# Energy-Aware Algorithms for Task Graph Scheduling, Replica Placement and Checkpoint Strategies

Guillaume Aupy, Anne Benoit, Paul Renaud-Goud, Yves Robert

# Energy-aware algorithms for task graph scheduling, replica placement and checkpoint strategies

Guillaume Aupy[1], Anne Benoit[1,2], Paul Renaud-Goud[3] and Yves Robert[1,2,4]

1. LIP, ENS Lyon, France
2. Institut Universitaire de France
3. LaBRI, Bordeaux, France
4. University Tennessee Knoxville, USA

January 2014

## 1 Introduction

The *energy consumption* of computational platforms has recently become a critical problem, both for economic and environmental reasons [35]. To reduce energy consumption, processors can run at different speeds. Faster speeds allow for a faster execution, but they also lead to a much higher (superlinear) power consumption. Energy-aware scheduling aims at minimizing the energy consumed during the execution of the target application, both for computations and for communications. The price to pay for a lower energy consumption usually is a much larger execution time, so the energy-aware approach makes better sense when coupled with some prescribed performance bound. In other words, we have a bi-criteria optimization problem, with one objective being energy minimization, and the other being performance-related.

In this chapter, we discuss several problems related to data centers, for which energy consumption is a crucial matter. Indeed, statistics showed that in 2012, some data centers consume more electricity than 250 000 european houses. If the *cloud* was a country, it would be ranked as the fifth world-wide rank in terms of demands in electricity, and the need is expected to be multiplied by three before 2020. We focus mainly on the energy consumption of processors, although a lot of electricity is now devoted to cooling the machines, and also for network communications.

Energy models are introduced in Section 2. Depending on the different research areas, several different energy models are considered, but they all share the

same core assumption: there is a static energy consumption, which is independent on the speed at which a processor is running, and a dynamic energy consumption, which increases superlinearly with the speed. The most common models for speeds are either to use continuous speeds in a given interval, or to consider a set of discrete speeds (the latter being more realistic for actual processors). We discuss further variants of the discrete model: in the VDD-hopping model, the speed of a task can be changed during execution, hence allowing to simulate the continuous case; the incremental model is similar to the discrete model with the additional assumption that the different speeds are spaced regularly. Finally, we propose a literature survey on energy models, and we provide an example to compare models.

The first case study is about task graph scheduling (see Section 3). We consider a task graph to be executed on a set of processors. We assume that the mapping is given, say by an ordered list of tasks to execute on each processor, and we aim at optimizing the energy consumption while enforcing a prescribed bound on the execution time. While it is not possible to change the allocation of a task, it is possible to change its speed. Rather than using a local approach such as backfilling, we consider the problem as a whole and study the impact of several speed variation models on its complexity. For continuous speeds, we give a closed-form formula for trees and series-parallel graphs, and we cast the problem into a geometric programming problem for general directed acyclic graphs. We show that the classical dynamic voltage and frequency scaling (DVFS) model with discrete speeds leads to an NP-complete problem, even if the speeds are regularly distributed (an important particular case in practice, which we analyze as the incremental model). On the contrary, the VDD-hopping model leads to a polynomial solution. Finally, we provide an approximation algorithm for the incremental model, which we extend for the general DVFS model.

Then in Section 4, we discuss a variant of the replica placement problem aiming at an efficient power management. We study optimal strategies to place replicas in tree networks, with the double objective to minimize the total cost of the servers, and/or to optimize power consumption. The client requests are known beforehand, and some servers are assumed to pre-exist in the tree. Without power consumption constraints, the total cost is an arbitrary function of the number of existing servers that are reused, and of the number of new servers. Whenever creating and operating a new server has higher cost than reusing an existing one (which is a very natural assumption), cost optimal strategies have to trade-off between reusing resources and load-balancing requests on new servers. We provide an optimal dynamic programming algorithm that returns the optimal cost, thereby extending known results from Wu, Lin and Liu [43, 33] without pre-existing servers. With power consumption constraints, we assume that servers operate under a set of $M$ different speeds depending upon the number of requests that they have to process. In practice,

$M$ is a small number, typically 2 or 3, depending upon the number of allowed voltages [24, 23]. Power consumption includes a static part, proportional to the total number of servers, and a dynamic part, proportional to a constant exponent of the server speed, which depends upon the model for power. The cost function becomes a more complicated function that takes into account reuse and creation as before, but also upgrading or downgrading an existing server from one speed to another. We show that with an arbitrary number of speeds, the power minimization problem is NP-complete, even without cost constraint, and without static power. Still, we provide an optimal dynamic programming algorithm that returns the minimal power, given a threshold value on the total cost; it has exponential complexity in the number of speeds $M$, and its practical usefulness is limited to small values of $M$. However, experiments conducted with this algorithm show that it can process large trees in reasonable time, despite its worst-case complexity.

The last case study investigates checkpointing strategies (see Section 5). Nowadays, high performance computing is facing a major challenge with the increasing frequency of failures [18]. There is a need to use fault tolerance or resilience mechanisms to ensure the efficient progress and correct termination of the applications in the presence of failures. A well-established method to deal with failures is *checkpointing*: a checkpoint is taken at the end of the execution of each chunk of work. During the checkpoint, we check for the accuracy of the result; if the result is not correct, due to a transient failure (such as a memory error or software error), the chunk is re-executed. This model with transient failures is one of the most used in the literature, see for instance [48, 17]. In this section, we aim at minimizing the energy consumption when executing a divisible workload under a bound on the total execution time, while resilience is provided through checkpointing. We discuss several variants of this multi-criteria problem. Given the workload, we need to decide how many chunks to use, what are the sizes of these chunks, and at which speed each chunk is executed (under the continuous model). Furthermore, since a failure may occur during the execution of a chunk, we also need to decide at which speed a chunk should be re-executed in the event of a failure. The goal is to minimize the expectation of the total energy consumption, while enforcing a deadline on the execution time, that should be met either in expectation (soft deadline), or in the worst case (hard deadline). For each problem instance, we propose either an exact solution, or a function that can be optimized numerically.

Finally, we provide concluding remarks in Section 6.

## 2 Energy models

As already mentioned, to help reduce energy dissipation, processors can run at different speeds. Their power consumption is the sum of a static part (the cost for a processor to be turned on, and the leakage power) and a dynamic part, which is a strictly convex function of the processor speed, so that the execution of a given amount of work costs more power if a processor runs in a higher speed [23]. More precisely, a processor running at speed $s$ dissipates $s^3$ watts [25, 38, 12, 4, 15] per time-unit, hence consumes $s^3 \times d$ joules when operated during $d$ units of time. Faster speeds allow for a faster execution, but they also lead to a much higher (superlinear) power consumption.

In this section, we survey different models for dynamic energy consumption, taken from the literature. These models are categorized as follows:

**CONTINUOUS model.** Processors can have arbitrary speeds, and can vary them continuously within the interval $[s_{min}, s_{max}]$. This model is unrealistic (any possible value of the speed, say $\sqrt{e^{\pi}}$, cannot be obtained) but it is theoretically appealing [5]. In the CONTINUOUS model, a processor can change its speed at any time during execution.

**DISCRETE model.** Processors have a discrete number of predefined speeds, which correspond to different voltages and frequencies that the processor can be subjected to [36]. These speeds are denoted as $s_1, ..., s_m$. Switching speeds is not allowed during the execution of a given task, but two different tasks scheduled on a same processor can be executed at different speeds.

**VDD-HOPPING model.** This model is similar to the DISCRETE one, with a set of different speeds $s_1, ..., s_m$, except that switching speeds during the execution of a given task is allowed: any rational speed can be simulated, by simply switching, at the appropriate time during the execution of a task, between two consecutive speeds [34]. In the VDD-HOPPING model, the energy consumed during the execution of one task is the sum, on each time interval with constant speed $s$, of the energy consumed during this interval at speed $s$.

**INCREMENTAL model.** In this variant of the DISCRETE model, there is a value $\delta$ that corresponds to the minimum permissible speed increment, induced by the minimum voltage increment that can be achieved when controlling the processor CPU. Hence, possible speed values are obtained as $s = s_{min} + i \times \delta$, where $i$ is an integer such that $0 \leq i \leq \frac{s_{max} - s_{min}}{\delta}$. Admissible speeds lie in the interval $[s_{min}, s_{max}]$. This model aims at capturing a realistic version of the DISCRETE model, where the different speeds are spread regularly

between $s_1 = s_{min}$ and $s_m = s_{max}$, instead of being arbitrarily chosen. It is intended as the modern counterpart of a potentiometer knob.

After the literature survey in Section 2.1, we provide a simple example in Section 2.2, in order to illustrate the different models.

## 2.1 Literature survey

Reducing the energy consumption of computational platforms is an important research topic, and many techniques at the process, circuit design, and micro-architectural levels have been proposed [32, 30, 22]. The dynamic voltage and frequency scaling (DVFS) technique has been extensively studied, since it may lead to efficient energy/performance trade-offs [26, 20, 5, 14, 29, 45, 42]. Current microprocessors (for instance, from AMD [1] and Intel [24]) allow the speed to be set dynamically. Indeed, by lowering supply voltage, hence processor clock frequency, it is possible to achieve important reductions in power consumption, without necessarily increasing the execution time. We first discuss different optimization problems that arise in this context, then we review energy models.

### 2.1.1 DVFS and optimization problems

When dealing with energy consumption, the most usual optimization function consists of minimizing the energy consumption, while ensuring a deadline on the execution time (i.e., a real-time constraint), as discussed in the following papers.

