated to them.

We adopt the set of speeds from the Intel Xscale processor. Following [27], the normalized speeds are $\{0.15, 0.4, 0.6, 0.8, 1\}$ and the fitted power function is given by $P(s) = 1550s^3 + 60$. From the discussion in Section 2.3, we assume the following model for the average error rate of fail-stop errors:

$$\lambda^F(s) = \lambda^F_{\text{ref}} \cdot 10^{\frac{d \cdot |s_{\text{ref}} - s|}{s_{\max} - s_{\min}}} , \tag{8}$$

where $s_{\text{ref}} \in [s_{\min}, s_{\max}]$ denotes the reference speed with the lowest error rate λ^F_{ref} among all possible speeds in the range. The above equation allows us to account for higher fail-stop error rates when the CPU speed is either too low or too high. In the simulations, the reference speed is set to be $s_{\text{ref}} = 0.6$ with an error rate of $\lambda^F_{\text{ref}} = 10^{-5}$ for fail-stop errors, and the sensitivity parameter is set to be $d = 3$. These parameters represent realistic settings reported in the literature [1, 3, 34], and they correspond to $0.83 \sim 129$ errors over the entire chain of computation depending on the processing speed chosen.

For silent errors, we assume that its error rate is related to that of the fail-stop errors by $\lambda^S(s) = \eta \cdot \lambda^F(s)$, where $\eta > 0$ is constant parameter. To achieve realistic scenarios, we try to vary η to assess the impact of both error sources on the performance. However, we point out that our approach is completely independent of the evolution of the error rates as a function of the speed. In a practical setting, we are given a set of discrete speeds and two error rates for each speed, one for fail-stop errors and one for silent errors. This is enough to instantiate our model.

In addition, we define cr to be the ratio between the checkpointing/recovery cost and the computational cost for the tasks, and define vr to be the ratio between the verification cost and the computational cost. By default, we execute the tasks using the reference speed s_{ref}, and set $\eta = 1$, $cr = 1$ and $vr = 0.01$. This initial setting corresponds to tasks with costly checkpoints (same order of magnitude as the costs of the tasks) and lightweight verifications (average cost 1% of task cost); examples of such tasks are data-oriented kernels processing large files and checksumming for verification. We will vary these parameters to study their impacts on the performance.

7.2 Results

7.2.1 SingleSpeed scenario for makespan

The first set of experiments is devoted to the evaluation of the time-optimal algorithms in the SingleSpeed scenario.

Impact of n and cost distribution. Figure 4(a) shows the expected makespan (normalized by the ideal execution time at the default speed, i.e., $W/0.6$) with different n and cost distributions. For the *HighLow* distribution, the large tasks are configured to contain 60% of the total computational cost. The results show that having more tasks reduces the expected makespan, since it enables the algorithms to place more checkpoints and verifications, as can be seen in Figure 4(b). The distribution that renders a larger variation in task sizes create more difficulty in the placement of checkpoints/verfications, thus resulting in worse makespan. The figure also compares the performance of the TIME-VC algorithm with that of TIME-VC+V. The latter algorithm, being more flexible, naturally leads to improved makespan under all cost distributions. Because of the additionally placed verifications, it also reduces the number of verified checkpoints in the optimal solution.

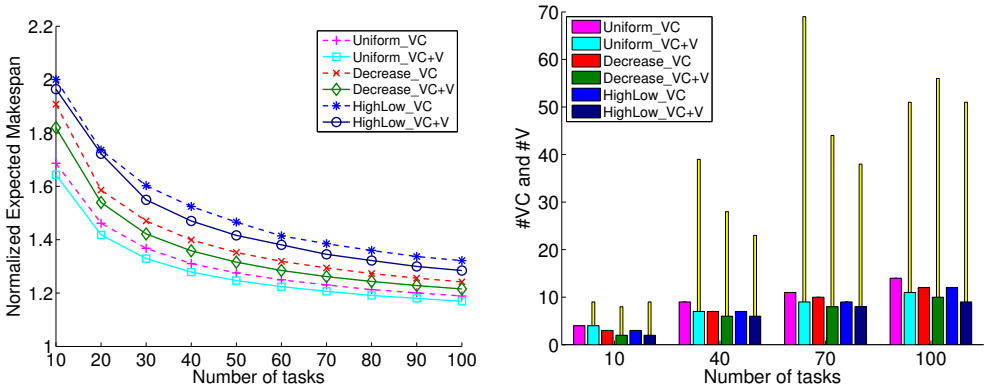
Comparison with a divisible load application. Figure 5(a) compares the makespan of the TIME-VC algorithm under *Uniform* cost distribution with the makespan of a divisible load application, whose total load is W and whose checkpointing cost is the same as the corresponding discrete tasks. For the divisible load application, we use Proposition 1 to compute the optimal period, the waste and then derive the makespan. In addition, Figure 5(b) compares the number of verified checkpoints in the two cases. We see that the makespan for divisible load is worse for large cr and becomes better as cr decreases. Furthermore, the makespans in both cases get closer when the number of tasks increases. This is because the checkpointing cost decreases with cr and as n increases, which makes the first order approximation used in Proposition 1 more accurate. Moreover, as divisible load does not impose restrictions in the checkpointing positions, it tends to place more checkpoints than the case with discrete tasks.

