Energy-aware scheduling under reliability and makespan constraints

Guillaume Aupy*, Anne Benoit* and Yves Robert*†
*LIP, Ecole Normale Supérieure de Lyon
Email: {Guillaume.Aupy|Anne.Benoit|Yves.Robert}@ens-lyon.fr
†University of Tennessee Knoxville, USA

***Abstract*—We consider a task graph mapped on a set of homogeneous processors. We aim at minimizing the energy consumption while enforcing two constraints: a prescribed bound on the execution time (or makespan), and a reliability threshold. Dynamic voltage and frequency scaling (DVFS) is an approach frequently used to reduce the energy consumption of a schedule, but slowing down the execution of a task to save energy is decreasing the reliability of the execution. In this work, to improve the reliability of a schedule while reducing the energy consumption, we allow for the re-execution of some tasks. We assess the complexity of the tri-criteria scheduling problem (makespan, reliability, energy) of deciding which task to re-execute, and at which speed each execution of a task should be done, with two different speed models: either processors can have arbitrary speeds (CONTINUOUS model), or a processor can run at a finite number of different speeds and change its speed during a computation (VDD-HOPPING model). We propose several novel tri-criteria scheduling heuristics under the continuous speed model, and we evaluate them through a set of simulations. The two best heuristics turn out to be very efficient and complementary.**

I. INTRODUCTION

Energy-aware scheduling has proven an important issue in the past decade, both for economical and environmental reasons. This holds true for traditional computer systems, not even to speak of battery-powered systems. More precisely, a processor running at speed s dissipates s^3 watts per unit of time [1]–[3], hence it consumes $s^3 \times d$ joules when operated during d units of time. To help reduce energy dissipation, processors can run at different speeds. A widely used technique to reduce energy consumption is *dynamic voltage and frequency scaling (DVFS)*, also known as speed scaling [1]–[3]. Indeed, by lowering supply voltage, hence processor clock frequency, it is possible to achieve important reductions in power consumption; faster speeds allow for a faster execution, but they also lead to a much higher (supra-linear) power consumption. There are two popular models for processor speeds. In the CONTINUOUS model, processors can have arbitrary speeds, and can vary them continuously in the interval $[f_{\min}, f_{\max}]$. This model is unrealistic (any possible value of the speed, say $\sqrt{e^\pi}$, cannot be obtained), but it is theoretically appealing [2]. In the VDD-HOPPING model, a processor can run at a finite number of different speeds $(f_1, ..., f_m)$. It can also change its speed during a computation (*hopping* between different voltages, and hence speeds). Any rational speed can therefore be simulated [4]. The energy consumed during the execution of one task is the sum, on each time interval with constant speed f, of the energy consumed during this interval at speed f.

Energy-aware scheduling aims at minimizing the energy consumed during the execution of the target application. Obviously, this goal makes sense only when coupled with some performance bound to achieve, otherwise, the optimal solution always is to run each processor at the slowest possible speed. In this paper, we consider a directed acyclic graph (DAG) of n tasks with precedence constraints, and the goal is to schedule such an application onto a fully homogeneous platform consisting of p identical processors. This problem has been widely studied with the objective of minimizing the total execution time, or *makespan*, and it is well known to be NP-complete [5]. Since the introduction of DVFS, many papers have dealt with the optimization of energy consumption while enforcing a deadline, i.e., a bound on the makespan [1]–[3], [6].

There are many situations in which the mapping of the task graph is given, say by an ordered list of tasks to execute on each processor, and we do not have the freedom to change the assignment of a given task. Such a problem occurs when optimizing for legacy applications, or accounting for affinities between tasks and resources, or even when tasks are pre-allocated [7], for example for security reasons. While it is not possible to change the allocation of a task, it is possible to change its speed. This technique, which consists in exploiting the slack due to workload variations, is called slack reclaiming [8], [9]. In our previous work [6], assuming that the mapping and a deadline are given, we have assessed the impact of several speed variation models on the complexity of the problem of minimizing the energy consumption. Rather than using a local approach such as backfilling [9], [10], which only reclaims gaps in the schedule, we have considered the problem as a whole.

While energy consumption can be reduced by using speed scaling techniques, it was shown in [11], [12] that reducing the speed of a processor increases the number of transient fault rates of the system; the probability of failures increases exponentially, and this probability cannot be neglected in large-scale computing [13]. In order to make up for the loss in *reliability* due to the energy efficiency, different models have been proposed for fault-tolerance:

- (i) *re-execution* is the model under study in this work, and it consists in re-executing a task that does not meet the reliability constraint; it was also studied in [11], [14], [15];
- (ii) *replication* was studied in [16], [17]; this model consists in executing the same task on several processors simultaneously, in order to meet the reliability constraints;
- and (iii) *checkpointing* consists in "saving" the work done at some certain points of the work, hence reducing the amount of work lost when a failure occurs [18], [19].

This work focuses on the re-execution model, for several reasons. On the one hand, replication is too costly in terms of both resource usage and energy consumption: even if the first execution turns out successful (no failure occurred), the other executions will still have to take place. Moreover, the decision of which tasks should be replicated cannot be taken when the mapping is already fixed. On the other hand, checkpointing is hard to manage with parallel processors, and too costly if there are not too many failures. Altogether, it is the "online/no-waste" characteristic of the corresponding algorithms that lead us focus on re-execution. The goal is then to ensure that each task is reliable enough, i.e., either its execution speed is above a threshold, ensuring a given reliability of the task, or the task is executed twice to enhance its reliability. There is a clear trade-off between energy consumption and reliability, since decreasing the execution speed of a task, and hence the corresponding energy consumption, is deteriorating the reliability. This calls for tackling the problem of considering the three criteria (makespan, reliability, energy) simultaneously. This tri-criteria optimization brings dramatic complications: in addition to choosing the speed of each task, as in the deadline/energy bi-criteria problem, we also need to decide which subset of tasks should be re-executed (and then choose both execution speeds). Few authors have tackled this problem; we detail below the closest works to ours [14]–[16].

Izosinov et al. [15] study a tri-criteria optimization problem with a given mapping on heterogeneous architectures. However, they do not have any formal energy model, and they assume that the user specifies the maximum number of failures per processor tolerated to satisfy the reliability constraint, while we consider any number of failures but ensure a reliability threshold for each task. Zhu and Aydin [14] are also addressing a tri-criteria optimization problem similar to ours, and choose some tasks that have to be re-executed to match the reliability constraint. However, they restrict to the scheduling problem on one single processor, and they consider only the energy consumption of the first execution of a task (best-case scenario) when re-execution is done. Finally, Assayad et al. [16] have recently proposed an off-line tri-criteria scheduling heuristic (TSH), which uses active replication to minimize the makespan, with a threshold on the global failure rate and the maximum power consumption. TSH is an improved critical-path list scheduling heuristic that takes into account power and reliability before deciding which task to assign and to duplicate onto the next free processors. The complexity of this heuristic is unfortunately exponential in the number of processors. Future work will be devoted to compare our heuristics to TSH, and hence to compare re-execution with replication.