In [36], Okuma et al. demonstrate that voltage scaling is far more effective than the shutdown approach, which simply stops the power supply when the system is inactive. Their target processor employs just a few discretely variable voltages. De Langen and Juurlink [31] discuss leakage-aware scheduling heuristics that investigate both DVS and processor shutdown, since static power consumption due to leakage current is expected to increase significantly. Chen et al. [13] consider parallel sparse applications, and they show that when scheduling applications modeled by a directed acyclic graph with a well-identified critical path, it is possible to lower the voltage during non-critical execution of tasks, with no impact on the execution time. Similarly, Wang et al. [42] study the slack time for non-critical jobs, they extend their execution time and thus reduce the energy consumption without increasing the total execution time. Kim et al. [29] provide power-aware scheduling algorithms for bag-of-tasks applications with deadline constraints, based on dynamic voltage scaling. Their goal is to minimize power consumption as well as to meet the deadlines specified by application users.

For real-time embedded systems, slack reclamation techniques are used. Lee and Sakurai [32] show how to exploit slack time arising from workload variation,

thanks to a software feedback control of supply voltage. Prathipati [37] discusses techniques to take advantage of run-time variations in the execution time of tasks; the goal is to determine the minimum voltage under which each task can be executed, while guaranteeing the deadlines of each task. Then, experiments are conducted on the Intel StrongArm SA-1100 processor, which has eleven different frequencies, and the Intel PXA250 XScale embedded processor with four frequencies. In [44], the goal of Xu et al. is to schedule a set of independent tasks, given a worst case execution cycle (WCEC) for each task, and a global deadline, while accounting for time and energy penalties when the processor frequency is changing. The frequency of the processor can be lowered when some slack is obtained dynamically, typically when a task runs faster than its WCEC. Yang and Lin [45] discuss algorithms with preemption, using DVS techniques; substantial energy can be saved using these algorithms, which succeed to claim the static and dynamic slack time, with little overhead.

Since an increasing number of systems are powered by batteries, maximizing battery life also is an important optimization problem. Battery-efficient systems can be obtained with similar techniques of dynamic voltage and frequency scaling, as described by Lahiri et al. in [30]. Another optimization criterion is the energy-delay product, since it accounts for a trade-off between performance and energy consumption, as for instance discussed by Gonzalez and Horowitz in [21].

### 2.1.2 Energy models

Several energy models are considered in the literature, and they can all be categorized in one of the four models investigated in this paper, i.e., CONTINUOUS, DISCRETE, VDD-HOPPING or INCREMENTAL.

The CONTINUOUS model is used mainly for theoretical studies. For instance, Yao et al. [46], followed by Bansal et al. [5], aim at scheduling a collection of tasks (with release time, deadline and amount of work), and the solution is the time at which each task is scheduled, but also, the speed at which the task is executed. In these papers, the speed can take any value, hence following the CONTINUOUS model.

We believe that the most widely used model is the DISCRETE one. Indeed, processors have currently only a few discrete number of possible frequencies [1, 24, 36, 37]. Therefore, most of the papers discussed above follow this model. Some studies exploit the continuous model to determine the smallest frequency required to run a task, and then choose the closest upper discrete value, as for instance [37] and [47].

Recently, a new local dynamic voltage scaling architecture has been developed, based on the VDD-HOPPING model [34, 6, 7]. It was shown in [32] that significant

power can be saved by using two distinct voltages, and architectures using this principle have been developed (see for instance [28]). Compared to traditional power converters, a new design with no needs for large passives or costly technological options has been validated in a STMicroelectronics CMOS 65nm low-power technology [34].

The INCREMENTAL model was introduced in [2]. The main rationale is that future technologies may well have an increased number of possible frequencies, and these will follow a regular pattern. For instance, note that the SA-1100 processor, considered in [37], has eleven frequencies that are equidistant, i.e., they follow the INCREMENTAL model. Lee and Sakurai [32] exploit discrete levels of clock frequency as $f$, $f/2$, $f/3$, ..., where $f$ is the master (i.e., the higher) system clock frequency. This model is closer to the DISCRETE model, although it exhibits a regular pattern similarly to the INCREMENTAL model.

## 2.2 Example

Energy-aware scheduling aims at minimizing the energy consumed during the execution of the target application. Obviously, it makes better sense only if it is coupled with some performance bound to achieve. For instance, whenever static energy can be neglected, the optimal solution always is to run each processor at the slowest possible speed. In the following, we do neglect static energy and discuss how to minimize dynamic energy consumption when executing a small task graph onto processors.

Consider an application with four tasks of costs $w_1 = 3$, $w_2 = 2$, $w_3 = 1$ and $w_4 = 2$, and three precedence constraints, as shown in Figure 1. We assume that $T_1$ and $T_2$ are allocated, in this order, onto processor $P_1$, while $T_3$ and $T_4$ are allocated, in this order, on processor $P_2$. The deadline on the execution time is $D = 1.5$.

We set the minimum and maximum speeds to $s_{min} = 0$ and $s_{max} = 6$ for the CONTINUOUS model. For the DISCRETE and VDD-HOPPING models, we use the set of speeds $s_1^{(d)} = 2$, $s_2^{(d)} = 5$ and $s_3^{(d)} = 6$. Finally, for the INCREMENTAL

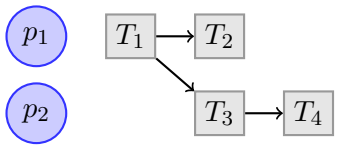



Figure 1: Execution graph for the example.

model, we set $\delta = 2$, $s_{min} = 2$ and $s_{max} = 6$, so that possible speeds are $s_1^{(i)} = 2$, $s_2^{(i)} = 4$ and $s_3^{(i)} = 6$. We aim at finding the optimal execution speed $s_i$ for each task $T_i$ ($1 \leq i \leq 4$), i.e., the values of $s_i$ that minimize the energy consumption.

With the CONTINUOUS model, the optimal speeds are non rational values, and we obtain:

$$s_1 = \frac{2}{3}(3 + 35^{1/3}) \simeq 4.18; s_2 = s_1 \times \frac{2}{35^{1/3}} \simeq 2.56; s_3 = s_4 = s_1 \times \frac{3}{35^{1/3}} \simeq 3.83.$$

Note that all speeds are in the interval $[s_{min}, s_{max}]$. These values are obtained thanks to the formulas derived in Section 3.2 below. The energy consumption is then $E_{opt}^{(c)} = \sum_{i=1}^{4} w_i \times s_i^2 = 3.s_1^2 + 2.s_2^2 + 3.s_3^2 \simeq 109.6$. The execution time is $\frac{w_1}{s_1} + \max\left(\frac{w_2}{s_2}, \frac{w_3+w_4}{s_3}\right)$, and with this solution, it is equal to the deadline $D$ (actually, both processors reach the deadline, otherwise we could slow down the execution of one task).

For the DISCRETE model, if we execute all tasks at speed $s_2^{(d)} = 5$, we obtain an energy $E = 8 \times 5^2 = 200$. A better solution is obtained with $s_1 = s_3^{(d)} = 6$, $s_2 = s_3 = s_1^{(d)} = 2$ and $s_4 = s_2^{(d)} = 5$, which turns out to be optimal: $E_{opt}^{(d)} = 3 \times 36 + (2+1) \times 4 + 2 \times 25 = 170$. Note that $E_{opt}^{(d)} > E_{opt}^{(c)}$, i.e., the optimal energy consumption with the DISCRETE model is much higher than the one achieved with the CONTINUOUS model. Indeed, in this case, even though the first processor executes during $3/6 + 2/2 = D$ time units, the second processor remains idle since $3/6 + 1/2 + 2/5 = 1.4 < D$. The problem turns out to be NP-hard (see Section 3.3.2), and the solution was found by performing an exhaustive search.

With the VDD-HOPPING model, we set $s_1 = s_2^{(d)} = 5$; for the other tasks, we run part of the time at speed $s_2^{(d)} = 5$, and part of the time at speed $s_1^{(d)} = 2$ in order to use the idle time and lower the energy consumption. $T_2$ is executed at speed $s_1^{(d)}$ during time $\frac{5}{6}$ and at speed $s_2^{(d)}$ during time $\frac{2}{30}$ (i.e., the first processor executes during time $3/5 + 5/6 + 2/30 = 1.5 = D$, and all the work for $T_2$ is done: $2 \times 5/6 + 5 \times 2/30 = 2 = w_2$). $T_3$ is executed at speed $s_2^{(d)}$ (during time $1/5$), and finally $T_4$ is executed at speed $s_1^{(d)}$ during time $0.5$ and at speed $s_2^{(d)}$ during time $1/5$ (i.e., the second processor executes during time $3/5 + 1/5 + 0.5 + 1/5 = 1.5 = D$, and all the work for $T_4$ is done: $2 \times 0.5 + 5 \times 1/5 = 2 = w_4$). This set of speeds turns out to be optimal (i.e., it is the optimal solution of the linear program introduced in Section 3.3.1), with an energy consumption $E_{opt}^{(v)} = (3/5 + 2/30 + 1/5 + 1/5) \times 5^3 + (5/6 + 0.5) \times 2^3 = 144$. As expected, $E_{opt}^{(c)} \leq E_{opt}^{(v)} \leq E_{opt}^{(d)}$, i.e., the VDD-HOPPING solution stands between the optimal CONTINUOUS solution, and the more constrained DISCRETE solution.

For the INCREMENTAL model, the reasoning is similar to the DISCRETE case, and the optimal solution is obtained by an exhaustive search: all tasks should be executed at speed $s_2^{(i)} = 4$, with an energy consumption $E_{opt}^{(i)} = 8 \times 4^2 = 128 > E_{opt}^{(c)}$. It turns out to be better than DISCRETE and VDD-HOPPING, since it has different discrete values of energy that are more appropriate for this example.