We could think of the following greedy algorithm as an alternative to the TIME-VC algorithm for a linear chain of tasks: position the next checkpoint as soon as the time spent on computing since the last checkpoint plus the checkpointing cost of the current task exceeds the optimal period given by Proposition 1. Figure 5 suggests that this linear-time algorithm (with cost $O(n)$) would give a good approximation of the optimal solution (returned by the TIME-VC algorithm with cost $O(n^3)$), at least for uniform distribution of task costs.

In the rest of this section, we will focus on the TIME-VC+V algorithm and $n = 100$ tasks with *Uniform* cost distribution.

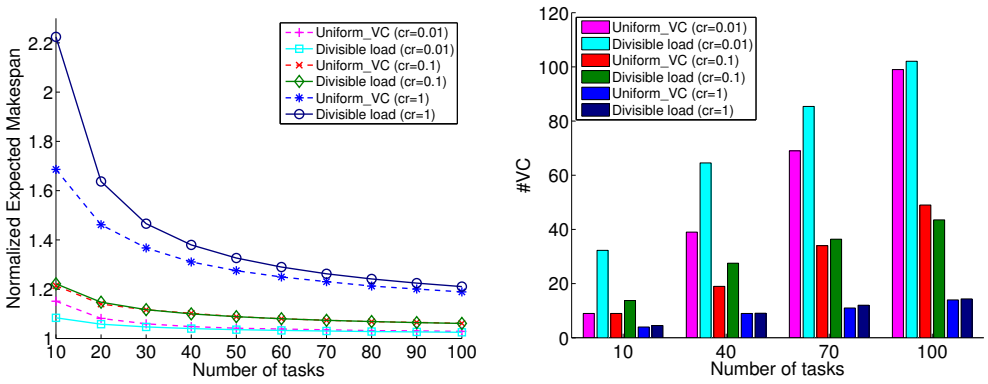
Impact of η and error mode. Figure 6(a) compares the performance under different error modes, namely, fail-stop (F) only, silent (S) only, and fail-stop plus silent with different values of η. As silent errors are harder to detect and hence to deal with, the S-only case leads to larger makespan than the F-only case. In the presence of both types of errors, the makespan becomes worse with larger η, i.e., with increased rate for silent errors, despite the algorithm's effort to place more checkpoints as shown in Figure 6(b). Moreover, the performance degrades significantly as the CPU speed is set below the reference speed s_{ref} for the error rate increases exponentially. A higher CPU speed, on the other hand, first improves the makespan by executing the tasks faster and then causes degradation due to a larger increase in the error rate.

Impact of cr and vr. Figure 7(a) presents the impact of checkpointing/recovery ratio (cr) and verification ratio (vr) on the performance. Clearly, a smaller cr (or vr) enables



(a) (b)

Figure 4: Impact of n and cost distribution on the performance of the TIME-VC and TIME-VC+V algorithms. In (b), the thick bars represent the verified checkpoints and the yellow thin bars represent the total number of verifications.



