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

Anne Benoit, Aurélien Cavelan, Yves Robert, Hongyang Sun

informatics *mathematics*

Inria

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

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

Project-Teams ROMA

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: HPC, resilience, checkpoint, verification, failures, fail-stop error, silent data corruption, silent error

---

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

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, point de sauvegarde, checkpoint, vérification, pannes, erreur fail-stop, corruption silencieuse de données, erreur silencieuse

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 it is paid throughout the duration of the execution, which calls for a shorter execution (at higher speed). Overall there are tradeoffs 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 rollback 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.). It 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 checkpoint (VC). 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 rollback 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, 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 seems 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.

One 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 executions. 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 execution 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 an adjusted speed to either speed up or to slow down the

re-execution after an error strikes, depending on the optimization objective. 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. For each execution scenario, we provide an optimal dynamic programming algorithm to determine the best locations of checkpoints and verifications (and for the MULTISPEED scenario we also 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 checkpoints with a verification mechanism.
- We consider several execution scenarios, first with a single speed, then in case of re-execution, and finally with several discrete speeds that can freely change after each checkpoint.
- For all scenarios and for both makespan and energy objectives, we consider two approaches, one using verified checkpoints only, and the other one using additional isolated verifications. We provide a dynamic programming algorithm that determines the best locations of checkpoints and of verifications across application tasks for each scenario/approach/objective combination.
- We provide an extensive set of simulations to support the theory and which enables us to assess the usefulness of each algorithm.

The rest of the paper is organized 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: for all three execution scenarios, we 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 back to 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, which 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 [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, which dramatically decrease the cost of detection. Many techniques have been advocated. They include memory scrubbing [22] and 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]. Heroux and Hoemmen [19] design a fault-tolerant GMRES capable of converging despite silent errors. 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 equi-distance verifications. For PCG, the verification amounts to checking the orthogonality of two vectors and to recomputing and checking 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]:

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

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 (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
$p_{i,j}^F(s)$	Probability that a fail-stop error strikes between tasks T_i and T_j
$p_{i,j}^S(s)$	Probability that a silent error strikes between tasks T_i and T_j
Energy	
P_{idle}	Static/idle 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 at speed s
$E_i^{\text{V}}(s)$	Energy needed to verify task T_i at speed s
E_i^{C}	Energy needed to checkpoint task T_i
E_i^{R}	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 (including verification), we use DVFS to change the speed of the processors, and assume a set $S = \{s_1, s_2, \ldots, s_K\}$ of K discrete computing speeds. During checkpointing and recovery, 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).

Assume w.l.o.g. that there is no overlap between CPU operations and I/O transfers. Then the total energy consumed during the execution of the application can be expressed as

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

where $T_{cpu}(s_i)$ is the time spent on computing at speed s_i, $T_{cpu} = \sum_{i=1}^{K} T_{cpu}(s_i)$ is the total time spent on computing, and T_{io} is the total time spent on 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, and to cope with silent errors, an additional verification mechanism is used. The time to checkpoint (the output of) task T_i is C_i, the time to recover from (the checkpoint of) task T_i is R_i, and the time to verify (the output of) task T_i at speed s is $V_i(s)$. We assume that both fail-stop errors and silent errors follow an exponential distribution with average rates $\lambda^F(s)$ and $\lambda^S(s)$, respectively, where s denotes the current computing speed. Given an error rate λ, let $p(\lambda, L) = 1 - e^{-\lambda L}$ denote the probability that a error strikes during an execution of length L. For convenience, we define $p_{i,j}^F(s) = p(\lambda^F(s), T_{i,j}(s))$ to be the probability that a fail-stop error strikes when executing from T_i to T_j, and define $p_{i,j}^S(s) = p(\lambda^S(s), T_{i,j}(s))$ similarly for silent errors.

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

3.4   Execution scenarios

We consider three different execution scenarios:

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

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

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-ONLY   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-ONLY   Minimize total energy consumption using the VC-ONLY approach.

ENERGY-VC+V   Minimize total energy consumption using the VC+V approach.

For the SINGLESPEED and REEXECSPEED scenarios, we have to decide the optimal locations of the checkpoints (VC-ONLY) and of the verifications (VC+V). For the MULTISPEED scenario, 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. For each approach, we present an optimal polynomial-time dynamic programming algorithm.

4.1   VC-only: Using verified checkpoints only

In 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-ONLY) or the total energy consumption (ENERGY-VC-ONLY).

4.1.1   Time-VC-Only: Minimizing Makespan

Theorem 1. *For the* SINGLESPEED *scenario, the* TIME-VC-ONLY *problem can be solved by a dynamic programming algorithm in $O(n^2)$ time.*

Proof. We define $Time_C(j,s)$ to be the optimal expected time to successfully execute tasks $T_1, \ldots, T_j$, where T_j has a verified checkpoint, and there are possibly other verified checkpoints from T_1 to T_{j-1}. Note that we always verify and checkpoint the last task T_n to save the final result, so the goal is to find $Time_C(n,s)$.

To compute $Time_C(j,s)$, we formulate the following dynamic program by trying all possible locations for the last checkpoint before T_j (see Figure 1):

$$Time_C(j,s) = \min_{0 \le i < j} \{Time_C(i,s) + T_C(i+1,j,s)\} + C_j \ ,$$

where $T_C(i,j,s)$ is the expected time to successfully execute the tasks T_i to T_j, provided that T_{i-1} and T_j are both verified and checkpointed while no other task in between is verified nor checkpointed. Note that we also account for the checkpointing cost C_j for task T_j, which is not included in the definition of T_C. To initialize the dynamic program, we define $Time_C(0,s) = 0$.
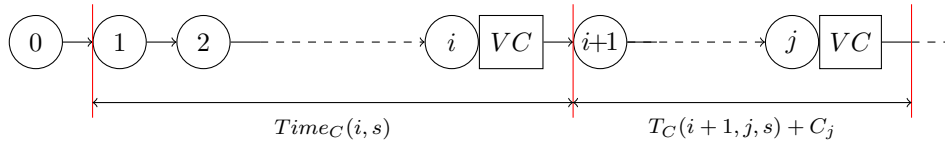

Figure 1: Illustration of the dynamic programming formulation for $Time_C(j,s)$.