# 3 Minimizing the energy of a schedule

In this section, we investigate energy-aware scheduling strategies for executing a task graph on a set of processors. The main originality is that we assume that the mapping of the task graph is given, say by an ordered list of tasks to execute on each processor. There are many situations in which this problem is important, such as optimizing for legacy applications, or accounting for affinities between tasks and resources, or even when tasks are pre-allocated [39], for example for security reasons. In such situations, assume that a list-schedule has been computed for the task graph, and that its execution time should not exceed a deadline $D$. We do not have the freedom to change the assignment of a given task, but we can change its speed to reduce energy consumption, provided that the deadline $D$ is not exceeded after the speed change. Rather than using a local approach such as backfilling [42, 37], which only reclaims gaps in the schedule, we consider the problem as a whole, and we assess the impact of several speed variation models on its complexity. We give the main complexity results without proofs (refer to [2] for details).

## 3.1 Optimization problem

Consider an application task graph $\mathcal{G} = (V, \mathcal{E})$, with $n = |V|$ tasks denoted as $V = \{T_1, T_2, \ldots, T_n\}$, and where the set $\mathcal{E}$ denotes the precedence edges between tasks. Task $T_i$ has a cost $w_i$ for $1 \leq i \leq n$. We assume that the tasks in $\mathcal{G}$ have been allocated onto a parallel platform made up of identical processors. We define the *execution graph* generated by this allocation as the graph $G = (V, E)$, with the following augmented set of edges:

- $\mathcal{E} \subseteq E$: if an edge exists in the precedence graph, it also exists in the execution graph;
- if $T_1$ and $T_2$ are executed successively, in this order, on the same processor, then $(T_1, T_2) \in E$.

The goal is to the minimize the energy consumed during the execution while enforcing a deadline $D$ on the execution time. We formalize the optimization problem in the simpler case where each task is executed at constant speed. This strategy

is optimal for the CONTINUOUS model (by a convexity argument) and for the DISCRETE and INCREMENTAL models (by definition). For the VDD-HOPPING model, we reformulate the problem in Section 3.3.1. Let $d_i$ be the duration of the execution of task $T_i$, $t_i$ its completion time, and $s_i$ the speed at which it is executed. We obtain the following formulation of the MINENERGY$(G, D)$ problem, given an execution graph $G = (V, E)$ and a deadline $D$; the $s_i$ values are variables, whose values are constrained by the energy model:

$$\begin{array}{lll} \text{Minimize} & & \sum_{i=1}^{n} s_i^3 \times d_i \\ \text{subject to} & \text{(i)} & w_i = s_i \times d_i \text{ for each task } T_i \in V \\ & \text{(ii)} & t_i + d_j \leq t_j \text{ for each edge } (T_i, T_j) \in E \\ & \text{(iii)} & t_i \leq D \text{ for each task } T_i \in V \end{array} \tag{1}$$

Constraint (i) states that the whole task can be executed in time $d_i$ using speed $s_i$. Constraint (ii) accounts for all dependencies, and constraint (iii) ensures that the execution time does not exceed the deadline $D$. The energy consumed throughout the execution is the objective function. It is the sum, for each task, of the energy consumed by this task, as we detail in the next section. Note that $d_i = w_i/s_i$, and therefore the objective function can also be expressed as $\sum_{i=1}^{n} s_i^2 \times w_i$.

## 3.2 The CONTINUOUS model

With the CONTINUOUS model, processor speeds can take any value between $s_{min}$ and $s_{max}$. We assume for simplicity that $s_{min} = 0$, i.e., there is no minimum speed. First we prove that, with this model, the processors do not change their speed during the execution of a task:

**Lemma 1** (constant speed per task)**.** *With the* CONTINUOUS *model, each task is executed at constant speed, i.e., a processor does not change its speed during the execution of a task.*

We derive in Section 3.2.1 the optimal speed values for special execution graph structures, expressed as closed form algebraic formulas, and we show that these values may be irrational (as already illustrated in the example in Section 2.2). Finally, we formulate the problem for general DAGs as a convex optimization program in Section 3.2.2.

### 3.2.1 Special execution graphs

Consider the problem of minimizing the energy of $n$ independent tasks (i.e., each task is mapped onto a distinct processor, and there are no precedence constraints in the execution graph), while enforcing a deadline $D$.

**Proposition 1** (independent tasks)**.** *When $G$ is composed of independent tasks $\{T_1, \ldots, T_n\}$, the optimal solution to* MINENERGY$(G, D)$ *is obtained when each task $T_i$ ($1 \leq i \leq n$) is computed at speed $s_i = \frac{w_i}{D}$. If there is a task $T_i$ such that $s_i > s_{max}$, then the problem has no solution.*

Consider now the problem with a linear chain of tasks. This case corresponds for instance to $n$ independent tasks $\{T_1, \ldots, T_n\}$ executed onto a single processor. The execution graph is then a linear chain (order of execution of the tasks), with $T_i \rightarrow T_{i+1}$, for $1 \leq i < n$.

**Proposition 2** (linear chain)**.** *When $G$ is a linear chain of tasks, the optimal solution to* MINENERGY$(G, D)$ *is obtained when each task is executed at speed $s = \frac{W}{D}$, with $W = \sum_{i=1}^{n} w_i$.*
*If $s > s_{max}$, then there is no solution.*

**Corollary 1.** *A linear chain with $n$ tasks is equivalent to a single task of cost $W = \sum_{i=1}^{n} w_i$.*

Indeed, in the optimal solution, the $n$ tasks are executed at the same speed, and they can be replaced by a single task of cost $W$, which is executed at the same speed and consumes the same amount of energy.

Finally, consider fork and join graphs. Let $V = \{T_1, \ldots, T_n\}$. We consider either a fork graph $G = (V \cup \{T_0\}, E)$, with $E = \{(T_0, T_i), T_i \in V\}$, or a join graph $G = (V \cup \{T_0\}, E)$, with $E = \{(T_i, T_0), T_i \in V\}$. $T_0$ is either the source of the fork or the sink of the join.

**Theorem 1** (fork and join graphs)**.** *When $G$ is a fork (resp. join) execution graph with $n+1$ tasks $T_0, T_1, \ldots, T_n$, the optimal solution to* MINENERGY$(G, D)$ *is the following:*

- *the execution speed of the source (resp. sink) $T_0$ is $s_0 = \dfrac{\left(\sum_{i=1}^{n} w_i^3\right)^{\frac{1}{3}} + w_0}{D}$ ;*
- *for the other tasks $T_i$, $1 \leq i \leq n$, we have $s_i = s_0 \times \dfrac{w_i}{\left(\sum_{i=1}^{n} w_i^3\right)^{\frac{1}{3}}}$ if $s_0 \leq s_{max}$.*

*Otherwise, $T_0$ should be executed at speed $s_0 = s_{max}$, and the other speeds are $s_i = \frac{w_i}{D'}$, with $D' = D - \frac{w_0}{s_{max}}$, if they do not exceed $s_{max}$ (Proposition 1 for independent tasks). Otherwise there is no solution.*

*If no speed exceeds $s_{max}$, the corresponding energy consumption is*

$$\mathbf{minE}(G, D) = \frac{\left((\sum_{i=1}^{n} w_i^3)^{\frac{1}{3}} + w_0\right)^3}{D^2} .$$

**Corollary 2** (equivalent tasks for speed)**.** *Consider a fork or join graph with tasks $T_i$, $0 \leq i \leq n$, and a deadline $D$, and assume that the speeds in the optimal solution to* MINENERGY$(G, D)$ *do not exceed $s_{max}$. Then, these speeds are the same as in the optimal solution for $n+1$ independent tasks $T'_0, T'_1, \ldots, T'_n$, where $w'_0 = \left(\sum_{i=1}^{n} w_i^3\right)^{\frac{1}{3}} + w_0$, and, for $1 \leq i \leq n$, $w'_i = w'_0 \cdot \frac{w_i}{\left(\sum_{i=1}^{n} w_i^3\right)^{\frac{1}{3}}}$.*

**Corollary 3** (equivalent tasks for energy)**.** *Consider a fork or join graph $G$ and a deadline $D$, and assume that the speeds in the optimal solution to* MINENERGY$(G, D)$ *do not exceed $s_{max}$. We say that the graph $G$ is* equivalent *to the graph $G^{(eq)}$, consisting of a single task $T_0^{(eq)}$ of weight $w_0^{(eq)} = \left(\sum_{i=1}^{n} w_i^3\right)^{\frac{1}{3}} + w_0$, because the minimum energy consumption of both graphs are identical:* **minE**$(G, D)$=**minE**$(G^{(eq)}, D)$.

### 3.2.2 General DAGs

For arbitrary execution graphs, we can rewrite the MINENERGY$(G, D)$ problem as follows:

$$\begin{array}{lll} \text{Minimize} & & \sum_{i=1}^{n} u_i^{-2} \times w_i \\ \text{subject to} & \text{(i)} & t_i + w_j \times u_j \leq t_j \text{ for each edge } (T_i, T_j) \in E \\ & \text{(ii)} & t_i \leq D \text{ for each task } T_i \in V \\ & \text{(iii)} & u_i \geq \frac{1}{s_{max}} \text{ for each task } T_i \in V \end{array} \tag{2}$$

Here, $u_i = 1/s_i$ is the inverse of the speed to execute task $T_i$. We now have a convex optimization problem to solve, with linear constraints in the non-negative variables $u_i$ and $t_i$. In fact, the objective function is a posynomial, so we have a geometric programming problem (see [10, Section 4.5]) for which efficient numerical schemes exist. However, as illustrated on simple fork graphs, the optimal speeds are not expected to be rational numbers but instead arbitrarily complex expressions (we have the cubic root of the sum of cubes for forks, and nested expressions of this form for trees). From a computational complexity point of view, we do not know how to encode such numbers in polynomial size of the input (the rational task weights and the execution deadline). Still, we can always solve the problem numerically and get fixed-size numbers that are good approximations of the optimal values.