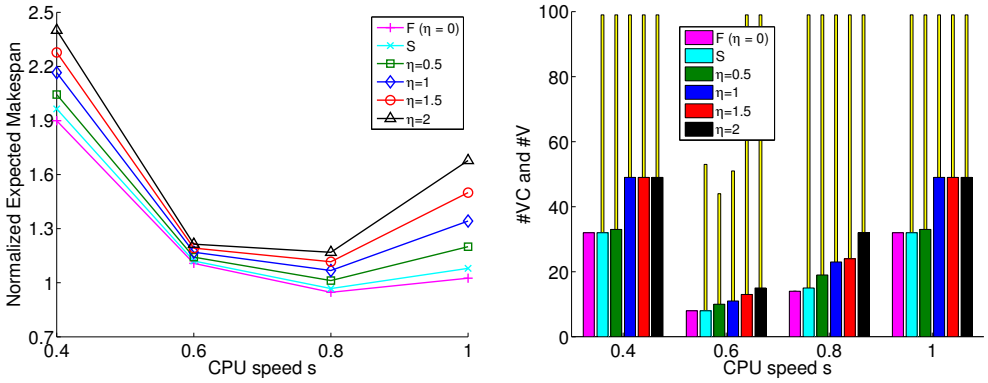
(a) (b)

Figure 5: Performance comparison of the TIME-VC algorithm for tasks with *Uniform* cost distribution and the optimal checkpointing algorithm for divisible load application.

the algorithm to place more checkpoints (or verifications), which leads to better makespan. Having more checkpoints also allows the algorithm to use faster speeds to complete the tasks. Finally, if checkpointing cost is on par with verification cost (e.g., $cr = 0.1$), reducing the verification cost can additionally increase the number of checkpoints (e.g., at $s = 0.6$), since each checkpoint also has a verification cost associated with it. For high checkpointing cost, however, reducing the verification cost could no longer influence the algorithm's checkpointing decisions.

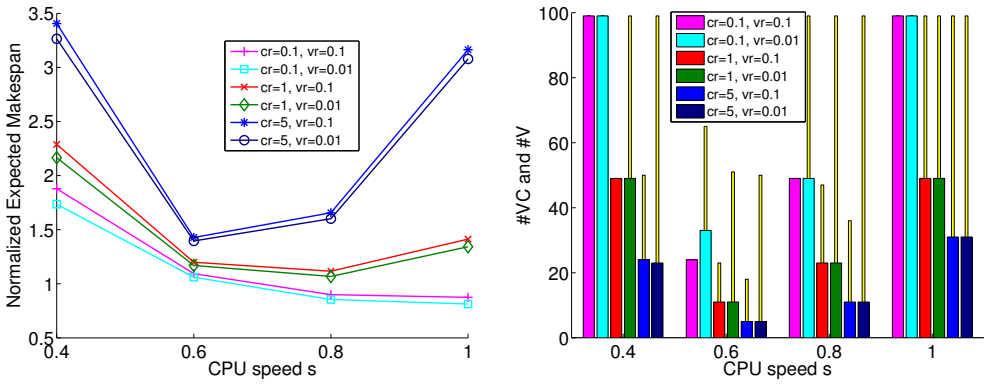
7.2.2 SingleSpeed scenario for energy

This set of experiments focuses on the evaluation of the ENERGY-VC+V algorithm in the SINGLESPEED scenario. The default power parameters are set to be $P_{idle} = 60$ and $P_{cpu}(s) = 1550s^3$ according to [27]. The dynamic power consumption due to I/O is equal to the dynamic power of the CPU at the lowest discrete speed 0.15. We will also vary these parameters to



(a) (b)

Figure 6: Impact of η and speed s on the performance. F denotes fail-stop error only and S denotes silent error only. Speed $s = 0.15$ leads to extremely large makespan, which is omitted in the figure.

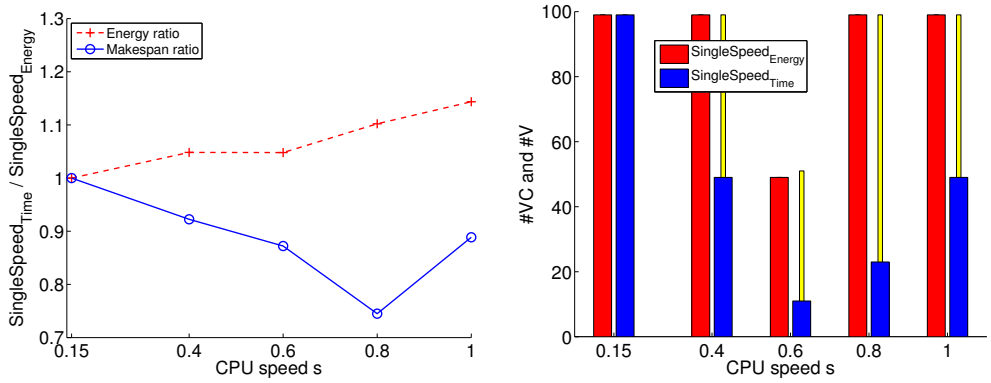


(a) (b)

Figure 7: Impact of cr and vr on the performance with different CPU speeds.