Given an application with dependence constraints and a mapping of this application on a homogeneous platform, we present in this paper theoretical results and tri-criteria heuristics that use re-execution in order to minimize the energy consumption under the constraints of both a reliability threshold per task and a deadline bound. The first contribution is a formal model for this tri-criteria scheduling problem (Section II). The second contribution is to provide theoretical results for the different speed models, CONTINUOUS (Section III) and VDD-HOPPING (Section IV). The third contribution is the design of novel tri-criteria scheduling heuristics that use re-execution to increase the reliability of a system under the CONTINUOUS model (Section V), and their evaluation through extensive simulations (Section VI). To the best of our knowledge, this work is the first attempt to propose practical solutions to this tri-criteria problem. Finally, we give concluding remarks and directions for future work in Section VII.

II. THE TRI-CRITERIA PROBLEM

Consider an application task graph $\mathcal{G} = (V, \mathcal{E})$, where $V = \{T_1, T_2, \ldots, T_n\}$ is the set of tasks, $n = |V|$, and where $\mathcal{E}$ is the set of precedence edges between tasks. For $1 \leq i \leq n$, task T_i has a weight w_i, that corresponds to the computation requirement of the task. We also consider particular class of task graphs, such as *linear chains* where $\mathcal{E} = \cup_{i=1}^{n-1}\{T_i \rightarrow T_{i+1}\}$, and *forks* with $n+1$ tasks $\{T_0, T_1, T_2, \ldots, T_n\}$ and $\mathcal{E} = \cup_{i=1}^{n}\{T_0 \rightarrow T_i\}$.

We assume that tasks are mapped onto a parallel platform made up of p identical processors. Each processor has a set of available speeds that is either continuous (in the interval $[f_{\min}, f_{\max}]$) or discrete (with m modes $\{f_1, \cdots, f_m\}$), depending on the speed model (CONTINUOUS or VDD-HOPPING). The goal is to minimize the energy consumed during the execution of the graph while enforcing a deadline bound and matching a reliability threshold. To match the reliability threshold, some tasks are executed once at a speed high enough to satisfy the constraint, while some other tasks need to be re-executed. We detail below the conditions that are enforced on the corresponding execution speeds. The problem is therefore to decide which task to re-execute, and at which speed to run each execution of a task.

In this section, for the sake of clarity, we assume that a task is executed at the same (unique) speed throughout execution, or at two different speeds in the case of re-execution. In Section III, we show that this strategy is indeed optimal for the CONTINUOUS model; in Section IV, we show that only two different speeds are needed for the VDD-HOPPING model (and we update the corresponding formulas accordingly). We now detail the three objective criteria (makespan, reliability, energy), and then define formally the problem.

A. Makespan

The makespan of a schedule is its total execution time. The first task is scheduled at time 0, so that the makespan of a schedule is simply the maximum time at which one of the processors finishes its computations. We consider a *deadline bound* D, which is a constraint on the makespan.

Let $\mathcal{E}xe(w_i, f)$ be the execution time of a task T_i of weight w_i at speed f. We assume that the cache size is adapted to the application, therefore ensuring that the execution time is linearly related to the frequency [18]: $\mathcal{E}xe(w_i, f) = \frac{w_i}{f}$. When a task is scheduled to be re-executed at two different speeds $f^{(1)}$ and $f^{(2)}$, we always account for both executions, even when the first execution is successful, and hence $\mathcal{E}xe(w_i, f^{(1)}, f^{(2)}) = \frac{w_i}{f^{(1)}} + \frac{w_i}{f^{(2)}}$. In other words, we consider a worst-case execution scenario, and the deadline D must be matched even in the case where all tasks that are re-executed fail during their first execution.

B. Reliability

To define the reliability, we use the fault model of Zhu et al. [11], [14]. *Transient* failures are faults caused by software errors for example. They invalidate only the execution of the current task and the processor subject to that failure will be able to recover and execute the subsequent task assigned to it (if any). In addition, we use the reliability model introduced by Shatz and Wang [20], which states that the radiation-induced transient faults follow a Poisson distribution. The parameter λ of the Poisson distribution is then:

$$\lambda(f) = \tilde{\lambda_0}\, e^{\tilde{d}\frac{f_{\max}-f}{f_{\max}-f_{\min}}}, \tag{1}$$

where $f_{\min} \leq f \leq f_{\max}$ is the processing speed, the exponent $\tilde{d} \geq 0$ is a constant, indicating the sensitivity of fault rates to DVFS, and $\tilde{\lambda_0}$ is the average fault rate corresponding to $f_{\max}$. We see that reducing the speed for energy saving increases the fault rate exponentially. The reliability of a task T_i executed once at speed f is $R_i(f) = e^{-\lambda(f)\times\mathcal{E}xe(w_i,f)}$. Because the fault rate is usually very small, of the order of 10^{-6} per time unit in [15], [21], 10^{-5} in [16], we can use the first order approximation of $R_i(f)$ as

$$\begin{aligned} R_i(f) &= 1 - \lambda(f) \times \mathcal{E}xe(w_i, f) \\ &= 1 - \lambda_0\, e^{-df} \times \frac{w_i}{f}, \end{aligned} \tag{2}$$

where $d = \frac{\tilde{d}}{f_{\max}-f_{\min}}$ and $\lambda_0 = \tilde{\lambda_0} e^{df_{\max}}$. This equation holds if $\varepsilon_i = \lambda(f) \times \frac{w_i}{f} \ll 1$. With, say, $\lambda(f) = 10^{-5}$, we need $\frac{w_i}{f} \leq 10^3$ to get an accurate approximation with $\varepsilon_i \leq 0.01$: the task should execute within 16 minutes. In other words, large (computationally demanding) tasks require reasonably high processing speeds with this model (which makes full sense in practice).

We want the reliability R_i of each task T_i to be greater than a given threshold, namely $R_i(f_{\texttt{rel}})$, hence enforcing a local constraint dependent on the task $R_i \geq R_i(f_{\texttt{rel}})$. If task T_i is executed only once at speed f, then the reliability of T_i is $R_i = R_i(f)$. Since the reliability increases with speed, we must have $f \geq f_{\texttt{rel}}$ to match the reliability constraint. If task T_i is re-executed (speeds $f^{(1)}$ and $f^{(2)}$), then the execution of T_i is successful if and only if one of the attempts do not fail, so that the reliability of T_i is $R_i = 1 - (1 - R_i(f^{(1)}))(1 - R_i(f^{(2)}))$, and this quantity should be at least equal to $R_i(f_{\texttt{rel}})$.

C. Energy

The total energy consumption corresponds to the sum of the energy consumption of each task. Let E_i be the energy consumed by task T_i. For one execution of task T_i at speed f, the corresponding energy consumption is $E_i(f) = \mathcal{E}xe(w_i, f) \times f^3 = w_i \times f^2$, which corresponds to the dynamic part of the classical energy models of the literature [1]–[3], [6]. Note that we do not take static energy into account, because all processors are up and alive during the whole execution.

If task T_i is executed only once at speed f, then $E_i = E_i(f)$. Otherwise, if task T_i is re-executed at speeds $f^{(1)}$ and $f^{(2)}$, it is natural to add up the energy consumed during both executions, just as we add up both execution times when enforcing the makespan deadline. Again, this corresponds to the worst-case execution scenario. We obtain $E_i = E_i(f_i^{(1)}) + E_i(f_i^{(2)})$. In this work, we aim at minimizing the total energy consumed by the schedule in the worst-case, assuming that all re-executions do take place. This worst-case energy is $E = \sum_{i=1}^{n} E_i$.