## 3.3 Discrete models

In this section, we present complexity results on the three energy models with a finite number of possible speeds. The only polynomial instance is for the VDD-HOPPING model, for which we write a linear program in Section 3.3.1. Then,

we give NP-completeness and approximation results in Section 3.3.2, for the DISCRETE and INCREMENTAL models.

### 3.3.1 The VDD-HOPPING model

**Theorem 2.** *With the* VDD-HOPPING *model,* MINENERGY$(G, D)$ *can be solved in polynomial time.*

*Proof.* Let $G$ be the execution graph of an application with $n$ tasks, and $D$ a deadline. Let $s_1, ..., s_m$ be the set of possible processor speeds. We use the following rational variables: for $1 \leq i \leq n$ and $1 \leq j \leq m$, $b_i$ is the starting time of the execution of task $T_i$, and $\alpha_{(i,j)}$ is the time spent at speed $s_j$ for executing task $T_i$. There are $n + n \times m = n(m+1)$ such variables. Note that the total execution time of task $T_i$ is $\sum_{j=1}^{m} \alpha_{(i,j)}$. The constraints are:

- $\forall 1 \leq i \leq n,\ b_i \geq 0$: starting times of all tasks are non-negative numbers;
- $\forall 1 \leq i \leq n,\ b_i + \sum_{j=1}^{m} \alpha_{(i,j)} \leq D$: the deadline is not exceeded by any task;
- $\forall 1 \leq i, i' \leq n$ such that $T_i \rightarrow T_{i'},\ t_i + \sum_{j=1}^{m} \alpha_{(i,j)} \leq t_{i'}$: a task cannot start before its predecessor has completed its execution;
- $\forall 1 \leq i \leq n,\ \sum_{j=1}^{m} \alpha_{(i,j)} \times s_j \geq w_i$: task $T_i$ is completely executed.

The objective function is then $\min \left( \sum_{i=1}^{n} \sum_{j=1}^{m} \alpha_{(i,j)} s_j^3 \right)$.

The size of this linear program is clearly polynomial in the size of the instance, all $n(m+1)$ variables are rational, and therefore it can be solved in polynomial time [40]. □

### 3.3.2 NP-completeness and approximation results

**Theorem 3.** *With the* INCREMENTAL *model (and hence the* DISCRETE *model),* MINENERGY$(G, D)$ *is NP-complete.*

Next we explain, for the INCREMENTAL and DISCRETE models, how the solution to the NP-hard problem can be approximated. Note that, given an execution graph and a deadline, the optimal energy consumption with the CONTINUOUS model is always lower than that with the other models, which are more constrained.

**Theorem 4.** *With the* INCREMENTAL *model, for any integer $K > 0$, the* MINENERGY$(G, D)$ *problem can be approximated within a factor $(1 + \frac{\delta}{s_{min}})^2 (1 + \frac{1}{K})^2$, in a time polynomial in the size of the instance and in $K$.*

**Proposition 3.**

- *For any integer $\delta > 0$, any instance of* MINENERGY$(G, D)$ *with the* CONTINUOUS *model can be approximated within a factor* $(1 + \frac{\delta}{s_{min}})^2$ *in the* INCREMENTAL *model with speed increment* $\delta$.
- *For any integer $K > 0$, any instance of* MINENERGY$(G, D)$ *with the* DISCRETE *model can be approximated within a factor* $(1 + \frac{\alpha}{s_1})^2(1 + \frac{1}{K})^2$*, with* $\alpha = \max_{1 \leq i < m}\{s_{i+1} - s_i\}$*, in a time polynomial in the size of the instance and in* $K$.

### 3.4 Final remarks

In this section, we have assessed the tractability of a classical scheduling problem, with task preallocation, under various energy models. We have given several results related to CONTINUOUS speeds. However, while these are of conceptual importance, they cannot be achieved with physical devices, and we have analyzed several models enforcing a bounded number of achievable speeds. In the classical DISCRETE model that arises from DVFS techniques, admissible speeds can be irregularly distributed, which motivates the VDD-HOPPING approach that mixes two consecutive speeds optimally. While computing optimal speeds is NP-hard with discrete speeds, it has polynomial complexity when mixing speeds. Intuitively, the VDD-HOPPING approach allows for smoothing out the discrete nature of the speeds. An alternate (and simpler in practice) solution to VDD-HOPPING is the INCREMENTAL model, where one sticks with unique speeds during task execution as in the DISCRETE model, but where consecutive speeds are regularly spaced. Such a model can be made arbitrarily efficient, according to our approximation results. Altogether, these results have laid the theoretical foundations for a comparative study of energy models.

## 4 Replica placement

In this section, we revisit the well-known replica placement problem in tree networks [16, 43, 8], with two new objectives: reusing pre-existing replicas, and enforcing an efficient power management. In a nutshell, the replica placement problem is the following: we are given a tree-shaped network where clients are periodically issuing requests to be satisfied by servers. The clients are known (both their position in the tree and their number of requests), while the number and location of the servers are to be determined. A client is a leaf node of the tree, and its requests can be served by one internal node. Note that the distribution tree (clients and nodes) is fixed in the approach. This key assumption is quite natural for a broad spectrum of applications, such as electronic, ISP, or VOD service delivery

(see [27, 16, 33] and additional references in [43]). The root server has the original copy of the database but cannot serve all clients directly, so a distribution tree is deployed to provide a hierarchical and distributed access to replicas of the original data.

In the original problem, there is no replica before execution; when a node is equipped with a replica, it can process a number of requests, up to its capacity limit. Nodes equipped with a replica, also called servers, serve all the clients located in their subtree (so that the root, if equipped with a replica, can serve any client). The rule of the game is to assign replicas to nodes so that the total number of replicas is minimized. This problem is well understood: it can be solved in time $O(N^2)$ (dynamic programming algorithm of [16]), or even in time $O(N \log N)$ (optimized greedy algorithm of [43]), where $N$ is the number of nodes.

We study in this section a more realistic model of the replica placement problem, for a dynamic setting and accounting for the energy consumption. The first contribution is to tackle the replica placement problem when the tree is equipped with pre-existing replicas before execution. This extension is a first step towards dealing with dynamic replica management: if the number and location of client requests evolve over time, the number and location of replicas must evolve accordingly, and one must decide how to perform a configuration change (at what cost?) and when (how frequently reconfigurations should occur?).

Another contribution of this section is to extend replica placement algorithms to cope with power consumption constraints. Minimizing the total power consumed by the servers has recently become a very important objective, both for economic and environmental reasons [35]. To help reduce power dissipation, processors equipped with Dynamic Voltage and Frequency Scaling technique are used, and we assume that they follow the DISCRETE model. An important result of this section is that minimizing power consumption is an NP-complete problem, independently of the incurred cost (in terms of new and pre-existing servers) of the solution. In fact, this result holds true even without pre-existing replicas, and without static power: balancing server speeds across the tree already is a hard combinatorial problem.

The cost of the best power-efficient solution may indeed be prohibitive, which calls for a bi-criteria approach: minimizing power consumption while enforcing a threshold cost that cannot be exceeded. We investigate the case where there is only a fixed number of speeds and show that there are polynomial-time algorithms capable of optimizing power for a bounded cost, even with pre-existing replicas, with static power and with a complex cost function. This result has a great practical significance, because state-of-the-art processors can only be operated with a restricted number of voltage levels, hence with a few speeds [24, 23].

Finally, we run simulations to show the practical utility of our algorithms, de-

spite their high worst-case complexity. We illustrate the impact of taking pre-existing servers into account, and how power can be saved thanks to the optimal bi-criteria algorithm.

The rest of the section is organized as follows. Section 4.1 is devoted to a detailed presentation of the target optimization problems, and provides a summary of new complexity results. The next two sections are devoted to the proofs of these results: Section 4.2 deals with computing the optimal cost of a solution, with pre-existing replicas in the tree, while Section 4.3 addresses all power-oriented problems. We report the simulation results in Section 4.4. Finally, we state some concluding remarks in Section 4.5.

## 4.1 Framework

This section is devoted to a precise statement of the problem. We start with the general problem without power consumption constraints, and next we recall the DISCRETE model of power consumption. Then we state the objective functions (with or without power), and the associated optimization problems. Finally we give a summary of all complexity results that we provide in the section.

### 4.1.1 Replica servers

We consider a distribution tree whose nodes are partitioned into a set of clients $\mathcal{C}$, and a set of $N$ nodes, $\mathcal{N}$. The clients are leaf nodes of the tree, while $\mathcal{N}$ is the set of internal nodes. Each client $i \in \mathcal{C}$ (leaf of the tree) is sending $r_i$ requests per time unit to a database object. Internal nodes equipped with a replica (also called *servers*) will process all requests from clients in their subtree. An internal node $j \in \mathcal{N}$ may have already been provided with a replica, and we let $\mathcal{E} \subseteq \mathcal{N}$ be the set of pre-existing servers. Servers in $\mathcal{E}$ will be either reused or deleted in the solution. Note that it would be easy to allow *client-server* nodes which play both the rule of a client and of a node (possibly a server), by dividing such a node into two distinct nodes in the tree.