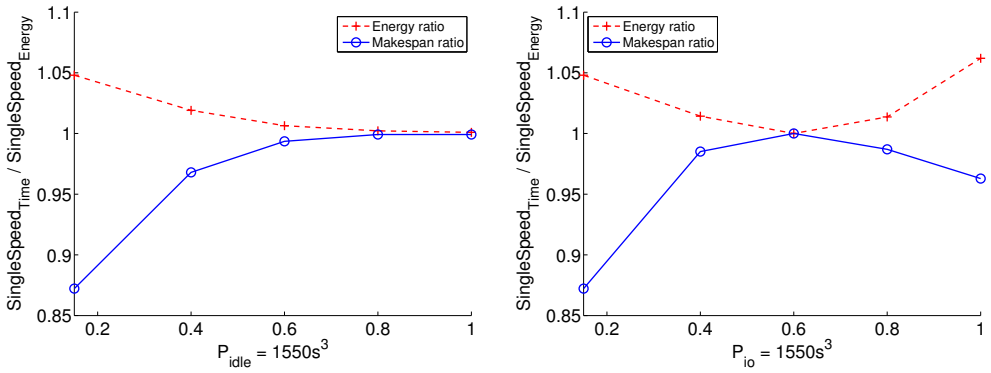
study their impacts.

Impact of CPU speed s. Figure 8 compares the performance of the Energy-VC+V algorithm in comparison with its makespan counterpart Time-VC+V for $n = 100$ tasks. At speed 0.15, the power consumed by the CPU is identical to that of I/O. This yields the same number of checkpoints placed by the two algorithms, which in turn leads to the same performance for both makespan and energy. As the CPU speed increases, the I/O power consumption becomes much smaller, so the energy algorithm tends to place more checkpoints to improve the energy consumption at the expense of makespan. From Figure 6, we know that the makespan of Time-VC+V degrades at speed $s = 1$. This diminishes its makespan advantage at the highest discrete speed. Figure 8 also suggests that the Time-VC+V algorithm running at speed $s = 0.8$ offers a good energy-makespan tradeoff. Compared to the Energy-VC+V algorithm, it provides more than 25% improvement in makespan with only 10% degradation in energy under the default parameter settings.



(a) (b)

Figure 8: Relative performance of the ENERGY-VC+V and TIME-VC+V algorithms with different CPU speeds.



(a) (b)

Figure 9: Impact of P_{idle} and P_{io} on the relative performance of the ENERGY-VC+V and TIME-VC+V algorithms at $s = 0.6$.

Impact of P_{idle} and P_{io}. Figure 9 shows the relative performance of the two algorithms by varying P_{idle} and P_{io} separately according to the dynamic power function $1550s^3$, while keeping the other one at the smallest CPU power, i.e., $1550 \cdot 0.15^3$. The CPU speed is fixed at $s = 0.6$. Figure 10 further shows the number of checkpoints in the ENERGY-VC+V algorithm at different P_{idle} and P_{io} values. (The TIME-VC+V algorithm is apparently not affected by these two parameters and always places 11 checkpoints in this experiment.) First, setting the smallest value for both parameters creates a big gap between the CPU and I/O power consumptions. This leads to a large number of checkpoints placed by the ENERGY-VC+V algorithm. Increasing P_{idle} closes this gap and hence reduces the number of checkpoints, which leads to the performance convergence of the two algorithms. While increasing P_{io} has the same effect, a larger value than $P_{cpu} = 1550 \cdot 0.6^3$ further reduces the number of checkpoints below 11, since checkpointing is now less power-efficient. This again gives the ENERGY-VC+V algorithm advantage in terms of energy.