Some authors [14] consider only the energy spent for the first execution, which seems unfair: re-execution comes at a price both in the deadline and in the energy consumption. Another possible approach would be to consider the expected energy consumption, which would require to weight the energy spent in the second execution of a task by the probability of this re-execution to happen. This would lead to a less conservative estimation of the energy consumption by averaging over many execution instances. However, the makespan deadline should be matched in all execution scenarios, and the execution speeds of the tasks have been dimensioned to account for the worst-case scenario, so it seems more important to report for the maximal energy that can be consumed over all possible execution instances.

D. Optimization problems

The two main optimization problems are derived from the two different speed models:

- TRI-CRIT-CONT. Given an application graph $\mathcal{G} = (V, \mathcal{E})$, mapped onto p homogeneous processors with continuous speeds, TRI-CRIT-CONT is the problem of deciding which tasks should be re-executed and at which speed each execution of a task should be processed, in order to minimize the total energy consumption E, subject to the deadline bound D and to the local reliability constraints $R_i \geq R_i(f_{\texttt{rel}})$ for each $T_i \in V$.
- TRI-CRIT-VDD. This is the same problem as TRI-CRIT-CONT, but with the VDD-HOPPING model.

We also introduce variants of the problems for particular application graphs: TRI-CRIT-CONT-CHAIN is the same problem as TRI-CRIT-CONT when the task graph is a linear chain, mapped on a single processor; and TRI-CRIT-CONT-FORK is the same problem as TRI-CRIT-CONT when the task graph is a fork, and each task is mapped on a distinct processor. We have similar definitions for the VDD-HOPPING model.

III. CONTINUOUS MODEL

Due to lack of space, the formal proofs for this section can be found in the companion research report [22]. As stated in Section II, we start by proving that with the CONTINUOUS model, it is always optimal to execute a task at a unique speed throughout its execution:

Lemma 1. *With the* CONTINUOUS *model, it is optimal to execute each task at a unique speed throughout its execution.*

The idea is to consider a task whose speed changes during the execution; we exhibit a speed such that the execution time of the task remains the same, but where both energy and reliability are potentially improved, by convexity of the functions. Next we show that not only a task is executed at a single speed, but that its re-execution (whenever it occurs) is executed at the same speed as its first execution:

Lemma 2. *With the* CONTINUOUS *model, it is optimal to re-execute each task (whenever needed) at the same speed as its first execution, and this speed f is such that $f_i^{(inf)} \leq f < \frac{1}{\sqrt{2}} f_{rel}$, where*

$$\lambda_0 w_i \frac{e^{-2df_i^{(inf)}}}{(f_i^{(inf)})^2} = \frac{e^{-df_{rel}}}{f_{rel}}. \tag{3}$$

Similarly to the proof of Lemma 1, we exhibit a unique speed for both executions, in case they differ, so that the execution time remains identical but both energy and reliability are improved. If this unique speed is greater than $\frac{1}{\sqrt{2}} f_{\mathrm{rel}}$, then it is better to execute the task only once at speed f_{rel}, and if f is lower than $f_i^{(\mathrm{inf})}$, then the reliability constraint is not matched. Note that both lemmas can be applied to any solution of the TRI-CRIT-CONT problem, not just optimal solutions, hence all heuristics of Section V will assign a unique speed to each task, be it re-executed or not. We are now ready to assess the problem complexity:

Theorem 1. *The* TRI-CRIT-CONT-CHAIN *problem is NP-hard, but not known to be in NP.*

Note that the problem is not known to be in NP because speeds could take any real values (CONTINUOUS model). The completeness comes from SUBSET-SUM [23]. The problem is NP-hard even for a linear chain application mapped on a single processor (and any general DAG mapped on a single processor becomes a linear chain).

Even if TRI-CRIT-CONT-CHAIN is NP-hard, we can characterize an optimal solution of the problem:

Proposition 1. *If $f_{rel} < f_{\max}$, then in any optimal solution of* TRI-CRIT-CONT-CHAIN*, either all tasks are executed only once, at constant speed $\max(\frac{\sum_{i=1}^{n} w_i}{D}, f_{rel})$; or at least one task is re-executed, and then all tasks that are not re-executed are executed at speed f_{rel}.*

In essence, Proposition 1 states that when dealing with a linear chain, we should first slow down the execution of each task as much as possible. Then, if the deadline is not too tight, i.e., if $f_{\mathrm{rel}} > \frac{\sum_{i=1}^{n} w_i}{D}$, there remains the possibility to re-execute some of the tasks (and of course it is NP-hard to decide which ones). Still, this general principle *"first slow-down and then re-execute"* will guide the design of type A heuristics in Section V.

While the general TRI-CRIT-CONT problem is NP-hard even with a single processor, the particular variant TRI-CRIT-CONT-FORK can be solved in polynomial time:

Theorem 2. *The* TRI-CRIT-CONT-FORK *problem can be solved in polynomial time.*

The difficulty to provide an optimal algorithm for the TRI-CRIT-CONT-FORK problem comes from the fact that the total execution time must be shared between the source of the fork, T_0, and the other tasks that all run in parallel. If we know D', the fraction of the deadline allotted for tasks $T_1, \ldots, T_n$ once the source has finished its execution, then we can decide which tasks are re-executed and all execution speeds. Indeed, if task T_i is executed only once, it is executed at speed $f_i^{(once)} = \min(\max(w_i/D', f_{\mathrm{rel}}), f_{\max})$. Otherwise, it is executed twice at speed $f_i^{(twice)} = \min(\max(2w_i/D', f_{\min}, f_i^{(\mathrm{inf})}), f_{\max})$, where $f_i^{(\mathrm{inf})}$ is the minimum speed at which task T_i can be executed twice (see Lemma 2). The energy consumption for task T_i is finally $E_i = \min\left(w_i \times f_i^{(once)}, 2w_i \times f_i^{(twice)}\right)$, and the case that reaches the minimum determines whether the task is re-executed or not. There remains to find the optimal value of D', which can be obtained by studying the function of the total energy consumption, and bounding the value of D' to a small number of possibilities. Note however that this algorithm does not provide any closed-form formula for the speeds of the tasks, and that there is an intricate case analysis due to the reliability constraints.

If we further assume that the fork is made of identical tasks (i.e., $w_i = w$ for $0 \leq i \leq n$), then we can provide a closed-form formula. However, Proposition 2 illustrates the inherent difficulty of this *simple* problem, with several cases to consider depending on the values of the deadline, and also the bounds on speeds ($f_{\min}$, $f_{\max}$, f_{rel}, etc.). First, since the tasks all have the same weight $w_i = w$, we get rid of the $f_i^{(\mathrm{inf})}$ introduced above, since they are all identical (see Equation (3)): $f_i^{(\mathrm{inf})} = f^{(\mathrm{inf})}$ for $0 \leq i \leq n$. Therefore we let $f_{\min} = \max(f_{\min}, f^{(\mathrm{inf})})$ in the proposition below:

Proposition 2. *In the optimal solution of* TRI-CRIT-CONT-FORK *with at least three identical tasks (and hence $n \geq 2$), there are only three possible scenarios: (i) no task is re-executed; (ii) the n successors are all re-executed but not the source; (iii) all tasks are re-executed. In each scenario,*

the source is executed at speed f_{src} (once or twice), and the n successors are executed at the same speed f_{leaf} (once or twice).
For a deadline $D < \frac{2w}{f_{\max}}$, there is no solution. For a deadline $D \in \left[\frac{2w}{f_{\max}}, \frac{w}{f_{rel}}\frac{(1+2n^{\frac{1}{3}})^{\frac{3}{2}}}{\sqrt{1+n}}\right]$, no task is re-executed (scenario (i)) and the values of f_{src} and f_{leaf} are the following:

- *if $\frac{2w}{f_{\max}} \leq D \leq \min\left(\frac{w}{f_{\max}}(1+n^{\frac{1}{3}}), w(\frac{1}{f_{rel}}+\frac{1}{f_{\max}})\right)$, then $f_{src} = f_{\max}$ and $f_{leaf} = \frac{w}{Df_{\max}-w}f_{\max}$;*
- *if $\frac{w}{f_{\max}}(1+n^{\frac{1}{3}}) \leq w(\frac{1}{f_{rel}}+\frac{1}{f_{\max}})$, then*
  - *if $\frac{w}{f_{\max}}(1+n^{\frac{1}{3}}) < D \leq \frac{w}{f_{rel}}\frac{1+n^{\frac{1}{3}}}{n^{\frac{1}{3}}}$, then $f_{src} = \frac{w}{D}(1+n^{\frac{1}{3}})$ and $f_{leaf} = \frac{w}{D}\frac{1+n^{\frac{1}{3}}}{n^{\frac{1}{3}}}$;*
  - *if $\frac{w}{f_{rel}}\frac{1+n^{\frac{1}{3}}}{n^{\frac{1}{3}}} < D \leq \frac{2w}{f_{rel}}$, then $f_{src} = \frac{w}{Df_{rel}-w}f_{rel}$ and $f_{leaf} = f_{rel}$;*
- *if $\frac{w}{f_{\max}}(1+n^{\frac{1}{3}}) > w(\frac{1}{f_{rel}}+\frac{1}{f_{\max}})$, then*
  - *if $w(\frac{1}{f_{rel}}+\frac{1}{f_{\max}})) < D \leq \frac{2w}{f_{rel}}$, then $f_{src} = \frac{w}{Df_{rel}-w}f_{rel}$ and $f_{leaf} = f_{rel}$;*
- *if $\frac{2w}{f_{rel}} < D \leq \frac{w}{f_{rel}}\frac{(1+2n^{\frac{1}{3}})^{\frac{3}{2}}}{\sqrt{1+n}}$, then $f_{src} = f_{leaf} = f_{rel}$.*

Note that for larger values of D, depending on $f_{\min}$, we can move to scenarios (ii) and (iii) with partial or total re-execution. The case analysis becomes even more painful, but remains feasible. Intuitively, the property that all tasks have the same weight is the key to obtaining analytical formulas, because all tasks have the same minimum speed $f^{(\text{inf})}$ dictated by Equation (3). Beyond the case analysis itself, the result of Proposition 2 is interesting: we observe that in all cases, the source task is executed faster than the other tasks. This shows that Proposition 1 does not hold for general DAGs, and suggests that some tasks may be more critical than others. A hierarchical approach, that categorizes tasks with different priorities, will guide the design of type B heuristics in Section V.

IV. VDD-HOPPING MODEL

Contrarily to the CONTINUOUS model, the VDD-HOPPING model uses discrete speeds. A processor can choose among a set $\{f_1, ..., f_m\}$ of possible speeds. A task can be executed at different speeds. As for the previous section, due to lack of space, the formal proofs for this section can be found in [22].

Let $\alpha_{(i,j)}$ be the time of computation of task T_i at speed f_j. The execution time of a task T_i is $\mathcal{E}xe(T_i) = \sum_{j=1}^{m} \alpha_{(i,j)}$, and the energy consumed during the execution is $E_i = \sum_{j=1}^{m} \alpha_{(i,j)} f_j^3$. Finally, for the reliability, the approximation used in Equation (2) still holds. However, the reliability of a task is now the product of the reliabilities for each time interval with constant speed, hence $R_i = \prod_{j=1}^{m}(1 - \lambda_0\ e^{-df_j}\alpha_{(i,j)})$. Using a first order approximation, we obtain

$$R_i = 1 - \lambda_0 \sum_{j=1}^{m} e^{-df_j}\alpha_{(i,j)} = 1 - \lambda_0 \sum_{j=1}^{m} h_j \alpha_{(i,j)}, \qquad (4)$$

where $h_j = e^{-df_j}, 1 \leq j \leq m$.

We first show that only two different speeds are needed for the execution of a task. This result was already known for the bi-criteria problem makespan/energy, and it is interesting to see that reliability does not alter it:

Proposition 3. *With the* VDD-HOPPING *model, each task is computed using at most two different speeds.*

The proof is conducted by considering that a task is computed at three different speeds, and by showing that we can get rid of one of those speeds, without deteriorating any of the objective criteria. The result follows by induction. We are now ready to assess the problem complexity:

Theorem 3. *The* TRI-CRIT-VDD-CHAIN *problem is NP-complete.*

The proof is similar to that of Theorem 1, assuming that there are only two available speeds, $f_{\min}$ and $f_{\max}$. Then we reduce the problem from SUBSET-SUM. Note that here again, the problem turns out to be NP-hard even with one single processor (linear chain of tasks).

Therefore, similarly to the CONTINUOUS case, the problem is NP-hard even for a linear chain application mapped on a single processor. In the following, we propose some polynomial time heuristics to tackle the general tri-criteria problem. While these heuristics are designed for the CONTINUOUS model, they can be easily adapted to the VDD-HOPPING model thanks to Proposition 3.

V. HEURISTICS FOR TRI-CRIT-CONT

In this section, building upon the theoretical results of Section III, we propose some polynomial time heuristics for the TRI-CRIT-CONT problem, which was shown NP-hard (see Theorem 1). Recall that the mapping of the tasks onto the processors is given, and we aim at reducing the energy consumption by exploiting re-execution and speed scaling, while meeting the deadline bound and all reliability constraints.

The first idea is inspired by Proposition 1: first we search for the optimal solution of the problem instance without re-execution, a phase that we call *deceleration*: we slow down some tasks if it can save energy without violating one of the constraints. Then we refine the schedule and choose the tasks that we want to re-execute, according to some criteria. We call *type A heuristics* such heuristics that obey this general scheme: first deceleration then re-execution. Type A heuristics are expected to be efficient on a DAG with a low degree of parallelism (optimal for a chain).

However, Proposition 2 (with fork graphs) shows that it might be better to re-execute highly parallel tasks before decelerating. Therefore we introduce *type B heuristics*, which first choose the set of tasks to be re-executed, and then try to slow down the tasks that could not be re-executed. We need to find good criteria to select which tasks to re-execute, so that type B heuristics prove efficient for DAGs with a high

degree of parallelism. In summary, type B heuristics obey the opposite scheme: first re-execution then deceleration.

For both heuristic types, the approach for each phase can be sketched as follows. Initially, each task is executed once at speed $f_{\max}$. Then, let d_i be the finish time of task T_i in the current configuration.

- *Deceleration:* We select a set of tasks that we execute at speed $f_{\texttt{dec}} = \max(f_{\texttt{rel}}, \frac{\max_{i=1..n} d_i}{D} f_{\max})$, which is the slowest possible speed meeting both the reliability and deadline constraints.
- *Re-execution:* We greedily select tasks for re-execution. The selection criterion is either by decreasing weights w_i, or by decreasing *super-weights* W_i. The super-weight of a task T_i is defined as the sum of the weights of the tasks (including T_i) whose execution interval is included into T_i's execution interval. The rationale is that the super-weight of a task that we slow down is an estimation of the total amount of work that can be slowed down together with that task, hence of the energy potentially saved: this corresponds to the total slack that can be reclaimed.