Without power consumption constraints, the problem is to find a *solution*, i.e., a set of servers capable of handling all requests, that minimizes some cost function. We formally define a valid solution before detailing its cost. We start with some notations. Let $r$ be the root of the tree. If $j \in \mathcal{N}$, then $\mathsf{children}_j \subseteq \mathcal{N} \cup \mathcal{C}$ is the set of children of node $j$, and $\mathsf{subtree}_j \subseteq \mathcal{N} \cup \mathcal{C}$ is the subtree rooted in $j$, excluding $j$. A solution is a set $\mathcal{R} \subseteq \mathcal{N}$ of servers. Each client $i$ is assigned a single server $\mathsf{server}_i \in \mathcal{R}$ that is responsible for processing all its $r_i$ requests, and this server is restricted to be the first ancestor of $i$ (i.e., the first node in the unique path that leads from $i$ up to the root $r$) equipped with a server (hence the name *closest*

for the request service policy). Such a server must exist in $\mathcal{R}$ for each client. In addition, all servers are identical and have a limited capacity, i.e., they can process a maximum number $W$ of requests. Let $\mathsf{req}_j$ be the number of requests processed by $j \in \mathcal{R}$. The capacity constraint writes

$$\forall j \in \mathcal{R}, \;\; \mathsf{req}_j = \sum_{i \in \mathcal{C} \mid j = \mathsf{server}_i} r_i \leq W. \tag{3}$$

Now for the cost function, because all servers are identical, the cost of operating a server can be normalized to 1. When introducing a new server, there is an additional cost $\mathsf{create}$, so that running a new server costs $1 + \mathsf{create}$ while reusing a server in $\mathcal{E}$ only costs 1. There is also a deletion cost $\mathsf{delete}$ associated to deleting each server in $\mathcal{E}$ that is not reused in the solution. Let $E = |\mathcal{E}|$ be the number of pre-existing servers. Let $R = |\mathcal{R}|$ be the total number of servers in the solution, and $e = |\mathcal{R} \cap \mathcal{E}|$ be the number of reused servers. Altogether, the cost is

$$\mathsf{cost}(\mathcal{R}) = R + (R - e) \times \mathsf{create} + (E - e) \times \mathsf{delete}. \tag{4}$$

This cost function is quite general. Because of the $\mathsf{create}$ and $\mathsf{delete}$ costs, priority is always given to reusing pre-existing servers. If $\mathsf{create} + 2 \times \mathsf{delete} < 1$, priority is given to minimizing the total number of servers $R$: indeed, if this condition holds, it is always advantageous to replace two pre-existing servers by a new one (if capacities permit).

### 4.1.2 With power consumption

With power consumption constraints, we assume that servers may operate under a set $\mathcal{M} = \{W_1, \ldots, W_M\}$ of different speeds, depending upon the number of requests that they have to process per time unit. Here speeds are indexed according to increasing values, and $W_M = W$, the maximal capacity. If a server $j \in \mathcal{R}$ processes $\mathsf{req}_j$ requests, with $W_{i-1} < \mathsf{req}_j \leq W_i$, then it is operated at speed $W_i$, and we let $\mathsf{speed}(j) = i$. The power consumption of a server $j \in \mathcal{R}$ obeys the DISCRETE model

$$\mathcal{P}(j) = \mathcal{P}^{(\mathsf{static})} + W^3_{\mathsf{speed}(j)}.$$

The total power consumption $\mathcal{P}(\mathcal{R})$ of the solution is the sum of the power consumption of all server nodes:

$$\mathcal{P}(\mathcal{R}) = \sum_{j \in \mathcal{R}} \mathcal{P}(j) = R \times \mathcal{P}^{(\mathsf{static})} + \sum_{j \in \mathcal{R}} W^3_{\mathsf{speed}(j)}. \tag{5}$$

With different power speeds, it is natural to refine the cost function, and to include a cost for changing the speed of a pre-existing server (upgrading it to a

higher speed, or downgrading it to a lower speed). In the most detailed model, we would introduce $\mathsf{create}_i$, the cost for creating a new server operated at speed $W_i$, $\mathsf{changed}_{i,i'}$, the cost for changing the speed of a pre-existing server from $W_i$ to $W_{i'}$, and $\mathsf{delete}_i$, the cost for deleting a pre-existing server operated at speed $W_i$.

Note that it is reasonable to let $\mathsf{changed}_{i,i} = 0$ (no change); values of $\mathsf{changed}_{i,i'}$ with $i < i'$ correspond to upgrade costs, while values with $i' < i$ correspond to downgrade costs. In accordance with these new cost parameters, given a solution $\mathcal{R}$, we count the number of servers as follows:

- $n_i$, the number of new servers operated at speed $W_i$;
- $e_{i,i'}$, the number of reused pre-existing servers whose operation speeds have changed from $W_i$ to $W_{i'}$; and
- $k_i$, the number of pre-existing server operated at speed $W_i$ that have not been reused.

The cost of the solution $\mathcal{R}$ with a total of $R = \sum_{i=1}^{M} n_i + \sum_{i=1}^{M} \sum_{i'=1}^{M} e_{i,i'}$ servers becomes:

$$\mathsf{cost}(\mathcal{R}) = R + \sum_{i=1}^{M} \mathsf{create}_i \times n_i + \sum_{i=1}^{M} \mathsf{delete}_i \times k_i + \sum_{i=1}^{M} \sum_{i'=1}^{M} \mathsf{changed}_{i,i'} \times e_{i,i'}. \quad (6)$$

Of course, this complicated cost function can be simplified to make the model more tractable; for instance all creation costs $\mathsf{create}_i$ can be set identical, all deletion costs $\mathsf{delete}_i$ can be set identical, all upgrade and downgrade values $\mathsf{changed}_{i,i'}$ can be set identical, and the latter can even be neglected.

### 4.1.3 Objective functions

Without power consumption constraints, the objective is to minimize the cost, as defined by Equation (4). We distinguish two optimization problems, either with pre-existing replicas in the tree or without:

- MINCOST-NOPRE, the classical cost optimization problem [16] without pre-existing replicas. Indeed, in that case, Equation (4) reduces to finding a solution with the minimal number of servers.
- MINCOST-WITHPRE, the cost optimization problem with pre-existing replicas.

With power consumption constraints, the first optimization problem is MINPOWER, which stands for minimizing power consumption, independently of the incurred cost. But the cost of the best power-efficient solution may indeed be prohibitive, which calls for a bi-criteria approach: MINPOWER-BOUNDEDCOST

is the problem to minimize power consumption while enforcing a threshold cost that cannot be exceeded. This bi-criteria problem can be declined in two versions, without pre-existing replicas (MINPOWER-BOUNDEDCOST-NOPRE) and with pre-existing replicas (MINPOWER-BOUNDEDCOST-WITHPRE).

### 4.1.4 Summary of results

In this section, we prove the following complexity results for a tree with $N$ nodes:

**Theorem 5.** MINCOST-WITHPRE can be solved in polynomial time with a dynamic programming algorithm whose worst case complexity is $O(N^5)$.

**Theorem 6.** MINPOWER is NP-complete.

**Theorem 7.** With a constant number $M$ of speeds, both versions of MINPOWER-BOUNDEDCOST can be solved in polynomial time with a dynamic programming algorithm. The complexity of this algorithm is $O(N^{2M+1})$ for MINPOWER-BOUNDEDCOST-NOPRE and $O(N^{2M^2+2M+1})$ for MINPOWER-BOUNDEDCOST-WITHPRE.

Note that MINPOWER remains NP-complete without pre-existing replicas, and without static power: the proof of Theorem 6 (see Section 4.3.2) shows that balancing server speeds across the tree already is a hard combinatorial problem. On the contrary, with a fixed number of speeds, there are polynomial-time algorithms capable of optimizing power for a bounded cost, even with pre-existing replicas, with static power and with a complex cost function. These algorithms can be viewed as pseudo-polynomial solutions to the MINPOWER-BOUNDEDCOST problems.

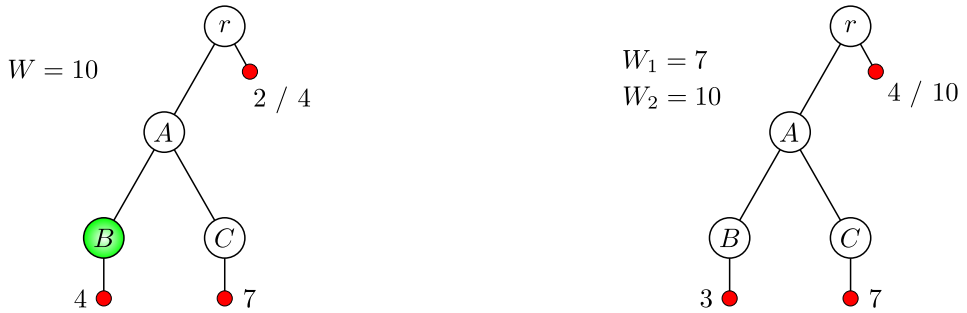
## 4.2 Complexity results: update strategies

In this section, we focus on the MINCOST-WITHPRE problem: we need to update the set of replicas in a tree, given a set of pre-existing servers, so as to minimize the cost function.

In Section 4.2.1, we show on an illustrative example that the strategies need to trade-off between reusing resources and load-balancing requests on new servers: the greedy algorithm proposed in [43] for the MINCOST-NOPRE problem is no longer optimal. We provide in Section 4.2.2 a dynamic programming algorithm that returns the optimal solution in polynomial time, and we prove its correctness.

### 4.2.1 Running example

We consider the example of Figure 2(a). There is one pre-existing replica in the tree at node $B$, and we need to decide whether to reuse it or not. For taking decisions locally at node $A$, the trade-off is the following:




(a) Reusing pre-existing replicas.

(b) Minimizing power consumption.