In the following, we show how to compute $T_C(i,j,s)$ for each (i,j) pair with $i \le j$. We start by considering only *silent errors* and use the notation $T_C^S(i,j,s)$ for that purpose. Silent errors can occur at any time during the computation but we can only detect them after all tasks have been executed. Thus, we always have to pay $T_{i,j}(s) + V_j(s)$, the time to execute from task T_i to T_j at speed s and to verify T_j. If the verification fails, which happens with probability $p_{i,j}^S(s)$, a silent error has occurred and we have to recover from T_{i-1} and start anew. For convenience, we assume that there is a virtual task T_0 that is always verified and checkpointed, with a recovery cost $R_0 = 0$. Mathematically, we can express $T_C^S(i,j,s)$ as

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

which simplifies to

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

Things are different when accounting for *fail-stop errors*, because the application will stop immediately when a fail-stop error occurs, even in the middle of the computation. Let $T_{lost_{i,j}}(s)$ denote the expected time lost during the execution from T_i to T_j if a fail-stop error strikes, and it can be expressed as

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

where $\mathbb{P}(X = x)$ denotes the probability that a fail-stop error strikes at time x. By definition, we have $\mathbb{P}(X < T_{i,j}(s)) = 1 - e^{-\lambda^F(s)T_{i,j}(s)}$. Integrating by parts, we can get

$$T_{lost_{i,j}}(s) = \frac{1}{\lambda^F(s)} - \frac{T_{i,j}(s)}{e^{\lambda^F(s)T_{i,j}(s)} - 1} \ . \tag{2}$$

Therefore, the expected execution time $T_C^F(i, j, s)$ when considering only fail-stop errors is given by

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

which simplifies to

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

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

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

When plugging $p_{i,j}^F(s)$, $p_{i,j}^S(s)$ and $T_{lost_{i,j}}(s)$ into the above equation, we get

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

By setting $T_C(i, j, s) = T_C^{SF}(i, j, s)$, we can now compute $Time_C(j, s)$ for all $j = 1, \cdots, n$. For the complexity, the computation of $T_C^{SF}(i, j, s)$ for all (i, j) pairs with $i \leq j$ takes $O(n^2)$ time. The computation of the dynamic programming table for $Time_C(j, s)$ also takes $O(n^2)$ time, as $Time_C(j, s)$ depends on at most j other entries in the same table, which are already computed. Therefore, the overall complexity is $O(n^2)$, and this concludes the proof. $\qquad\square$

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 [33, 11] as follows.

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

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

where C is the checkpointing cost, V is the verification cost, $\lambda^F(s)$ is the rate of fail-stop errors at speed s and $\lambda^S(s)$ is the rate of silent errors at speed s.

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

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

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

Now with both fail-stop and silent errors, the error-free waste becomes $Waste_{ef} = (V + C)/T$, because we lose $V + C$ time-units every period of length T. The waste due to failures becomes $Waste_{fail} = \lambda^F(s)(R + T/2) + \lambda^S(s)(R + T)$. 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,

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

Differentiating, we find that $T_{opt}(s) = \sqrt{\frac{2(V+C)}{\lambda^F(s)+2\lambda^S(s)}}$ minimizes the waste. 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 $1/\lambda^S(s)$ for silent errors. □

In Section 7 (see Figure 5), we compared the makespan derived from this formula with the performance of the VC-ONLY algorithm. 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-Only: Minimizing Energy

Proposition 2. *For the* SINGLESPEED *scenario, the* ENERGY-VC-ONLY *problem can be solved by a dynamic programming algorithm in $O(n^2)$ time.*

Proof. The proof is similar to that of the TIME-VC-ONLY problem presented in Section 4.1.1. Here, we replace $Time_C(j, s)$ with $Energy_C(j, s)$, which is the optimal expected energy to successfully execute the tasks T_1 to T_j. Instead of $T_C^{SF}(i, j, s)$, we use $E_C^{SF}(i, j, s)$ to denote the total expected energy to successfully execute all the tasks from T_i to T_j without any checkpoint and verification in between, while T_{i-1} and T_j are both verified and checkpointed. The goal is to find $Energy_C(n, s)$ and the corresponding dynamic program is formulated as:

$$Energy_C(j, s) = \min_{0 \le i < j} \{Energy_C(i, s) + E_C^{SF}(i + 1, j, s)\} + E_j^C .$$

Previously we used $T_{lost_{i,j}}(s)$ to denote the expected time lost when a fail-stop error occurs. Now, we use $E_{lost_{i,j}}(s) = T_{lost_{i,j}}(s)\,(P_{cpu}(s) + P_{idle})$ to denote the expected energy lost when a fail-stop error stroke during the execution from task T_i to task T_j. Therefore, we can express $E_C^{SF}(i,j,s)$ as follows:

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

which simplifies to

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

Clearly, the time complexity is the same as that of the makespan minimization algorithm (in Theorem 1).   □

4.2 VC+V: Using verified checkpoints and single verifications

In this approach, we can place additional verifications between two checkpoints, which allows to detect (silent) errors before reaching the next checkpoint, and hence to avoid wasted execution by performing early recoveries. 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* Single Speed *scenario, the* Time-VC+V *problem can be solved by a dynamic programming algorithm in $O(n^3)$ time.*

Proof. In the Time-VC-Only problem, we were only allowed to place verified checkpoints. Here, we can add single verifications that are not associated with a checkpoint. The main idea is to replace T_C in the dynamic program of Theorem 1 by another expression $Time_V(i,j,s)$, which denotes the optimal expected time to successfully execute from task T_i to task T_j (and to verify it), provided that T_{i-1} has a verified checkpoint and only single verifications are allowed within tasks $T_i, \ldots, T_{j-1}$. Furthermore, we use $Time_{VC}(j,s)$ to denote the optimal expected time to successfully execute the first j tasks, where T_j has a verified checkpoint, and there are possibly other verified checkpoints and single verifications before T_j. The goal is to find $Time_{VC}(n,s)$. The dynamic program to compute $Time_{VC}(j,s)$ can be formulated as (see Figure 2):

$$
Time_{VC}(j,s) = \min_{0 \le i < j}\{Time_{VC}(i,s) + Time_V(i+1,j,s)\} + C_j \;.
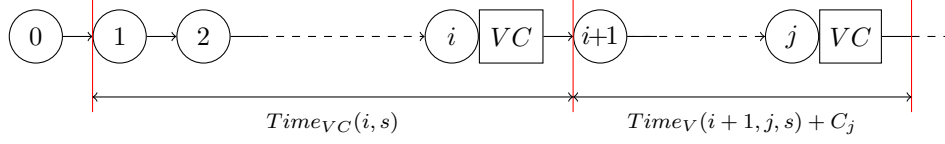$$

Figure 2: Illustration of the dynamic programming formulation for $Time_{VC}(j, s)$.

In particular, we try all possible locations for the last checkpoint before T_j, and for each location T_i, we compute the optimal expected time $Time_V(i + 1, j, s)$ to execute tasks T_{i+1} to T_j with only single verifications in between. We also account for the checkpointing time C_j, which is not included in the definition of $Time_V$. By initializing the dynamic program with $Time_{VC}(0, s) = 0$, we can then compute the optimal solution as in the TIME-VC-ONLY problem.