We introduce further notations before listing the heuristics:

- *SUS* (Slack-Usage-Sort) is a function that sorts tasks by decreasing super-weights.
- *ReExec* is a function that tries to re-execute the current task T_i, at speed $f_{\texttt{re-ex}} = \frac{2c}{1+c} f_{\texttt{rel}}$, where $c = 4\sqrt{\frac{2}{7}} \cos\frac{1}{3}(\pi - \tan^{-1}\frac{1}{\sqrt{7}}) - 1$ (≈ 0.2838) (note that $f_{\texttt{re-ex}}$ is the optimal speed in the proof of Theorem 1). If it succeeds, it also re-executes at speed $f_{\texttt{re-ex}}$ all the tasks that are taken into account to compute the super-weight of T_i. Otherwise, it does nothing.
- *ReExec&SlowDown* performs the same re-executions as *ReExec* when it succeeds. But if the re-execution of the current task T_i is not possible, it slows down T_i as much as possible and does the same for all the tasks that are taken into account to compute the super-weight of T_i.

We now detail the heuristics:

H$f_{\max}$**.** In this heuristic, tasks are simply executed once at maximum speed.

Hno-reex. In this heuristic, there is no re-execution, and we simply consider the possible deceleration of the tasks. We set a uniform speed for all tasks, equal to $f_{\texttt{dec}}$, so that both the reliability and deadline constraints are matched. Note that heuristics **H**$f_{\max}$ and **Hno-reex** are identical except for a constant ratio on the speeds of each task, $\frac{f_{\max}}{f_{\texttt{dec}}}$. Therefore, the energy ration between both heuristics is always equal to $\left(\frac{f_{\max}}{f_{\texttt{dec}}}\right)^2$ (for instance, if $f_{\max} = 1$ and $f_{\texttt{dec}} = 2/3$, then the energy ratio is equal to 2.25).

A.Greedy. This is a type A heuristic, where we first set the speed of each task to $f_{\texttt{dec}}$ (deceleration). Let Greedy-List be the list of all the tasks sorted according to decreasing weights w_i. Each task T_i in Greedy-List is re-executed at speed $f_{\texttt{re-ex}}$ whenever possible. Finally, if there remains some slack at the end of the processing, we slow down both executions of each re-executed task as much as possible.

A.SUS-Crit. This is a type A heuristic, where we first set the speed of each task to $f_{\texttt{dec}}$. Let List-SW be the list of all tasks that belong to a critical path, sorted according to SUS. We apply ReExec to List-SW (re-execution). Finally we reclaim slack for re-executed tasks, similarly to the final step of **A.Greedy**.

B.Greedy. This is a type B heuristic. We use Greedy-List as in heuristic **A.Greedy**. We try to re-execute each task T_i of Greedy-List when possible. Then, we slow down both executions of each re-executed task T_i of Greedy-List as much as possible. Finally, we slow down the speed of each task of Greedy-List that turn out not re-executed, as much as possible.

B.SUS-Crit. This is a type B heuristic. We use List-SW as in heuristic **A.SUS-Crit**. We apply ReExec to List-SW (re-execution). Then we run Heuristic B.Greedy.

B.SUS-Crit-Slow. This is a type B heuristic. We use List-SW, and we apply ReExec&SlowDown (re-execution). Then we use Greedy-List: for each task T_i of Greedy-List, if there is enough time, we execute twice T_i at speed $f_{\texttt{re-ex}}$ (re-execution); otherwise, we execute T_i only once, at the slowest admissible speed.

Best. This is simply the minimum value over the seven previous heuristics, for reference.

The complexity of all these heuristics is bounded by $\mathrm{O}(n^4 \log n)$, where n is the number of tasks. The most time-consuming operation is the computation of List-SW (the list of all elements belonging to a critical path, sorted according to SUS).

VI. Simulations

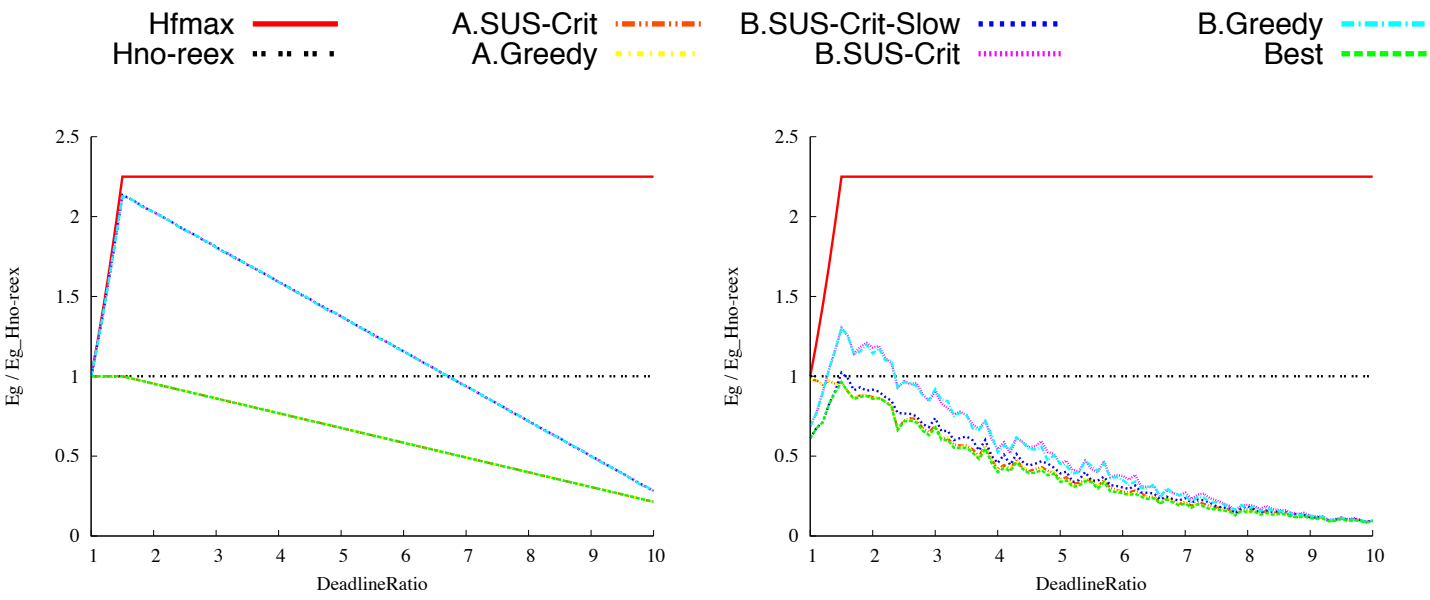
In this section, we report extensive simulations to assess the performance of the heuristics presented in Section V. The source code is publicly available at [24] (together with additional results that were omitted due to lack of space).

A. Simulation settings

In order to evaluate the heuristics, we have generated DAGs using the random DAG generation library GGEN [25]. Since GGEN does not assign a weight to the tasks of the DAGs, we use a function that gives a random float value in the interval $[0, 10]$. Each simulation uses a DAG with 100 nodes and 300 edges. We observe similar patterns for other numbers of edges, see [24] for further information.