Figure 2: Examples

- either we keep server $B$, and there are 7 requests going up in the tree from node $A$;
- either we remove server $B$ and place a new server at node $C$, hence having only 4 requests going up in the tree from node $A$;
- either we keep the replica at node $B$ and add one at node $A$ or $C$, thereby having no traversing request any more.

The choice cannot be made locally, since it depends upon the remainder of the tree: if the root $r$ has two client requests, then it was better to keep the pre-existing server $B$. However, if it has four requests, two new servers are needed to satisfy all requests, and one can then remove server $B$ which becomes useless (i.e., keep one server at node $C$ and one server at node $r$).

From this example, it seems very difficult to design a greedy strategy to minimize the solution cost, while accounting for pre-existing replicas. We propose in the next section a dynamic programming algorithm that solves the MINCOST-WITHPRE problem.

#### 4.2.2 Dynamic programming algorithm

Let $W$ be the total number of requests that a server can handle, and $r_i$ the number of requests issued by client $i \in \mathcal{C}$.

At each node $j \in \mathcal{N}$, we fill a table of maximum size $(E+1) \times (N-E+1)$ which indicates, for exactly $0 \leq e \leq E$ existing servers and $0 \leq n \leq N-E$ new servers in the subtree rooted in $j$ (excluding $j$), the solution which leads to the minimum number of requests that have not been processed in the subtree. This

solution for $(e, n)$ values at node $j$ is characterized by the minimum number of requests that is obtained, $minr^j_{(e,n)}$, and by the number of requests processed at each node $j' \in \mathsf{subtree}_j$, $req^j_{(e,n)}(j')$. Note that each entry of the table has a maximum size $O(N)$ (in particular, this size is reached at the root of the tree). The $req$ variables ensure that it is possible to reconstruct the solution once the traversal of the tree is complete.

First, tables are initialized to default values (no solution). We set $minr^j_{(e,n)} = W + 1$ to indicate that there is no solution, because in any valid solution, we have $minr^j_{(e,n)} \leq W$. The main algorithm then fills the tables while performing a bottom-up traversal of the tree, and the solution can be found within the table of the root node. Initially, we fill the table for nodes $j$ which have only client nodes: $minr^j_{(0,0)} = \sum_{i \in \mathsf{children}_j \cap \mathcal{C}} r_i$, and $minr^j_{(k,l)} = W + 1$ for $k > 0$ or $l > 0$. There are no nodes in the subtree of $j$, thus no $req$ variables to set. The variable $client(j)$ keeps track of the number of requests directly issued by a client at node $j$. Also, recall that the decision whether to place a replica at node $j$ or not is not accounted for in the table of $j$, but when processing the parent of node $j$.

Then, for a node $j \in \mathcal{N}$, we perform the same initialization, before processing children nodes one by one. To process child $i$ of node $j$, first, we copy the current table of node $j$ into a temporary one, with values $tminr$ and $treq$. Note that the table is initially almost empty, but this copy is required since we process children one after the other, and when we merge the $k^{th}$ children node of $j$, the table of $j$ already contains information from the merge with the previous $k-1$ children nodes. Then, for $0 \leq e \leq E$ and $0 \leq n \leq N - E$, we need to compute the new $minr^j_{(e,n)}$, and to update the $req^j_{(e,n)}$ values. We try all combinations with $e'$ existing replicas and $n'$ new replicas in the temporary table (i.e., information about children already processed), $e - e'$ existing replicas and $n - n'$ new replicas in the subtree of child $i$. We furthermore try solutions with a replica placed at node $i$, and we account for it in the value of $e$ if $i \in \mathcal{E}$ (i.e., for a given value $e'$, we place only $e - e' - 1$ replica in the subtree of $i$, plus one on $i$); otherwise we account for it in the value of $n$. Each time we find a solution which is better than the one previously in the table (in terms of $minr$), we copy the values of $req$ from the temporary table and the table of $i$, in order to retain all the information about the current best solution. The key of the algorithm resides in the fact that during this *merging* process, the optimal solution will always be one which lets the minimum of requests pass through the subtree (see Lemma 2).

The solution to the replica placement problem with pre-existing servers MINCOST-WITHPRE is computed by scanning all solutions in order to return a valid one of minimum cost. To prove that the algorithm returns an optimal solution, we show first that the solutions that are discarded while filling the tables, never lead to a

better solution than the one that is finally returned:

**Lemma 2.** Consider a subtree rooted at node $j \in \mathcal{N}$. If an optimal solution uses $e$ pre-existing servers and places $n$ new servers in this subtree, then there exists an optimal solution of same cost, for which the placement of these servers minimizes the number of requests traversing $j$.

*Proof.* Let $\mathcal{R}_{opt}$ be the set of replicas in the optimal solution with $(e, n)$ servers (i.e., $e$ pre-existing and $n$ new in $\mathsf{subtree}_j$). We denote by $rmin$ the minimum number of requests that must traverse $j$ in a solution using $(e, n)$ servers, and by $\mathcal{R}_{loc}$ the corresponding (local) placement of replicas in $\mathsf{subtree}_j$.

If $\mathcal{R}_{opt}$ is such that more than $rmin$ requests are traversing node $j$, we can build a new global solution which is similar to $\mathcal{R}_{opt}$, except for the subtree rooted in $j$ for which we use the placement of $\mathcal{R}_{loc}$. The cost of the new solution is identical to the cost of $\mathcal{R}_{opt}$, therefore it is an optimal solution. It is still a valid solution, since $\mathcal{R}_{loc}$ is a valid solution and there are less requests than before to handle in the remaining of the tree (only $rmin$ requests traversing node $j$).

This proves that there exists an optimal solution which minimizes the number of requests traversing each node, given a number of pre-existing and new servers. □

The algorithm computes all local optimal solutions for all values $(e, n)$. During the merge procedure, we try all possible numbers of pre-existing and new servers in each subtree, and we minimize the number of traversing requests, thus finding an optimal local solution. Thanks to Lemma 2, we know that there is a global optimal solution which builds upon these local optimal solutions.

We can show that the execution time of this algorithm is in $O(N \times (N - E + 1)^2 \times (E + 1)^2)$, where $N$ is the total number of nodes, and $E$ is the number of pre-existing nodes. This corresponds to the $N$ calls to the merging procedure. The algorithm is therefore of polynomial complexity, at most $O(N^5)$ for a tree with $N$ nodes. This concludes the proof of Theorem 5. For a formalization of the algorithm and the details about its execution time, please refer to [9].

### 4.3 Complexity results with power

In this section, we tackle the MINPOWER and MINPOWER-BOUNDEDCOST problems. First in Section 4.3.1, we use an example to show why minimizing the number of requests traversing the root of a subtree is no longer optimal, and we illustrate the difficulty to take local decisions even when restricting to the simpler mono-criterion MINPOWER problem. Then in Section 4.3.2, we prove the

NP-completeness of the latter problem with an arbitrary number of speeds (Theorem 6). However, we propose a pseudo-polynomial algorithm to solve the problem in Section 4.3.3. This algorithm turns out to be polynomial when the number of speeds is constant, hence usable in a realistic setting with two or three speeds (Theorem 7).

### 4.3.1 Running example

Consider the example of Figure 2(b). There are two speeds, $W_1 = 7$ and $W_2 = 10$, and we focus on the power minimization problem. We assume that the power consumption of a node running at speed $W_i$ is $400 + W_i^3$, for $i = 1, 2$ (400 is the static power). We consider the subtree rooted in $A$. Several decisions can be taken locally:

- place a server at node $A$, running at speed $W_2$, hence minimizing the number of traversing requests. Another solution without traversing requests is to have two servers, one at node $B$ and one at node $C$, both running at speed $W_1$, but this would lead to a higher power consumption, since $800 + 2 \times 7^3 > 400 + 10^3$;
- place a server running at speed $W_1$ at node $C$, thus having 3 requests going through node $A$.

The choice cannot be made greedily, since it depends upon the rest of the tree: if the root $r$ has four client requests, then it is better to let some requests through (one server at node $C$), since it optimizes power consumption. However, if it has ten requests, it is necessary to have no request going through $A$, otherwise node $r$ is not able to process all its requests.

From this example, it seems very hard to design a greedy strategy to minimize the power consumption. Similarly, if we would like to reuse the algorithm of Section 4.2 to solve the MINPOWER-BOUNDEDCOST-WITHPRE bi-criteria problem, we would need to account for speeds. Indeed, the best solution of subtree $A$ with one server is no longer always the one that minimizes the number of requests (in this case, placing one server on node $A$), since it can be better for power consumption to let three requests traverse node $A$ and balance the load upper in the tree.

We prove in the next section the NP-completeness of the problem, when the number of speeds is arbitrary. However, we can adapt the dynamic programming algorithm, which becomes exponential in the number of speeds, but hence remains polynomial for a constant number of speeds (see Section 4.3.3).

### 4.3.2 NP-completeness of MINPOWER

In this section, we prove Theorem 6, i.e., the NP-completeness of the MINPOWER problem, even with no static power, when there is an arbitrary number of speeds.

*Proof of Theorem 6.* We consider the associated decision problem: given a total power consumption $\mathcal{P}$, is there a solution that does not consume more than $\mathcal{P}$?

First, the problem is clearly in NP: given a solution, i.e., a set of servers, and the speed of each server, it is easy to check in polynomial time that no capacity constraint is exceeded, and that the power consumption meets the bound.

To establish the completeness, we use a reduction from 2-Partition [19]. We consider an instance $\mathcal{I}_1$ of 2-Partition: given $n$ strictly positive integers $a_1, a_2, \ldots, a_n$, does there exist a subset $I$ of $\{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Let $S = \sum_{i=1}^{n} a_i$; we assume that $S$ is even (otherwise there is no solution).