It remains to compute $Time_V(i, j, s)$ for each (i, j) pair with $i \leq j$. To this end, we formulate another dynamic program by trying all possible locations for the last single verification before T_j (see Figure 3):

$$Time_V(i, j, s) = \min_{i-1 \leq l < j} \{Time_V(i, l, s) + T_V(l + 1, j, i - 1, s)\} \; ,$$

where $T_V(i, j, l_c, s)$ is the expected time to successfully execute all the tasks from T_i to T_j (and to verify T_j), knowing that if an error strikes, we can recover from T_{l_c}, the last task before T_i to have a verified checkpoint.
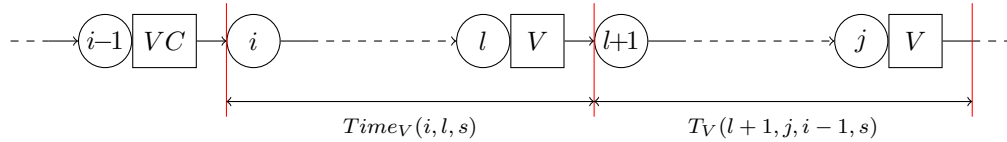

Figure 3: Illustration of the dynamic programming formulation for $Time_V(i, j, s)$.

First, we show how to compute $T_V(i, j, l_c, s)$. When accounting for only *silent errors* (with notation T_V^S), we always execute from task T_i to task T_j and then verify T_j. In case of failure, we recover from T_{l_c} and redo the entire computation from T_{l_c+1} to T_j, which contains a single verification after T_{i-1} and possibly other single verifications between T_{l_c+1} and T_{i-2}. Hence, we have

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

which simplifies to

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

When there are only *fail-stop errors*, we do not need to perform any single verification, and hence the problem becomes simply the TIME-VC-ONLY problem. When accounting for both *silent and fail-stop errors* (with notation T_V^{SF}), we apply the same method as in the proof of Theorem 1. Specifically, if a fail-stop error strikes between two verifications, we directly perform a recovery; otherwise we check for silent errors:

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

When plugging $p_{i,j}^F$, $p_{i,j}^S$ and $T_{lost_{i,j}}$ into the above equation, we get

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

Notice that $T_V(i, j, l_c, s)$ depends on the value of $Time_V(l_c + 1, i - 1, s)$, except when $l_c + 1 = i$, in which case we initialize $Time_V(i, i - 1, s) = 0$. Hence, in the dynamic program, $Time_V(i, j, s)$ can be expressed as a function of $Time_V(i, l, s)$ for all $l = i - 1, \cdots, j - 1$.

Finally, the complexity is dominated by the computation of the second dynamic programming table for $Time_V(i, j, s)$, which contains $O(n^2)$ entries and each entry depends on at most n other entries that are already computed. Hence, the overall complexity of the algorithm is $O(n^3)$, and this concludes the proof. □

4.2.2 Energy-VC+V: Minimizing Energy

Proposition 3. *For the* SINGLESPEED *scenario, the* ENERGY-VC+V *problem can be solved by a dynamic programming algorithm in $O(n^3)$ time.*

Proof. The proof is similar to that of the TIME-VC+V problem presented in Section 4.2.1. Here, we replace $Time_{VC}(j, s)$ with $Energy_{VC}(j, s)$, which is the optimal expected energy to successfully execute the tasks T_1 to T_j. Moreover, instead of $Time_V(i, j, s)$, we use $Energy_V(i, j, s)$ and replace $T_V^{SF}(i, j, l_c, s)$ with $E_V^{SF}(i, j, l_c, s)$. The goal is to find $Energy_{VC}(n, s)$, and the two dynamic programs can be correspondingly formulated as:

$$Energy_{VC}(j) = \min_{0 \le i < j}\{Energy_{VC}(i, s) + Energy_V(i + 1, j, s)\} + E_j^C \ ,$$

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

In particular, $E_V^{SF}(i, j, l_c, s)$ has similar construction as $T_V^{SF}(i, j, l_c, s)$, and it denotes the expected total energy consumption to successfully execute the tasks from T_i to T_j, knowing that the last checkpoint is at T_{l_c} and there might be additional single verifications between T_{l_c+1} and T_{i-2}. When accounting for both fail-stop and silent errors, $E_V^{SF}(i, j, l_c, s)$ can be expressed as:

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

which simplifies to

$$E_V^{SF}(i, j, l_c, s) = (P_{idle} + P_{cpu}(s)) \, e^{\lambda^S(s)T_{i,j}(s)} \left(\frac{e^{\lambda^F(s)T_{i,j}(s)} - 1}{\lambda^F(s)} + V_j(s) \right)$$
$$+ \left(e^{(\lambda^F(s) + \lambda^S(s))T_{i,j}(s)} - 1 \right) \left(E_{l_c}^R + Energy_V(l_c + 1, i - 1, s) \right) \; .$$

Clearly, the complexity is the same as that of the makespan minimization algorithm (in Theorem 2). □

5 ReExecSpeed Scenario

In the REEXECSPEED scenario, we are given two CPU speeds s and σ, where s is the regular speed and σ is the adjusted speed. The regular speed s is used for the first execution of the tasks, while σ is used to speed up or to slow down any potential re-execution in order to improve makespan or energy. We always account for both *silent and fail-stop errors*.

In this scenario, and for both VC-ONLY and VC+V approaches, we need to derive two independent expressions to compute the expected execution time (or energy consumption). The first expression is for the first execution of the tasks with the first speed until the first error is encountered. Once the first error strikes, we recover from the last checkpoint and start re-executing the tasks with the second speed until we reach the next checkpoint. This latter expression for either the time or energy is essentially the same as that in the SINGLESPEED scenario, but with speed σ.

5.1 VC-only: Using verified checkpoints only

In this approach, we aim at finding the best positions of verified checkpoints in order to minimize the makespan (TIME-VC-ONLY) or the total energy consumption (ENERGY-VC-ONLY).

5.1.1 Time-VC-Only: Minimizing Makespan

Theorem 3. *For the* REEXECSPEED *scenario, the* TIME-VC-ONLY *problem can be solved by a dynamic programming algorithm in $O(n^2)$ time.*

Proof. The proof extends that of Theorem 1 in Section 4.1.1. To account for the two speeds, we replace $Time_C(j, s)$ with $Time_{C_{re}}(j, s, \sigma)$, which denotes the optimal expected time to successfully execute the tasks T_1 to T_j, where T_j has a verified checkpoint. Similarly, we replace $T_C(i, j, s)$ with $T_{C_{re}}(i, j, s, \sigma)$ as the expected time to successfully execute the tasks T_i to T_j, where both T_{i-1} and T_j are verified and checkpointed. The goal is to find $Time_{C_{re}}(n, s, \sigma)$, and the dynamic program is formulated as:

$$Time_{C_{re}}(j, s, \sigma) = \min_{0 \le i < j} \{ Time_{C_{re}}(i, s, \sigma) + T_{C_{re}}(i + 1, j, s, \sigma) \} + C_j \; .$$

Note that the checkpointing cost after T_j is included in $Time_{C_{re}}(j, s, \sigma)$ but not in $T_{C_{re}}(i, j, s, \sigma)$. We initialize the dynamic program with $Time_{C_{re}}(0, s, \sigma) = 0$.