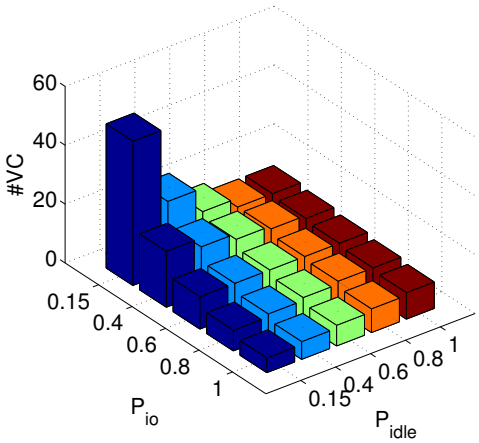


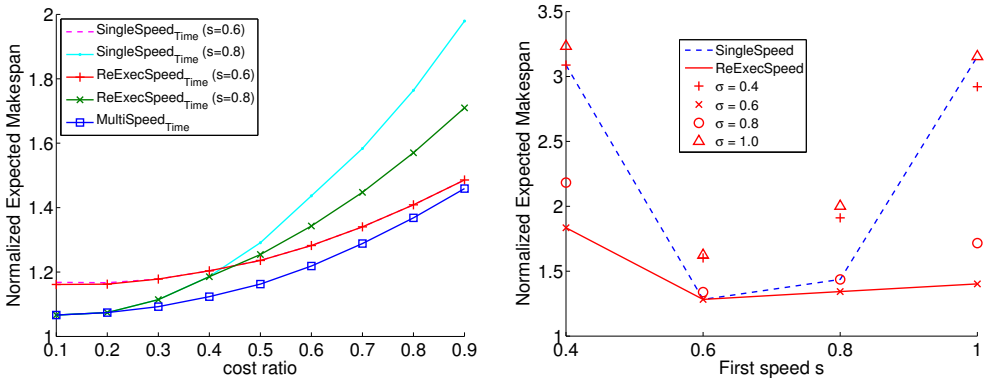
Figure 10: Number of checkpoints placed by the ENERGY-VC+V algorithm with different P_{io}, P_{idle} values ($= 1550s^3$) at $s = 0.6$.

7.2.3 ReExecSpeed and MultiSpeed scenarios

This set of experiments evaluates the REEXECSPEED and MULTISPEED scenarios for both makespan and energy. To distinguish them from the SINGLESPEED model, we consider the *HighLow* distribution, which yields a larger variance among the computational costs of the tasks. In the simulation, we again focus on the VC+V algorithms for $n = 100$ tasks, and vary the *cost ratio*, which is the percentage of computational cost in the large tasks compared to the total computational cost.

Figure 11(a) compares the makespan of the TIME-VC+V algorithms under the three scenarios. For the SINGLESPEED and REEXECSPEED scenarios, only $s = 0.6$ and $s = 0.8$ are drawn, since the other speeds lead to much larger makespans. For a small cost ratio, no task has a very large computational cost, so the faster speed $s = 0.8$, despite its higher error rate, appears to give the best performance as we have already seen in Figure 6(a). When the cost ratio increases, tasks with large cost start to emerge. With the high error rate of $s = 0.8$, these tasks will experience many re-executions, thus degrading the makespan. Here, $s = 0.6$ becomes the best speed due to its smaller error rate. In the REEXECSPEED scenario, regardless of the initial speed s, the best re-execution speed σ is always 0.6 or 0.8 depending on the cost ratio, and it improves upon the respective SINGLESPEED scenario with the same initial speed, as we can see in Figure 11(b) for cost ratio of 0.6. However, the improvement is marginal compared to the best performance achievable in the SINGLESPEED scenario. The MULTISPEED scenario, with its flexibility to choose different speeds depending on the costs of the tasks, always provides the best performance. The advantage is especially evident at medium cost ratios with up to 6% improvement, as this situation contains a good mix of large and small tasks, which is hard to deal with by using fixed speed(s). Figure 12 shows similar results for the energy consumption of the ENERGY-VC+V algorithms under the three scenarios, with more than 7% improvement in the MULTISPEED scenario. In this case, speed $s = 0.4$ consumes less energy at small cost ratio due to its better power efficiency.

Finally, Figure 13 shows the relative performance of the ENERGY-VC+V and TIME-VC+V algorithms under the MULTISPEED scenario. As small cost ratio favors speed 0.4 for the energy algorithm and 0.8 for the time algorithm, it distinguishes the two algorithms in



(a) (b)

Figure 11: Performance comparison of the TIME-VC+V algorithms in MULTISPEED, REEXECSPEED and SINGLESPEED scenarios for $n = 100$ tasks with *HighLow* cost distribution. In (b), the cost ratio is 0.6.