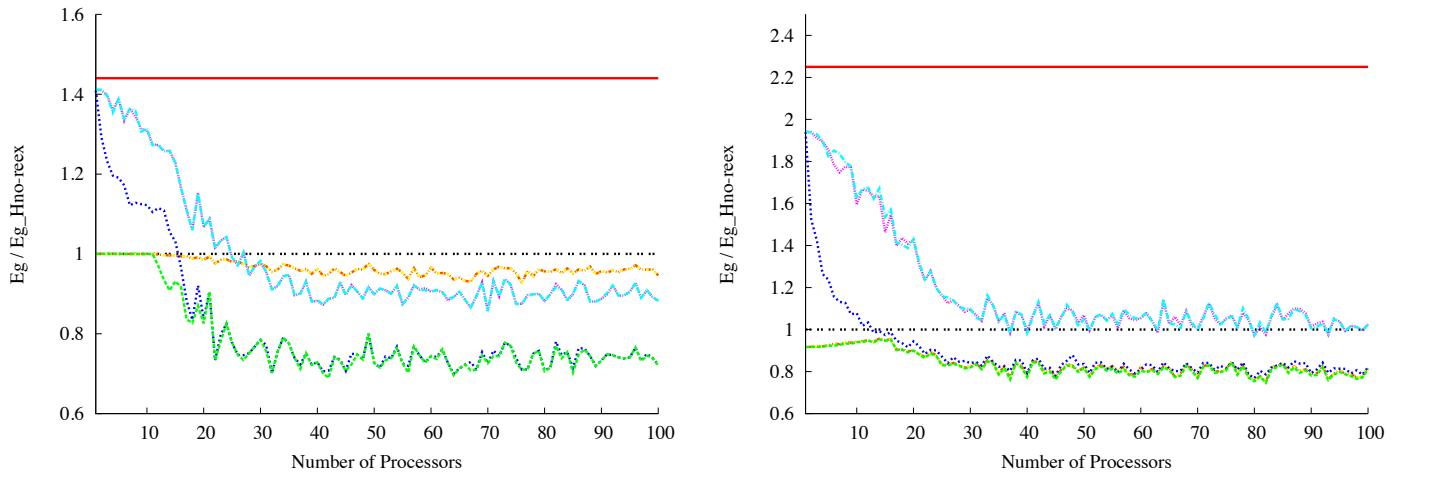
We apply a critical-path list scheduling algorithm to map the DAG onto the p processors: we assign the most urgent ready task (with largest bottom-level) to the first available processor. The bottom-level is defined as $bl(T_i) = w_i$ if T_i has no successor task, and $bl(T_i) = w_i + \max_{(T_i,T_j)\in\mathcal{E}} bl(T_j)$ otherwise.

We choose a reliability constant $\lambda_0 = 10^{-5}$ [16] (we obtain identical results with other values, see below). Each reported result is the average on ten different DAGs with the same number of nodes and edges, and the energy consumption is



(a) 1 processor (b) 50 processors

Figure 1: Comparative study when the deadline ratio varies.



(a) $D_{\texttt{ratio}}$ = 1.2 (b) $D_{\texttt{ratio}}$ = 2.4

Figure 2: Comparative study when the number of processors p varies.

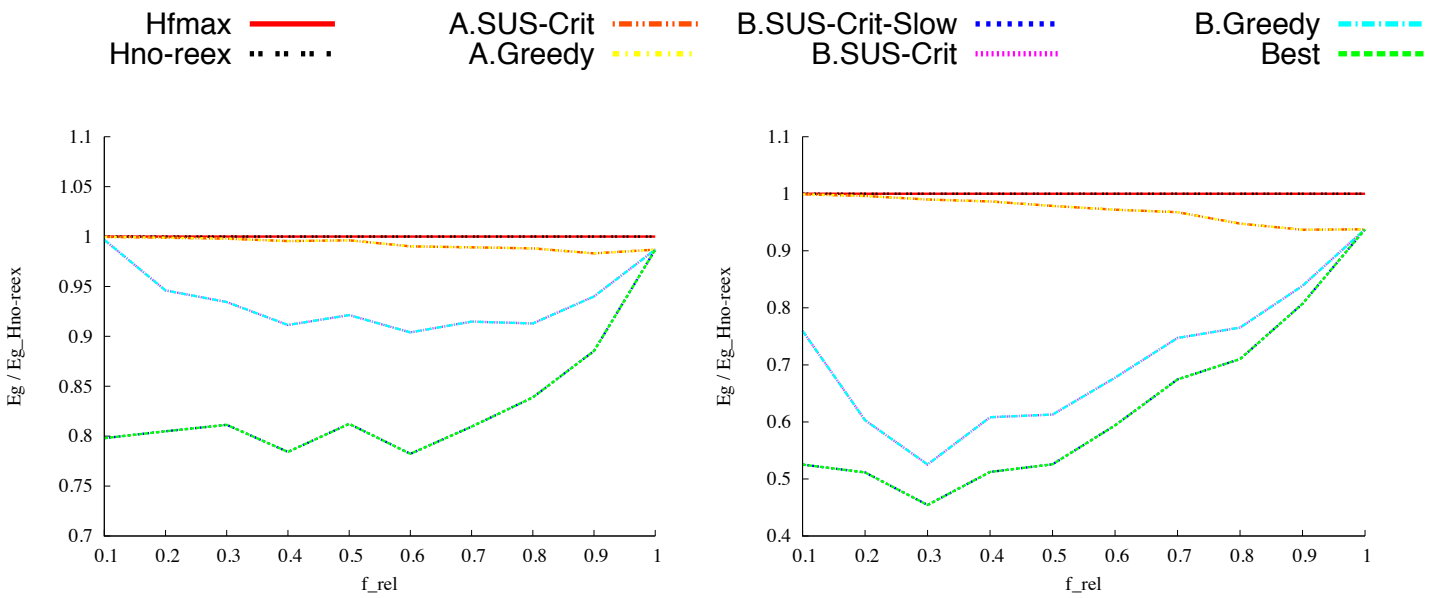
normalized with the energy consumption returned by the **Hno-reex** heuristic. If the value is lower than 1, it means that we have been able to save energy thanks to re-execution.

We analyze the influence of three different parameters: the tightness of the deadline D, the number of processors p, and the reliability speed $f_{\texttt{rel}}$. In fact, the absolute deadline D is irrelevant, and we rather consider the *deadline ratio* $D_{\texttt{ratio}} = \frac{D}{D_{\texttt{min}}}$, where $D_{\texttt{min}}$ is the execution time when executing each task once and at maximum speed $f_{\max}$ (heuristic H$f_{\max}$). Intuitively, when the deadline ratio is close to 1, there is almost no flexibility and it is difficult to re-execute tasks, while when the deadline ratio is larger we expect to be able to slow down and re-execute many tasks, thereby saving much more energy.

B. *Simulation results*

First note that with a single processor, heuristics A.SUS-Crit and A.Greedy are identical, and heuristics B.SUS-Crit and B.Greedy are identical (by definition, the only critical path is the whole set of tasks).