To compute $T_{C_{re}}(i, j, s, \sigma)$ for each (i, j) pair with $i \le j$, we need to distinguish the first execution (before the first error) and all the potential re-executions (after at least one

error). Let $T^{SF}_{C_{first}}(i,j,s)$ denote the expected time to execute the tasks T_i to T_j for the very first time, before the first error is encountered, and let $T^{SF}_C(i,j,\sigma)$ denote the expected time to successfully execute the tasks T_i to T_j in the re-executions but using speed σ. While $T^{SF}_C(i,j,\sigma)$ can be computed in the same way as in Section 4.1.1, $T^{SF}_{C_{first}}(i,j,s)$ is computed by considering two possible scenarios: either a fail-stop error has occurred during the execution from T_i to T_j, and we lose $T_{lost_{i,j}}(s)$ time as given in Equation (2), or there was no fail-stop error and we check whether a silent error has occurred. Note that in both cases we do not account for the re-executions, as they are handled by T^{SF}_C separately (with the second speed). Therefore,

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

Let $p^{\mathrm{Err}}_{i,j}(s)$ denote the probability that at least one error is detected in the first execution of the tasks from T_i to T_j at speed s. Note that we account for both *silent and fail-stop errors*, and we can only detect silent errors if no fail-stop error has occurred, thus $p^{\mathrm{Err}}_{i,j}(s) = p^F_{i,j}(s) + (1 - p^F_{i,j}(s))p^S_{i,j}(s)$. If no error strikes during the first execution, i.e., $p^{\mathrm{Err}}_{i,j}(s) = 0$, then the time to execute from T_i to T_j is exactly $T^{SF}_{C_{first}}(i,j,s)$, which means that all the tasks have been executed successfully with the first speed. If at least one error occurs with probability $p^{\mathrm{Err}}_{i,j}(s) > 0$, then $T^{SF}_{C_{first}}(i,j,s)$ is the time lost trying to execute the tasks with the first speed. In this case, we need to recover from the last checkpoint and use $T^{SF}_C(i,j,\sigma)$ to re-execute all the tasks from T_i to T_j with the second speed until we pass the next checkpoint. Therefore,

$$T_{C_{re}}(i,j,s,\sigma) = T^{SF}_{C_{first}}(i,j,s) + p^{\mathrm{Err}}_{i,j}(s)\left(R_{i-1} + T^{SF}_C(i,j,\sigma)\right) \ . \tag{3}$$

Despite the two steps needed to compute $T_{C_{re}}(i,j,s,\sigma)$, the complexity remains the same as in the SINGLESPEED scenario (Theorem 1). This concludes the proof. $\qquad\square$

5.1.2 Energy-VC-Only: Minimizing Energy

Proposition 4. *For the* REEXECSPEED *scenario, the* ENERGY-VC-ONLY *problem can be solved by a dynamic programming algorithm in $O(n^2)$ time.*

Proof. The proof is similar to that of the TIME-VC-ONLY problem presented in Section 5.1.1. We replace $Time_{C_{re}}(j,s,\sigma)$ with $Energy_{C_{re}}(j,s,\sigma)$ while accounting for both speeds. The goal is to find $Energy_{C_{re}}(n,s,\sigma)$, and the dynamic program is formulated as:

$$Energy_{C_{re}}(j,s,\sigma) = \min_{0 \leq i < j}\left\{Energy_{C_{re}}(i,s,\sigma) + E_{C_{re}}(i+1,j,s,\sigma)\right\} + E^{\mathrm{C}}_j \ .$$

Similarly to $T_{C_{re}}(i,j,s,\sigma)$, we introduce $E_{C_{re}}(i,j,s,\sigma)$, which is also composed of two parts: one for the first execution with the first speed s and the other for all potential re-executions with the second speed σ. Again, the first part only includes the energy spent on executing the tasks until the first error is encountered and it does not account for any recovery cost. As a result, the associated energy cost is only contributed by the CPU and is directly proportional to the time $T^{SF}_{C_{first}}(i,j,s)$ spent on executing the tasks in the first execution. We can express $E_{C_{re}}(i,j,s,\sigma)$ as:

$$E_{C_{re}}(i,j,s,\sigma) = (P_{idle} + P_{cpu}(s))T^{SF}_{C_{first}}(i,j,s) + p^{\mathrm{Err}}_{i,j}(s)\left(E^{\mathrm{R}}_{i-1} + E^{SF}_C(i,j,\sigma)\right) \ , \tag{4}$$

where $E^{SF}_C(i,j,\sigma)$ can be computed in the same way as in Section 4.1.2. Clearly, the complexity is the same as that of the makespan minimization algorithm (in Theorem 3). $\qquad\square$

5.2 VC+V: Using verified checkpoints and single verifications

In the VC+V approach, we can place single verifications between two verified checkpoints to achieve early detection of (silent) errors. As two speeds are used in this scenario, we will place two sets of single verifications. The first set is used during the first execution of the tasks until the first error is encountered, in which case we recover from the last checkpoint and start re-executing the tasks using the second set of verifications with the second speed. The problem is to find the best positions for the verified checkpoints as well as the best positions for the two sets of single verifications in order to minimize the total execution time (TIME-VC+V) or the total energy consumption (ENERGY-VC+V).

Because two sets of single verifications need to be placed, we formulate two independent dynamic programs to determine their respective optimal positions. The overall complexity, however, remains the same as that of 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 in $O(n^3)$ time.*

Proof. The VC+V approach follows the same reasoning as the VC-ONLY approach (see Section 5.1.1). Here, we replace $Time_{C_{re}}(j, s, \sigma)$ with $Time_{VC_{re}}(j, s, \sigma)$, and replace $T_{C_{re}}(i, j, s, \sigma)$ with $T_{VC_{re}}(i, j, s, \sigma)$. Note that both expressions follow the new re-execution model and account for both sets of single verifications. The goal is to find $Time_{VC_{re}}(n, s, \sigma)$, and the dynamic program is formulated as:

$$Time_{VC_{re}}(j, s, \sigma) = \min_{0 \leq i < j} \{Time_{VC_{re}}(i, s, \sigma) + T_{VC_{re}}(i+1, j, s, \sigma)\} + C_j .$$

Note that the checkpointing cost after T_j is included in $Time_{VC_{re}}(j, s, \sigma)$ but not in $T_{VC_{re}}(i, j, s, \sigma)$. We initialize the dynamic program with $Time_{VC_{re}}(0, s, \sigma) = 0$.