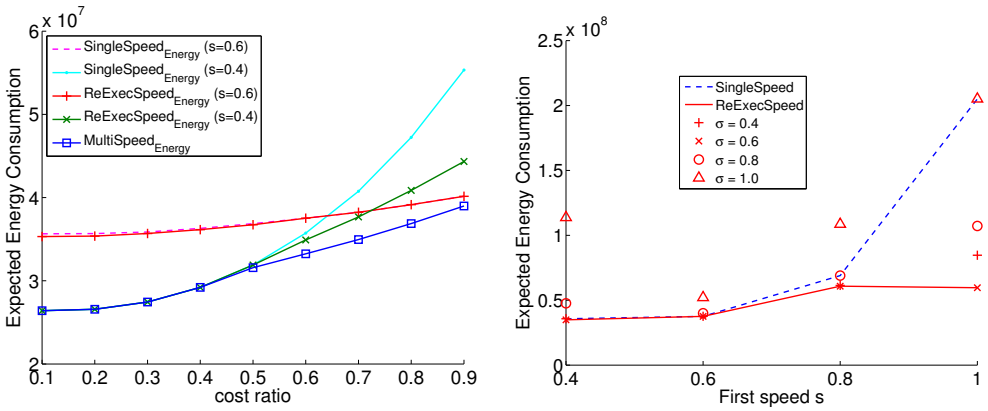
terms of their respective optimization objectives, by up to 100% in makespan and even more in energy consumption. Increasing the cost ratio creates more computationally demanding tasks, which need to be executed at speed 0.6 for both makespan- and energy-efficiency as it incurs fewer errors. This reduces the performance gap of the two algorithms as well as the number of checkpoints placed by them.

7.2.4 Summary

To summarize, we have evaluated and compared various algorithms under different execution scenarios. The algorithms under the most flexible VC+V and MULTISPEED scenario generally provide better performance, which in practice would translate to shorter makespan or lower energy consumption.

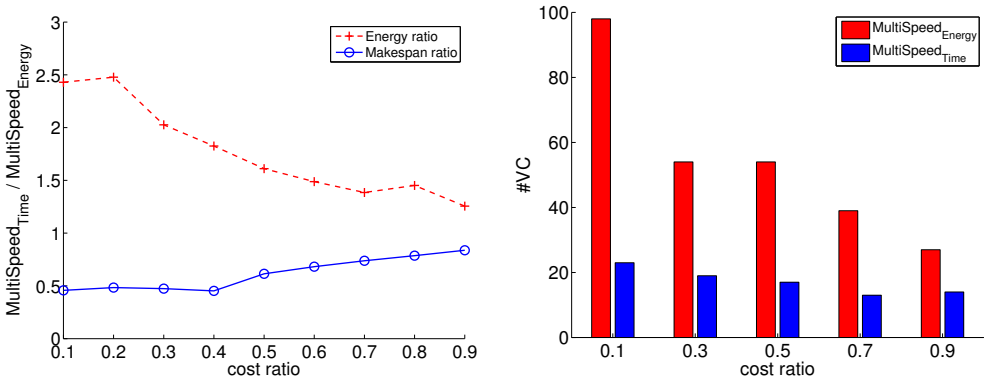
For tasks with similar computational costs as in the *Uniform* distribution, we observe that the SINGLESPEED algorithm, or the greedy approximation in the context of divisible load application, could in fact provide comparable solutions with much lower computational complexity. The REEXECSPEED algorithms show only marginal benefit compared to SINGLESPEED, but clear performance improvements are observed from the MULTISPEED algorithms, especially for tasks with very different costs. The results also show that the optimal solutions are often achieved by processing around the reference speed that yields the least number of failures.

In terms of computation time, the most advanced VC+V algorithms in the MULTISPEED scenario are about 30 times slower than the simple VC-ONLY algorithm in the SINGLESPEED scenario. For $n = 100$, the advanced algorithms use less than a minute to find the optimal solution. Due to the high complexity, however, it may require tens of minutes or even hours for larger n. As application workflows rarely exceed a few tens of tasks, these algorithms could be feasibly applied in many practical contexts to generate better solutions.