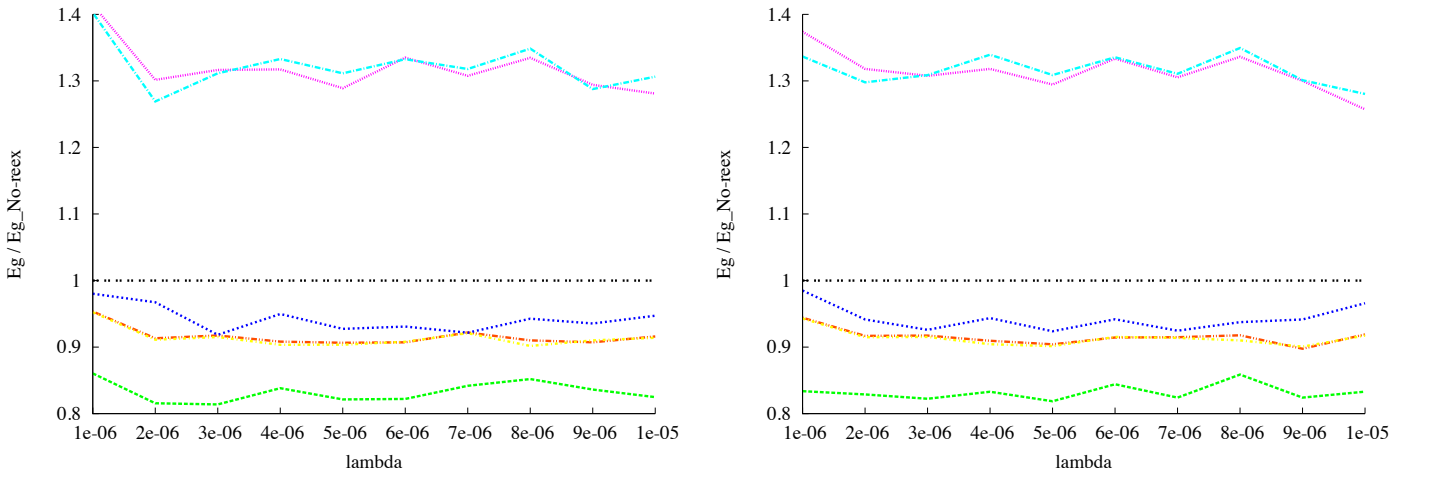
Deadline ratio. In this set of simulations, we let $p \in \{1, 10, 50, 70\}$ and $f_{\texttt{rel}} = \frac{2}{3} f_{\max}$. Figure 1 reports results for $p = 1$ and $p = 50$. When $p = 1$, we see that the results are identical for all heuristics of type A, and identical for all heuristics of type B. As expected from Proposition 1, type A



(a) $D_{\texttt{ratio}}$ = 1; 10 processors

(b) $D_{\texttt{ratio}}$ = 1; 50 processors

Figure 3: Comparative study when the reliability $f_{\texttt{rel}}$ varies.



(a) $D_{\texttt{ratio}}$ = 1.1; 50 processors

(b) $D_{\texttt{ratio}}$ = 1.5; 50 processors

Figure 4: Comparative study when λ_0 varies.

heuristics are better (see Figure 1a). With more processors (10, 50, 70), the results have the same general shape: see Figure 1b with 50 processors. When $D_{\texttt{ratio}}$ is small, type B heuristics are better. When $D_{\texttt{ratio}}$ increases up to 1.5, type A heuristics are closer to type B ones. Finally, when $D_{\texttt{ratio}}$ gets larger than 5, all heuristics converge towards the same result, where all tasks are re-executed.

Number of processors. In this set of simulations, we let $D_{\texttt{ratio}} \in \{1.2, 1.6, 2, 2.4\}$ and $f_{\texttt{rel}} = \frac{2}{3} f_{\max}$. Figure 2 confirms that type A heuristics are particularly efficient when the number of processors is small, whereas type B heuristics are at their best when the number of processors is large. Figure 2a confirms the superiority of type B heuristics for tight deadlines, as was observed in Figure 1b.

Reliability $f_{\texttt{rel}}$. In this set of simulations, we let $p \in \{1, 10, 50, 70\}$ and $D_{\texttt{ratio}} \in \{1, 1.5, 3\}$. In Figure 3, there are four different curves: the line at 1 corresponds to Hno-reex and H$f_{\max}$, then come the heuristics of type A (that all obtain exactly the same results), then B.SUS-Crit and B.Greedy that also obtain the same results, and finally the best heuristic is B.SUS-Crit-Slow. Note that B.SUS-Crit and B.Greedy return the same results because they have the same behavior when $D_{\texttt{ratio}} = 1$: there is no liberty of action on the critical

paths. However B.SUS-Crit-Slow gives better results because of the way it decelerates the important tasks that cannot be re-executed.

When $D_{\texttt{ratio}}$ is really tight (equal to 1), decreasing the value of $f_{\texttt{rel}}$ from 1 to 0.9 makes a real difference with type B heuristics. We observe an energy gain of 10% when the number of processors is small (10 in Figure 3a) and of 20% with more processors (50 in Figure 3b).

Reliability constant λ_0. In Figure 4, we let λ_0 vary from 10^{-5} to 10^{-6}, and observe very similar results throughout this range of values. Note that we did not plot H$f_{\max}$ in this figure to ease the readability.

C. Understanding the results

A.SUS-Crit and A.Greedy, and B.SUS-Crit and B.Greedy, often obtain similar results, which might lead us to underestimate the importance of critical path tasks. However, the difference between B.SUS-Crit-Slow and B.SUS-Crit shows otherwise. Tasks that belong to a critical path must be dealt with first.

A striking result is the impact of both the number of processors and the deadline ratio on the effectiveness of the heuristics. Heuristics of type A, as suggested by Proposition 1, have much better results when there is a small number of processors. When the number of processors increases, there is a difference between small and large deadline ratio. In particular, when the deadline ratio is small, heuristics of type B have better results. Indeed, heuristics of type A try to accommodate as many tasks as possible, and as a consequence, no task can be re-executed. On the contrary, heuristics of type B try to favor some tasks that are considered as important. This is highly profitable when the deadline is tight.

Note that all these heuristics take in average less than one *ms* to execute on one instance, which is very reasonable. The heuristics that compute the critical path (*.SUS-Crit-*) are the longest, and may take up to two seconds when there are few processors. Indeed, the less processors, the more edges there are in the dependence graph once the task graph is mapped, and hence it increases the complexity of finding the critical path. However, with more than ten processors, the running time never exceeds two *ms*.

Altogether we have identified two very efficient and complementary heuristics, A.SUS-Crit and B.SUS-Crit-Slow. Taking the best result out of those two heuristics always gives the best result over all simulations.

VII. Conclusion and future work

In this paper, we have accounted for the energy cost associated to task re-execution in a more realistic and accurate way than the best-case model used in [14]. Coupling this energy model with the classical reliability model used in [20], we have been able to formulate a tri-criteria optimization problem: how to minimize the energy consumed given a deadline bound and a reliability constraint? The "antagonistic" relation between speed and reliability renders this tri-criteria problem much more challenging than the standard bi-criteria (makespan, energy) version. We have stated two variants of the problem, for processor speeds obeying either the CONTINUOUS or the VDD-HOPPING model. We have assessed the intractability of this tri-criteria problem, even in the case of a single processor. In addition, we have provided several complexity results for particular instances.