To compute $T_{VC_{re}}(i, j, s, \sigma)$, where both T_{i-1} and T_j are verified and checkpointed, we again distinguish two parts: the optimal expected time $Time_{V_{first}}(i, j, s)$ to execute the tasks T_i to T_j in the first execution using the first set of single verifications with speed s, and the optimal expected time $Time_V(i, j, \sigma)$ to successfully execute the tasks T_i to T_j in all subsequent re-executions using the second set of single verifications with speed σ. Similarly to the proof of the TIME-VC-ONLY problem in Section 5.1.1, $T_{VC_{re}}(i, j, s, \sigma)$ always includes the cost $Time_{V_{first}}(i, j, s)$ regardless of whether a error strikes during the first execution. Let $p_{i,j}^{\text{Err}}(s)$ denote the probability that at least one error is detected in the first execution, and it is again given by $p_{i,j}^{\text{Err}}(s) = p_{i,j}^F(s) + (1 - p_{i,j}^F(s))p_{i,j}^S(s)$. If an error indeed strikes during the first execution, then we need to recover from the last checkpoint and use $Time_V(i, j, \sigma)$ to re-execute all the tasks from T_i to T_j with the second speed until we pass the next checkpoint. Therefore,

$$T_{VC_{re}}(i, j, s, \sigma) = Time_{V_{first}}(i, j, s) + p_{i,j}^{\text{Err}}(s)\left(R_{i-1} + Time_V(i, j, \sigma)\right) . \tag{5}$$

Here, $Time_V(i, j, \sigma)$ follows the same dynamic programming formulation as in Section 4.2.1 but using speed σ. $Time_{V_{first}}(i, j, s)$, on the other hand, denotes the optimal expected time to execute the tasks T_i to T_j at speed s until the first error strikes. Hence, it should not include

the recovery cost nor the re-executions. The following describes a dynamic programming formulation to compute $Time_{V_{first}}(i, j, s)$:

$$Time_{V_{first}}(i, j, s) = \min_{i-1 \leq l < j} \left\{ Time_{V_{first}}(i, l, s) + (1 - p_{i,l}^{\text{Err}}(s))(T_{V_{first}}^{SF}(l+1, j, s)) \right\} ,$$

where $p_{i,l}^{\text{Err}}(s) = p_{i,l}^{F}(s) + (1 - p_{i,l}^{F}(s))p_{i,l}^{S}(s)$ is the probability that at least one error is detected when executing the tasks T_i to T_l, and $T_{V_{first}}^{SF}(i, j, s)$ denotes the expected time to execute the tasks T_i to T_j with both T_{i-1} and T_j verified. In this formulation, we include the second term only when no error has happened during the first term, otherwise we have to recover and re-execute the tasks with the second speed, which is handled by $Time_V(i, j, \sigma)$.

To compute $T_{V_{first}}^{SF}(i, j, s)$, we consider two possible scenarios: either a fail-stop error has occurred and we lose $T_{lost_{i,j}}(s)$ time, or there was no fail-stop error and we have to check whether a silent error has occurred. Recall that we do not account for the re-executions. Therefore,

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

Finally, we initialize this dynamic program with $Time_{V_{first}}(i, i-1, s) = 0$ for all $i = 1, \cdots, n$.

The complexity is dominated by the computation of $Time_V(i, j, s)$ and $Time_{V_{first}}(i, j, s)$, both of which take $O(n^3)$ time. Therefore, the overall complexity remains the same as in the SINGLESPEED scenario (Theorem 2). □

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 in* $O(n^3)$ *time.*

Proof. The proof is similar to that of the TIME-VC+V problem in Section 5.2.1. We replace $Time_{VC_{re}}(j, s, \sigma)$ with $Energy_{VC_{re}}(j, s, \sigma)$, and replace $T_{VC_{re}}(i, j, s, \sigma)$ with $E_{VC_{re}}(i, j, s, \sigma)$. Note that both expressions account for the two sets of single verifications with the corresponding speed for each set, as it was done for the makespan problem. The goal is to find $Energy_{VC_{re}}(n, s, \sigma)$, and the dynamic program is formulated as:

$$Energy_{VC_{re}}(j, s, \sigma) = \min_{0 \leq i < j} \left\{ Energy_{VC_{re}}(i, s, \sigma) + E_{VC_{re}}(i+1, j, s, \sigma) \right\} + E_j^{\text{C}} .$$

As with the minimization of the makespan, we compute $E_{VC_{re}}(i, j, s, \sigma)$ by breaking it into two parts: the optimal expected energy $Energy_{V_{first}}(i, j, s)$ to execute the tasks in the first execution and the optimal expected energy $Energy_V(i, j, \sigma)$ to successfully execute the tasks in the re-executions. Hence, analogous to Equation (5), $E_{VC_{re}}(i, j, s, \sigma)$ can be expressed as

$$E_{VC_{re}}(i, j, s, \sigma) = Energy_{V_{first}}(i, j, s) + p_{i,j}^{\text{Err}}(s) \left(E_{i-1}^{\text{R}} + Energy_V(i, j, \sigma) \right) . \tag{6}$$

Since the first execution does not include for any recovery cost, the optimal energy $Energy_{V_{first}}(i, j, s)$ spent during this time comes only from CPU and it is directly proportional to the optimal time $Time_{V_{first}}(i, j, s)$. Hence, we can express:

$$Energy_{V_{first}}(i, j, s) = (P_{idle} + P_{cpu}(s))Time_{V_{first}}(i, j, s) .$$

For the re-executions, the optimal energy $Energy_V(i, j, \sigma)$ can be obtained by following the same dynamic programming formulation as in Section 4.2.2 but using speed σ. Clearly, the complexity is the same as that of the makespan minimization algorithm (in Theorem 4). □

6   MultiSpeed Scenario

In this section, we investigate the most flexible scenario, MULTISPEED, which is built upon the REEXECSPEED scenario, to get even more control over the execution time and the energy consumption, but at the cost of a higher complexity. Instead of having two fixed speeds, we are given a set $S = \{s_1, s_2, \cdots, s_K\}$ of K discrete speeds. We call a sequence of tasks between two checkpoints a *segment* of the chain, and we allow each segment to use one speed for the first execution, and a second speed for all potential re-executions, where both speeds belong to S. The two speeds can well be different for different segments.

6.1   VC-only: Using verified checkpoints only

In this approach, we aim at finding the best positions for the checkpoints, as well as the best speed pair for each segment, in order to minimize the expected makespan (TIME-VC-ONLY) or the total energy consumption (ENERGY-VC-ONLY).