We build an instance $\mathcal{I}_2$ of our problem where each server has $n + 2$ speeds. We assume that the $a_i$'s are sorted in increasing order, i.e., $a_1 \leq \cdots \leq a_n$. The speeds are then, in increasing order:

- $W_1 = K$;
- $\forall 1 \leq i \leq n, W_{i+1} = K + a_i \times X$;
- $W_{n+2} = K + S \times X$;

where the values of $K$ and $X$ will be determined later.

We furthermore set that there is no static power, and the power consumption for a server running at capacity $W_i$ is therefore $\mathcal{P}_i = W_i^3$. The idea is to have $K$ large and $X$ small, so that we have an upper bound on the power consumed by a server running at capacity $W_{i+1}$, for $1 \leq i \leq n$:

$$W_{i+1}^3 = (K + a_i \times X)^3 \leq K^3 + a_i + \frac{1}{n}. \tag{7}$$

To ensure that Equation (7) is satisfied, we set

$$X = \frac{1}{3 \times K^2},$$

and then we have $(K + a_i \times X)^3 = K^3(1 + \frac{a_i}{3K^3})^3$, with $K > S$ and therefore $\frac{a_i}{3K^3} < 1$. We set $x_i = \frac{a_i}{3K^3}$, and we want to ensure that:

$$(1 + x_i)^3 \leq 1 + 3 \times x_i + \frac{1}{n \times K^3}. \tag{8}$$

To do so, we study the function

$$f(x) = (1+x)^3 - (1+3\times x) - 5x^2,$$

and we show that $f(x) \leq 0$ for $x \leq \frac{1}{2}$ (thanks to the term in $-5x^2$). We have $f(0) = 0$, and $f'(x) = 3(1+x)^2 - 3 - 10x$. We have $f'(0) = 0$, and $f''(x) = 6(1+x) - 10$. For $x \leq \frac{1}{2}$, $f''(x) < 0$. We deduce that $f'(x)$ is non increasing for $x \leq \frac{1}{2}$, and since $f'(0) = 0$, $f'(x)$ is negative for $x \leq \frac{1}{2}$.

Finally, $f(x)$ is non increasing for $x \leq \frac{1}{2}$, and since $f(0) = 0$, we have $(1+x)^3 < (1+3\times x) + 5x^2$ for $x \leq \frac{1}{2}$.

Equation (8) is therefore satisfied if $5x_i^2 \leq \frac{1}{n\times K^3}$, i.e., $K^3 \geq \frac{5a_i^2\times n}{3^2}$. This condition is satisfied for

$$K = n \times S^2,$$

and we then have $x_i < \frac{1}{2}$, which ensures that the previous reasoning was correct. Finally, with these values of $K$ and $X$, Equation (7) is satisfied.

Then, the distribution tree is the following: the root node $r$ has one client with $K + \frac{S}{2} \times X$ requests, and $n$ children $A_1, \ldots, A_n$. Each node $A_i$ has a client with $a_i \times X$ requests, and a children node $B_i$ which has $K$ requests. Figure 3 illustrates the instance of the reduction.

Finally, we ask if we can find a placement of replicas with a maximum power consumption of:

$$\mathcal{P}_{max} = (K + S\times X)^3 + n\times K^3 + \frac{S}{2} + \frac{n-1}{n}.$$

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$, since $K$ and $X$ are of polynomial size. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Let us assume first that $\mathcal{I}_1$ has a solution, $I$. The solution for $\mathcal{I}_2$ is then as follows: there is one server at the root, running at capacity $W_{n+2}$. Then, for $i \in I$,

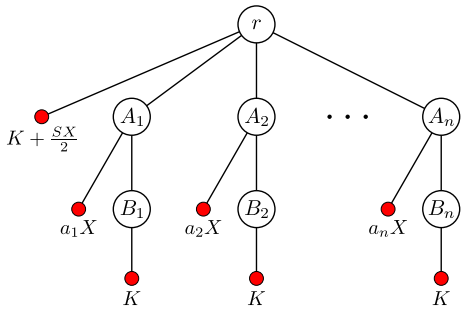



Figure 3: Illustration of the NP-completeness proof.

we place a server at node $A_i$ running at capacity $W_{1+i}$, while for $i \notin I$, we place a server at node $B_i$ running at capacity $W_1$. It is easy to check that all capacity constraints are satisfied for nodes $A_i$ and $B_i$. At the root of the tree, there are $K + \frac{S}{2} \times X + \sum_{i \notin I} a_i \times X$, which sums up to $K + S \times X$. The total power consumption is then $\mathcal{P} = (K + S \times X)^3 + \sum_{i \in I}(K + a_i \times X)^3 + \sum_{i \notin I} K^3$. Thanks to Equation (7), $\mathcal{P} \leq (K + S \times X)^3 + \sum_{i \in I}\left(K^3 + a_i + \frac{1}{n}\right) + \sum_{i \notin I} K^3$, and finally, $\mathcal{P} \leq (K + S \times X)^3 + n \times K^3 + \sum_{i \in I} a_i + \frac{n-1}{n}$. Since $I$ is a solution to 2-Partition, we have $\mathcal{P} \leq \mathcal{P}_{max}$. Finally, $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. There is a server at the root node $r$, which runs at speed $W_{n+2}$, since this is the only way to handle its $K + \frac{S}{2} \times X$ requests. This server has a power consumption of $(K+S\times X)^3$. Then, there cannot be more than $n$ other servers. Indeed, if there were $n + 1$ servers, running at the smallest speed $W_1$, their power consumption would be $(n + 1)K^3$, which is strictly greater than $n \times K^3 + \frac{S}{2} + 1$. Therefore, the power consumption would exceed $\mathcal{P}_{max}$. So, there are at most $n$ extra servers.

Consider that there exists $i \in \{1, \ldots, n\}$ such that there is no server, neither on $A_i$ nor on $B_i$. Then, the number of requests at node $r$ is at least $2K$; however, $2K > W_{n+2}$, so the server cannot handle all these requests. Therefore, for each $i \in \{1, \ldots, n\}$, there is exactly one server either on $A_i$ or on $B_i$. We define the set $I$ as the indices for which there is a server at node $A_i$ in the solution. Now we show that $I$ is a solution to $\mathcal{I}_1$, the original instance of 2-Partition.

First, if we sum up the requests at the root node, we have:

$$K + \frac{S}{2} \times X + \sum_{i \notin I} a_i \times X \leq K + S \times X.$$

Therefore, $\sum_{i \notin I} a_i \leq \frac{S}{2}$.

Now, if we consider the power consumption of the solution, we have:

$$(K + S \times X)^3 + \sum_{i \in I}(K + a_i \times X)^3 + \sum_{i \notin I} K^3 \leq \mathcal{P}_{max}.$$

Let us assume that $\sum_{i \in I} a_i > \frac{S}{2}$. Since the $a_i$ are integers, we have $\sum_{i \in I} a_i \geq \frac{S}{2} + 1$. It is easy to see that $(K + a_i \times X)^3 > K^3 + a_i$. Finally, $\sum_{i \in I}(K + a_i \times X)^3 + \sum_{i \notin I} K^3 \geq n \times K^3 + \sum_{i \in I} a_i \geq n \times K^3 + \frac{S}{2} + 1$. This implies that the total power consumption is greater than $\mathcal{P}_{max}$, which leads to a contradiction, and therefore $\sum_{i \in I} a_i \leq \frac{S}{2}$.

We conclude that $\sum_{i \notin I} a_i = \sum_{i \in I} a_i = \frac{S}{2}$, and so the solution $I$ is a 2-Partition for instance $\mathcal{I}_1$. This concludes the proof. □

### 4.3.3 A pseudo-polynomial algorithm for MINPOWER-BOUNDEDCOST

In this section, we sketch how to adapt the algorithm of Section 4.2 to account for power consumption. As illustrated in the example of Section 4.3.1, the current algorithm may lead to a non-optimal solution for the power consumption if used only with the higher speed for servers. Therefore, we refine it and compute, in each subtree, the optimal solution with, for $1 \leq j, j' \leq M$,

- exactly $n_j$ new servers running at speed $W_j$;
- exactly $e_{j,j'}$ pre-existing servers whose operation speeds have changed from $W_j$ to $W_{j'}$.

Recall that we previously had only two parameters, $N$ the number of new servers, and $E$ the number of pre-existing servers, thus leading to a total of $(N - E + 1)^2 \times (E + 1)^2$ iterations for the merging. Now, the number of iterations is $(N - E + 1)^{2M} \times (E + 1)^{2M^2}$, since we have $2 \times M$ loops of maximum size $N - E + 1$ over the $n_j$ and $n'_j$, and $2 \times M^2$ loops of maximum size $E + 1$ over the $e_{j,j'}$ and $e'_{j,j'}$.

The new algorithm is similar, except that during the merge procedure, we must consider the type of the current node that we are processing (existing or not), and furthermore set it to all possible speeds: we therefore add a loop of size $M$. The principle is similar, except that we need to have larger tables at each node, and to iterate over all parameters. The complexity of the $N$ calls to this procedure is now in $O(N \times M \times (N - E + 1)^{2M} \times (E + 1)^{2M^2})$.

Of course, we need also to update the initialization and main procedures to account for the increasing number of parameters. For the algorithm, first we compute all costs, accounting for the cost of changing speeds, and then we scan all solutions, and return one whose cost is not greater than the threshold, and which minimizes the power consumption. The most time-consuming part of the algorithm is still the call to the merging procedure, hence a complexity in $O(N \times M \times (N - E + 1)^{2M} \times (E + 1)^{2M^2})$.