We have designed and evaluated some polynomial-time heuristics for the TRI-CRIT-CONT problem that are based on the failure probability, the task weights, and the processor speeds. These heuristics aim at minimizing the energy consumption while enforcing reliability and deadline constraints. They rely on *dynamic voltage and frequency scaling* (DVFS) to decrease the energy consumption. But because DVFS lowers the reliability of the system, the heuristics use *re-execution* to compensate for the loss. After running several heuristics on a wide class of problem instances, we have identified two heuristics that are complementary, and that together are able to produce good results on most instances. The good news is that these results bring the first efficient practical solutions to the tri-criteria optimization problem, despite its theoretically challenging nature. In addition, while the heuristics do not modify the mapping of the application, it is possible to couple them with a list scheduling algorithm, as was done in the simulations, in order to solve the more general problem in which the mapping is not already given.

Future work involves several promising directions. On the theoretical side, it would be very interesting to prove a competitive ratio for the heuristic that takes the best out of A.SUS-Crit and B.SUS-Crit-Slow. However, this is quite a challenging work for arbitrary DAGs, and one may try to design approximation algorithms only for special graph structures, e.g., series-parallel graphs. However, looking back at the complicated case analysis needed for an elementary fork-graph with identical weights (Proposition 2), we cannot underestimate the difficulty of this problem.

Still on the theoretical side, it could be interesting to study the expected energy consumption, instead of the worst-case energy consumption. However, the complexity of the problems is likely to increase drastically. Consider the most simple instance that can be envisioned: a single task of weight w, a deadline D and a reliability threshold R. Let f_1 be the speed of the first execution and f_2 that of the second execution. The expected energy consumption is $w(f_1^2 + (1 - R(f_1))f_2^2)$, subject to the constraints $\frac{w}{f_1} + \frac{w}{f_2} \leq D$, and $1-(1-R(f_1))(1-R(f_2)) \geq R$. While we can (painfully) solve this instance, we will have a complicated case analysis for elementary fork-graphs with identical weights, just as in the worst-case study. As a matter of fact, Zhu et al. [11], in their pioneering work on the tri-criteria problem, briefly consider the expected energy. However they consider that every re-execution should be executed at the same speed, which leads to exactly the same schedules as those computed in this paper for the worst-case analysis.

While we have designed heuristics for the TRI-CRIT-CONT model in this paper, we could easily adapt them to the TRI-CRIT-VDD model: for a solution given by a heuristic for TRI-CRIT-CONT, if a task should be executed at the continuous speed f, then we would execute it at the two closest discrete speeds that bound f, while matching the execution time and reliability for this task. There remains to quantify the performance loss incurred by the latter constraints.

Finally, we point out that energy reduction and reliability will be even more important objectives with the advent of massively parallel platforms, made of a large number of clusters of multi-cores. More efficient solutions to the tri-criteria optimization problem (makespan, energy, reliability) could be achieved through combining replication with re-execution. A promising (and ambitious) research direction would be to search for the best trade-offs that can be achieved between these techniques that both increase reliability, but whose impact on execution time and energy consumption is very different. We believe that the comprehensive set of theoretical results and simulations given in this paper will provide solid foundations for further studies, and constitute a partial yet important first step for solving the problem at very large scale.

ACKNOWLEDGMENTS.

The authors are with Université de Lyon, France. A. Benoit and Y. Robert are with the Institut Universitaire de France. This work was supported in part by the ANR *RESCUE* project.

REFERENCES

[1] H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," in *Proc. of Int. Parallel and Distributed Processing Symposium (IPDPS)*. IEEE CS Press, 2003, pp. 113–121.

[2] N. Bansal, T. Kimbrel, and K. Pruhs, "Speed scaling to manage energy and temperature," *Journal of the ACM*, vol. 54, no. 1, pp. 1 – 39, 2007.

[3] J.-J. Chen and T.-W. Kuo, "Multiprocessor energy-efficient scheduling for real-time tasks," in *Proc. of Int. Conf. on Parallel Processing (ICPP)*. IEEE CS Press, 2005, pp. 13–20.

[4] S. Miermont, P. Vivet, and M. Renaudin, "A Power Supply Selector for Energy- and Area-Efficient Local Dynamic Voltage Scaling," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Springer Berlin / Heidelberg, 2007, vol. 4644, pp. 556–565.

[5] P. Brucker, *Scheduling Algorithms*. Springer, 2007.

[6] G. Aupy, A. Benoit, F. Dufossé, and Y. Robert, "Reclaiming the energy of a schedule, models and algorithms," *Concurrency and Computation: Practice and Experience*, 2012, also available as INRIA research report 7598 at graal.ens-lyon.fr/~abenoit.

[7] V. J. Rayward-Smith, F. W. Burton, and G. J. Janacek, "Scheduling parallel programs assuming preallocation," in *Scheduling Theory and its Applications*, P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, Eds. John Wiley and Sons, 1995.

[8] S. Lee and T. Sakurai, "Run-time voltage hopping for low-power real-time systems," in *Proc. of Annual Design Automation Conf. (DAC)*, 2000, pp. 806–809.

[9] R. B. Prathipati, "Energy efficient scheduling techniques for real-time embedded systems," Master's thesis, Texas A&M University, May 2004.

[10] L. Wang, G. von Laszewski, J. Dayal, and F. Wang, "Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS," in *Proc. of CCGrid'2010, the 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, May 2010, pp. 368 –377.

[11] D. Zhu, R. Melhem, and D. Mossé, "The effects of energy management on reliability in real-time embedded systems," in *Proc. of IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*. Washington, DC, USA: IEEE CS Press, 2004, pp. 35–40.

[12] V. Degalahal, L. Li, V. Narayanan, M. Kandemir, and M. J. Irwin, "Soft errors issues in low-power caches," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, pp. 1157–1166, October 2005.

[13] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam, "Fault-aware job scheduling for BlueGene/L systems," in *Proc. of Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2004, pp. 64–73.

[14] D. Zhu and H. Aydin, "Energy management for real-time embedded systems with reliability requirements," in *Proc. of IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, 2006, pp. 528–534.

[15] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles, "Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems," in *Proc. of IEEE/ACM Int. Conf. on Hardware/software codesign and system synthesis (CODES+ISSS)*, 2007, pp. 233–238.

[16] I. Assayad, A. Girault, and H. Kalla, "Tradeoff exploration between reliability power consumption and execution time," in *Proc. of Conf. on Computer Safety, Reliability and Security (SAFECOMP)*. Washington, DC, USA: IEEE CS Press, 2011.

[17] A. Girault, E. Saule, and D. Trystram, "Reliability versus performance for critical applications," *J. Parallel Distrib. Comput.*, vol. 69, pp. 326–336, March 2009.

[18] R. Melhem, D. Mosse, and E. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Trans. on Computers*, vol. 53, p. 2004, 2003.

[19] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *Proc. of Conf. on Design, Automation and Test in Europe (DATE)*. IEEE CS Press, 2003, p. 10918.

[20] S. M. Shatz and J.-P. Wang, "Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems," *IEEE Transactions on Reliability*, vol. 38, pp. 16–27, 1989.

[21] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, "Fault-tolerant platforms for automotive safety-critical applications," in *Proc. of Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*. ACM Press, 2003, pp. 170–177.

[22] G. Aupy, A. Benoit, and Y. Robert, "Energy-aware scheduling under reliability and makespan constraint," INRIA, France, Research Report 7757, Feb. 2012, available at graal.ens-lyon.fr/~abenoit.

[23] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[24] G. Aupy, "Source code and data." 2012. [Online]. Available: http://gaupy.org/tri-criteria-scheduling.

[25] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proc. of 3rd Int. ICST Conf. on Simulation Tools and Techniques (SIMUTools 2010)*, mar 2010, p. 10.