6.1.1   Time-VC-Only: Minimizing Makespan

Theorem 5. *For the* MULTISPEED *scenario, the* TIME-VC-ONLY *problem can be solved by a dynamic programming algorithm in $O(n^2K^2)$ time.*

Proof. The proof is built upon that of Theorem 3 in Section 5.1.1. Here, we use $Time_{C_{mul}}(j)$ to denote the optimal expected time to successfully execute tasks T_1 to T_j, where T_j has a verified checkpoint and there are possibly other verified checkpoints in between. Also, we use $T_{C_{mul}}(i, j)$ to denote the optimal expected time to successfully execute the tasks T_i to T_j, where both T_{i-1} and T_j are verified and checkpointed. In both expressions, the two execution speeds for each segment can be arbitrarily chosen from the discrete set S. The goal is to find $Time_{C_{mul}}(n)$, and the dynamic program can be formulated as:

$$Time_{C_{mul}}(j) = \min_{0 \leq i < j} \{Time_{C_{mul}}(i) + T_{C_{mul}}(i+1, j)\} + C_j \;,$$

which is initialized with $Time_{C_{mul}}(0) = 0$. Recall that $T_{C_{re}}(i, j, s, \sigma)$ from the REEXECSPEED scenario (see Equation (3)) already accounts for two speeds that are fixed. We can use it to compute $T_{C_{mul}}(i, j)$ by trying all possible speed pairs:

$$T_{C_{mul}}(i, j) = \min_{s, \sigma \in S} T_{C_{re}}(i, j, s, \sigma) \;.$$

The complexity is now dominated by the computation of $T_{C_{mul}}(i, j)$ for all (i, j) pairs with $i \leq j$, and it takes $O(n^2K^2)$ time. After $T_{C_{mul}}(i, j)$ is computed, the dynamic programming table can then be filled in $O(n^2)$ time. $\square$

6.1.2   Energy-VC-Only: Minimizing Energy

Proposition 6. *For the* MULTISPEED *scenario, the* ENERGY-VC-ONLY *problem can be solved by a dynamic programming algorithm in $O(n^2K^2)$ time.*

Proof. The proof is similar to that of the TIME-VC-ONLY problem in Section 6.1.1. We replace $Time_{C_{mul}}(j)$ with $Energy_{C_{mul}}(j)$ and replace $T_{C_{mul}}(i, j)$ with $E_{C_{mul}}(i, j)$. The goal

is to find $Energy_{C_{mul}}(n)$ and the dynamic program is formulated as:

$$Energy_{C_{mul}}(j) = \min_{0 \leq i < j} \{Energy_{C_{mul}}(i) + E_{C_{mul}}(i+1, j)\} + E_j^C .$$

Similarly, $E_{C_{mul}}(i, j)$ can be computed from $E_{C_{re}}(i, j, s, \sigma)$ (see Equation (4)) by trying all possible speed pairs :

$$E_{C_{mul}}(i, j) = \min_{s, \sigma \in S} E_{C_{re}}(i, j, s, \sigma) .$$

Clearly, the complexity is the same as that of the makespan minimization algorithm (in Theorem 5). □

6.2 VC+V: Using verified checkpoints and single verifications

In this approach, we aim at finding the best positions for the checkpoints and verifications, as well as the best speed pair for each segment, in order to minimize the expected makespan (Time-VC-Only) or the total energy consumption (Energy-VC-Only).

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 in* $O(n^3 K^2)$ *time.*

Proof. The proof is similar to that of the Time-VC-Only problem in Theorem 5. Here, we replace $Time_{C_{mul}}(j)$ with $Time_{VC_{mul}}(j)$ and replace $T_{C_{mul}}(i, j)$ with $T_{VC_{mul}}(i, j)$. Again, the two expressions denote the optimal execution times with the best speed pair chosen from S for each segment. The goal is to find $Time_{VC_{mul}}(n)$, and the dynamic program is formulated as:

$$Time_{VC_{mul}}(j) = \min_{0 \leq i < j} \{Time_{VC_{mul}}(i) + T_{VC_{mul}}(i+1, j)\} + C_j ,$$

which is initialized with $Time_{VC_{mul}}(0) = 0$. We can compute $T_{VC_{mul}}(i, j)$ from $T_{VC_{re}}(i, j, s, \sigma)$ (see Equation (5)) by trying all possible speed pairs:

$$T_{VC_{mul}}(i, j) = \min_{s, \sigma \in S} T_{VC_{re}}(i, j, s, \sigma) .$$

The complexity is still dominated by the computation of $T_{VC_{mul}}(i, j)$, which amounts to computing $T_{VC_{re}}(i, j, s, \sigma)$ for all (i, j) pairs and for K^2 possible pairs of speeds (see Theorem 4). Therefore, the overall complexity is $O(n^3 K^2)$. □

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 in* $O(n^3 K^2)$ *time.*

Proof. The proof follows that of the Time-VC+V problem in Section 6.2.1. We replace $Time_{VC_{mul}}(j)$ with $Energy_{VC_{mul}}(j)$ and replace $T_{VC_{mul}}(i, j)$ with $E_{VC_{mul}}(i, j)$. The goal is to find $Energy_{VC_{mul}}(n)$, and the dynamic program is formulated as:

$$Energy_{VC_{mul}}(j) = \min_{0 \leq i < j} \{Energy_{VC_{mul}}(i) + E_{VC_{mul}}(i+1, j)\} + E_j^C ,$$

where $E_{VC_{mul}}(i,j)$ is computed from $E_{VC_{re}}(i,j,s,\sigma)$ (see Equation (6)) by trying all possible speed pairs:

$$E_{VC_{mul}}(i,j) = \min_{s,\sigma \in S} E_{VC_{re}}(i,j,s,\sigma) \ .$$

Clearly, the complexity is the same as that of the makespan minimization algorithm (in 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 ≈ 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:

$$\lambda^F(s) = \lambda_{\text{ref}}^F \cdot 10^{\frac{d \cdot |s_{\text{ref}} - s|}{s_{\text{max}} - s_{\text{min}}}} \ , \tag{7}$$

where $s_{\text{ref}} \in [s_{\text{min}}, s_{\text{max}}]$ denotes the reference speed with the lowest error rate λ_{ref}^F 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 $\lambda_{\text{ref}}^F = 10^{-5}$ 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-ONLY 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-ONLY 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-ONLY 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
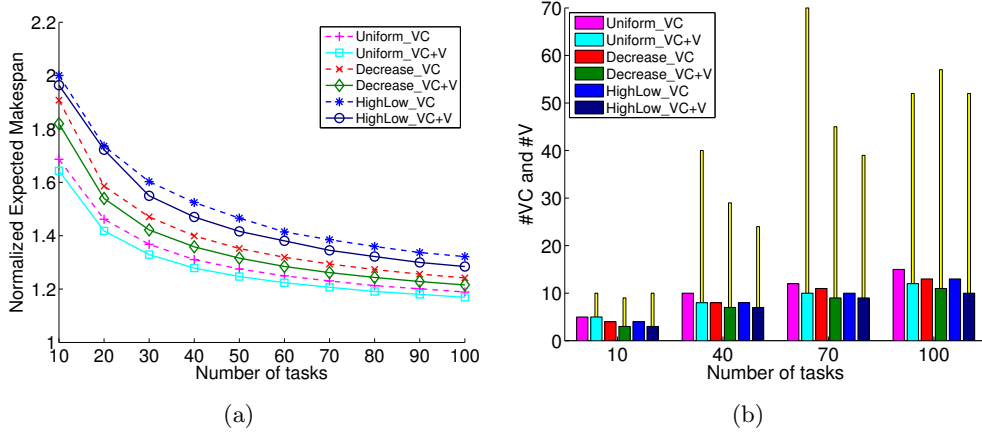
Figure 4: Impact of n and cost distribution on the performance of the TIME-VC-ONLY and TIME-VC+V algorithms. In (b), the thick bars represent the verified checkpoints and the yellow thin bars represent the total number of verifications.
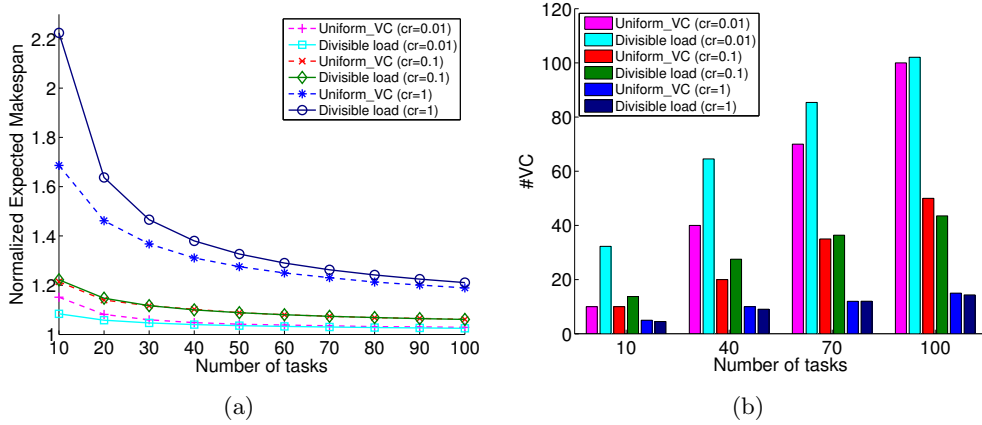