(a) (b)

Figure 12: Performance comparison of the ENERGY-VC+V algorithms in MULTISPEED, RE-EXECSPEED and SINGLESPEED scenarios for $n = 100$ tasks with *HighLow* cost distribution. In (b), the cost ratio is 0.6.



(a) (b)

Figure 13: Impact of cost ratio on the relative performance of the ENERGY-VC+V and TIME-VC+V algorithms under the MULTISPEED scenario.

8 Conclusion

In this paper, we have presented a general-purpose solution that combines checkpointing and verification mechanisms to cope with both fail-stop errors and silent data corruptions. By using dynamic programming, we have devised polynomial-time algorithms that decide the optimal checkpointing and verification positions on a linear chain of tasks. The algorithms can be applied to several execution scenarios to minimize the expected execution time (makepsan) or energy consumption. In addition, we have extended the classical bound of Young/Daly for divisible load applications to handle both fail-stop and silent errors. The results are supported by a set of extensive simulations, which demonstrate the quality and tradeoff of our optimal algorithms under a wide range of parameter settings. One useful future direction is to extend our study from linear chains to other application workflows, such as tree graphs, fork-join graphs, series-parallel graphs, or even general DAGs.

References

[1] I. Assayad, A. Girault, and H. Kalla. Tradeoff exploration between reliability, power consumption, and execution time for embedded systems. *International Journal on Software Tools for Technology Transfer*, 15(3):229–245, 2013.

[2] G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni. On the combination of silent error detection and checkpointing. In *PRDC 2013, the 19th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society Press, 2013.

[3] G. Aupy, A. Benoit, and Y. Robert. Energy-aware scheduling under reliability and makespan constraints. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, pages 1–10, 2012.

[4] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):3:1–3:39, 2007.

[5] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *CoRR*, abs/1312.2674, 2013.

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

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

[8] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proc. 22nd Int. Conf. on Supercomputing*, ICS '08, pages 155–164. ACM, 2008.

[9] 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.

[10] Z. Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *Proc. 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 167–176. ACM, 2013.

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

[12] A. Das, A. Kumar, B. Veeravalli, C. Bolchini, and A. Miele. Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pages 61:1–61:6, 2014.

[13] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *IEEE International on Reliability Physics Symposium (IRPS)*, pages 5B.4.1–5B.4.7, 2011.

[14] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder. Temperature management in data centers: Why some (might) like it hot. *SIGMETRICS Perform. Eval. Rev.*, 40(1):163–174, 2012.

[15] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for HPC. In *Proc. ICDCS '12*. IEEE Computer Society, 2012.

[16] 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.

[17] W.-C. Feng. Making a case for efficient supercomputing. *Queue*, 1(7):54–64, Oct. 2003.

[18] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proc. of the ACM/IEEE SC Int. Conf.*, SC '12. IEEE Computer Society Press, 2012.

[19] M. Heroux and M. Hoemmen. Fault-tolerant iterative methods via selective reliability. Research report SAND2011-3915 C, Sandia National Laboratories, 2011.

[20] C.-H. Hsu and W. chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the ACM/IEEE Supercomputing Conference*, pages 1–9, 2005.

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

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

[23] G. Lu, Z. Zheng, and A. A. Chien. When is multi-version checkpointing needed. In *3rd Workshop for Fault-tolerance at Extreme Scale (FTXS)*. ACM Press, 2013. `https://sites.google.com/site/uchicagolssg/lssg/research/gvr`.

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

[25] 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.

[26] M. Patterson. The effect of data center temperature on energy efficiency. In *Proceedings of 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, pages 1167–1174, 2008.

[27] N. B. Rizvandi, A. Y. Zomaya, Y. C. Lee, A. J. Boloori, and J. Taheri. Multiple frequency selection in dvfs-enabled processors to minimize energy consumption. In A. Y. Zomaya and Y. C. Lee, editors, *Energy-Efficient Distributed Computing Systems*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2012.

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

[29] O. Sarood, E. Meneses, and L. V. Kale. A 'cool' way of improving the reliability of HPC machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 58:1–58:12, 2013.

[30] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proc. ICS '12*. ACM, 2012.

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

[32] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, page 374, 1995.

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

[34] B. Zhao, H. Aydin, and D. Zhu. Reliability-aware dynamic voltage scaling for energy-constrained real-time embedded systems. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 633–639, 2008.

[35] D. Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 35–40, 2004.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399