Figure 5: Performance comparison of the TIME-VC-ONLY algorithm for tasks with *Uniform* cost distribution and the optimal checkpointing algorithm for divisible load application.

(with cost $O(n)$) would give a good approximation of the optimal solution (returned by the TIME-VC-ONLY algorithm with cost $O(n^2)$), 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
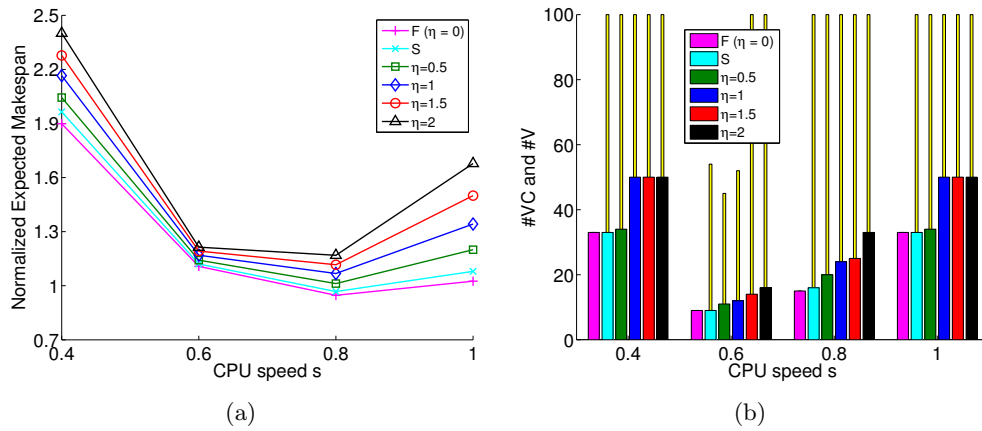
Figure 6: Impact of η and speed s on the performance. F denotes fail-stop error only and S denotes silent error only. Speed $s = 0.15$ leads to extremely large makespan, which is omitted in the figure.

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 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.
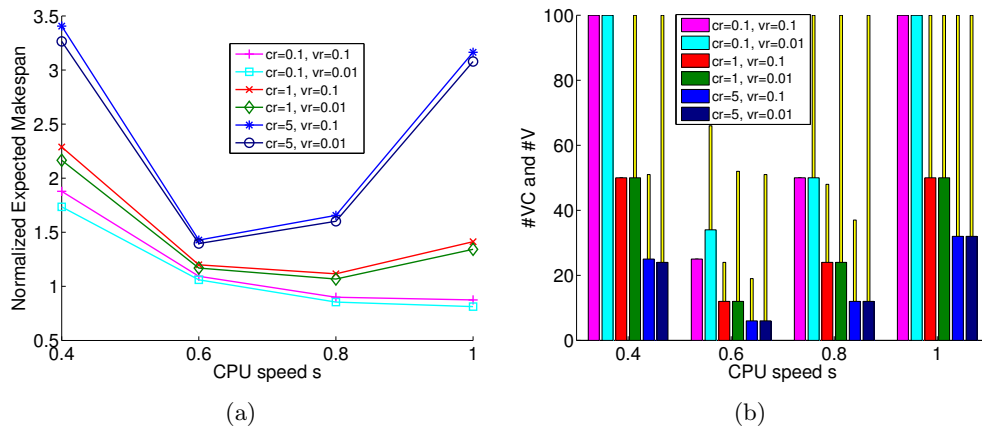
Figure 7: Impact of cr and vr on the performance with different CPU speeds.
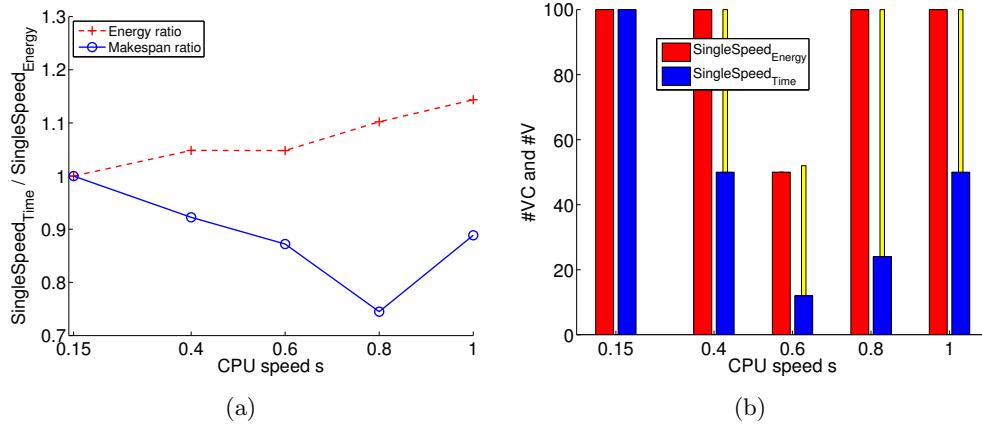
Figure 8: Relative performance of the ENERGY-VC+V and TIME-VC+V algorithms with different CPU speeds.

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 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.

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}
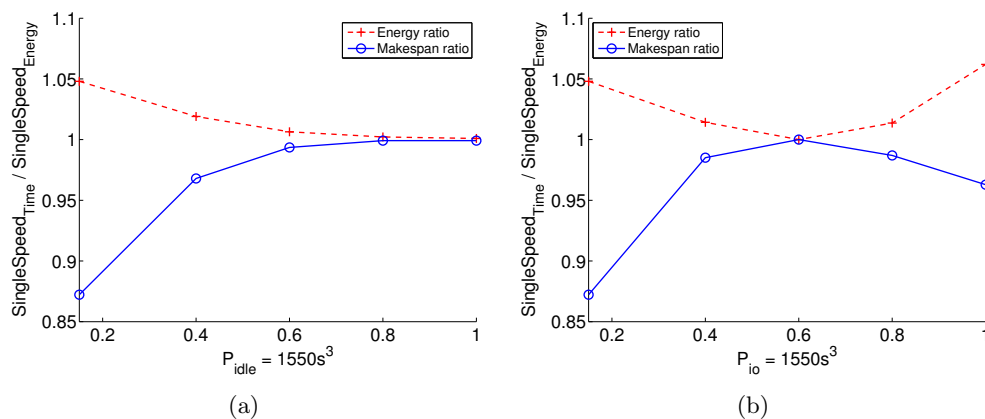
(a)   (b)

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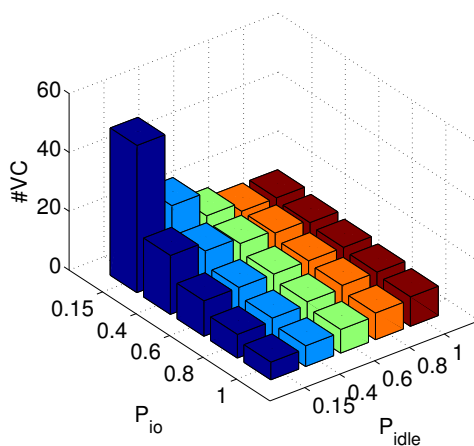

Figure 10: Number of checkpoints placed by the ENERGY-VC+V algorithm with different P_{io}, P_{idle} values $(= 1550s^3)$ at $s = 0.6$.

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.

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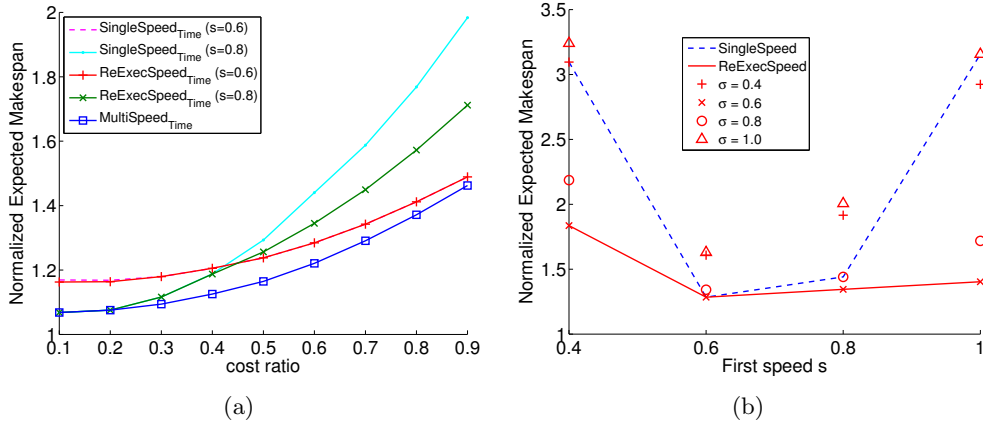
Figure 11: Performance comparison of the TIME-VC+V algorithms in MULTISPEED, REEX-
ECSPEED and SINGLESPEED scenarios for $n = 100$ tasks with *HighLow* cost distribution. In
(b), the cost ratio is 0.6.

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
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 closes the performance gap of the two algorithms as well as the
number of checkpoints placed by them. In either case, the number of checkpoints also reduces
with the cost ratio, because the total computational cost in the small tasks shrinks, thus fewer
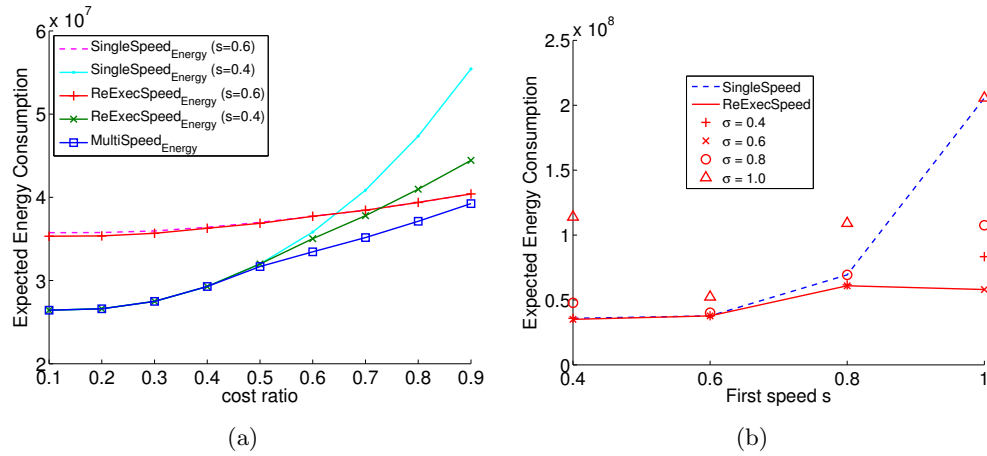checkpoints are needed among them.

(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.
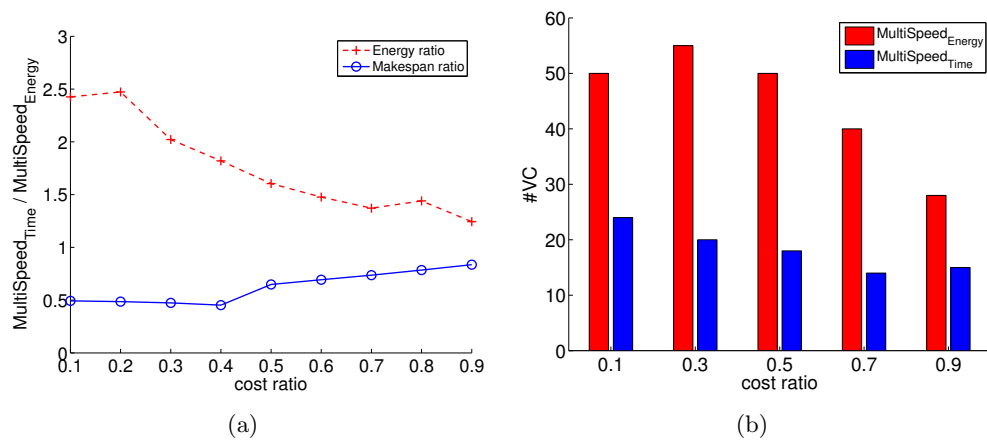

(a) (b)

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

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 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 take less than a second to find the optimal solution for $n = 100$ tasks. As application workflows rarely exceed a few tens of tasks, these algorithms could be efficiently applied in many practical contexts to determine the optimal checkpointing and verification locations.

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 (makespan) 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.