With a constant number of capacities, this algorithm is polynomial, which proves Theorem 7. For instance, with $M = 2$, the worst case complexity is $O(N^{13})$. Without pre-existing servers, this complexity is reduced to $O(N^5)$.

## 4.4 Simulations

In this section, we compare our algorithms with the algorithms of [43], which do not account for pre-existing servers and for power consumption. First in Section 4.4.1, we focus on the impact of pre-existing servers. Then we consider the

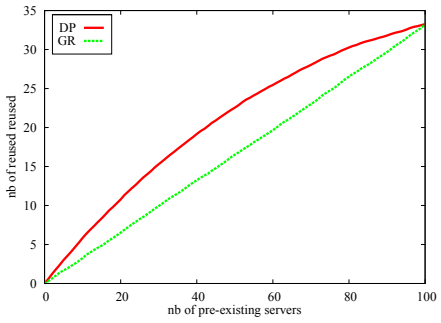
power consumption minimization criterion in Section 4.4.2. Note that experiments have been run sequentially on an Intel Xeon 5250 processor.

### 4.4.1 Impact of pre-existing servers

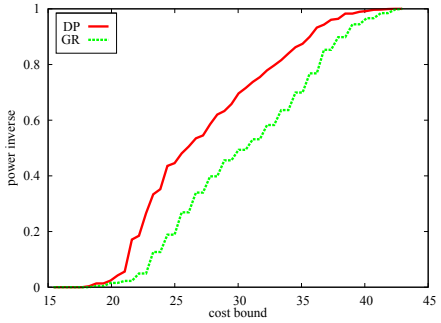
In this set of experiments, we randomly build a set of distribution trees with $N = 100$ internal nodes of maximum capacity $W = 10$. Each internal node has between 6 and 9 children, and clients are distributed randomly throughout the tree: each internal node has a client with a probability $0.5$, and this client has between 1 and 6 requests.

In the first experiment, we draw 200 random trees without any existing replica in them. Then we randomly add $0 \leq E \leq 100$ pre-existing servers in each tree. Finally, we execute both the greedy algorithm (GR) of [43], and the algorithm of Section 4.2 (DP) on each tree, and since both algorithms return a solution with the minimum number of replicas, the cost of the solution is directly related to the number of pre-existing replicas that are reused. Figure 4(a) shows the average number of pre-existing servers that are reused in each solution over the 200 trees, for each value of the number $E$ of pre-existing servers. When the tree has a very small ($E \approx 0$) or very large ($E \approx N$) number of pre-existing replicas, both algorithms return the same solution. Still, DP achieves an average reuse of $4.13$ more servers than GR, and it can reuse up to 15 more servers.

In a second experiment, we study the behavior of the algorithms in a *dynamic* setting, with 20 update steps. At each step, starting from the current solution, we update the number of requests per client and recompute an optimal solution with both algorithms, starting from the servers that were placed at the previous step.




(a) Experiment 1: increasing number of pre-existing servers.




(b) Experiment 3: Power minimization.

Figure 4: Experiments 1 and 3.

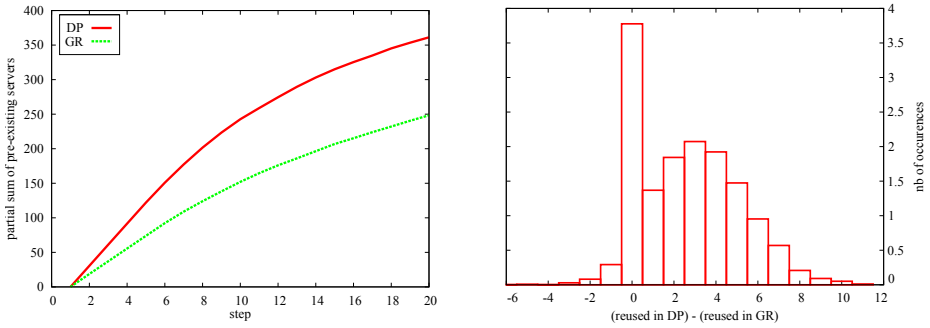



Figure 5: Experiment 2: consecutive executions of the algorithms.

Initially, there are no pre-existing servers, and at each step, both algorithms obtain a different solution. However, they always reach the same total number of servers since they have the same requests; but after the first step, they may have a different set of pre-existing servers. Similarly to Experiment 1, the simulation is conducted on 200 distinct trees, and results are averaged over all trees. In Figure 5 (left), at each step, we compare the number of existing replicas in the solutions found by the two algorithms, and hence the cost of the solutions. We plot the cumulative number of servers that have been reused so far (hence accounting for all previous steps). As expected, the DP algorithm makes a better reuse of pre-existing replicas. Figure 5 (right) compares, at each step, the number of pre-existing servers reused by DP and by GR. We count the average number of steps (over 20) at which each value is reached. It occasionally happens that the greedy algorithm performs a better reuse, because it is not starting from the same set of pre-existing servers, but overall this experiment confirms the better reuse of the dynamic programming algorithm, even when the algorithms are applied on successive steps.

Note however that taking pre-existing replicas into account has an impact on the execution time of the algorithm: in these experiments, GR runs in less than one second per tree, while DP takes around forty seconds per tree. Also, we point out that the shape of the trees does not seem to modify the results: we present in [9] similar results with trees where each node has between 2 and 4 children.

### 4.4.2 With power consumption

To study the practical applicability of the bi-criteria algorithm (DP) for the MINPOWER-BOUNDEDCOST problem (see Section 4.3.3), we have implemented it with two speeds $W_1 = 5$ and $W_2 = 10$, and compared it with the algorithm in [43]; this algorithm does not account for power minimization, but minimizes the value of the

maximal capacity $W$ when given a cost bound. More precisely, in the experiment we try all values $5 \leq W \leq 10$, and compute the corresponding cost and power consumption. To be fair, when a server has 5 requests or less, we operate it under the first speed $W_1$. Given a bound on the cost, we keep the solution that minimizes the power consumption. We call GR this version of the algorithm in [43] modified for power as explained above.

We randomly build 100 trees with 50 nodes each, and we select 5 nodes as pre-existing servers. Clients have between 1 and 5 requests, so that a solution with replicas in the first speed can always be found. The cost function is such that, for any $i, i' \in \{1, 2\}$, $\mathsf{create}_i = 0.1$, $\mathsf{delete}_i = 0.01$ and $\mathsf{changed}_{i,i'} = 0.001$. The power consumed by a server in speed $i$ is $\mathcal{P}_i = \frac{1}{10} W_1^3 + W_i^3$. In Figure 4(b), we plot the inverse of the power of a solution, given a bound on the cost (the higher the better). If the algorithm fails to find a solution for a tree, the value is 0, and we average the inverse of the power over the 100 trees, for both algorithms. For intermediate cost values, our algorithm is much better than the version of [43] in terms of power consumption: GR consumes in average more than 30% more power than DP, when the cost bound is between 29 and 34.

Here again, it takes more time to obtain the optimal solution with DP than to run the greedy algorithm several times: GR runs in around one second per tree, while DP takes around five minutes per tree. Also, we have performed some more experiments with slightly different parameters, but got no significant differences, as is shown in [9], in particular with no pre-existing replicas at all.

#### 4.4.3 Running time of the algorithms

Recall that the theoretical complexity of GR is of order $O(N \log N)$ (without power and without pre-existing servers), while DP is of order $O(N^5)$, both for the version with power (two speeds) but without pre-existing servers, and for the version without power but with pre-existing servers. In practice, the execution times of GR are always very small (a few milliseconds). For DP, we have plotted its execution time as a function of $N$ (see [9]). Run time measurements show that the experimental values have a shape in $N^5$, which confirms the theoretical complexity. Moreover, our DP algorithms run in less than $N^5$ microseconds for reasonable values of $N$, which allows the use of these algorithms in practical situations.

Indeed, without power, we are able to process trees with 500 nodes and 125 pre-existing servers in 30 minutes; with power and no pre-existing server, we can process trees with 300 nodes in one hour. The algorithm with power and pre-existing servers is the most time-consuming: it takes around one hour to process a tree with 70 nodes and 10 pre-existing servers.

## 4.5 Concluding remarks

In this section, we have addressed the problem of updating the placement of replicas in a tree network. We have provided an optimal dynamic programming algorithm whose cost is at most $O(N^5)$, where $N$ is the number of nodes in the tree. This complexity may seem high for very large problem sizes, but our implementation of the algorithm is capable of managing trees with up to $500$ nodes in half an hour, which is reasonable for a large spectrum of applications (e.g., such as database updates during the night).

The optimal placement update algorithm is a first step towards dealing with dynamic replica management. When client requests evolve over time, the placement of the replicas must be updated at regular intervals, and the overall cost is a trade-off between two extreme strategies: (i) "lazy" updates, where there is an update only when the current placement is no longer valid; the update cost is minimized, but changes in request volume and location since the last placement may well lead to poor resource usage; and (ii) systematic updates, where there is an update every time step; this leads to an optimized resource usage but encompasses a high update cost. Clearly, the rates and amplitudes of the variations of the number of requests issued by each client in the tree are very important to decide for a good update interval. Still, establishing the cost of an update is a key result to guide such a decision. When un-frequent updates are called for, or when resources have a high cost, the best solution is likely to use our optimal but expensive algorithm. On the contrary, with frequent updates or low-cost servers, we may prefer to resort to faster (but sub-optimal) update heuristics.

Our main contribution is to have provided the theoretical foundations for a single step reconfiguration, whose complexity is important to guide the design of lower-cost heuristics. Also, we have done a first attempt to take power consumption into account, in addition to usual performance-related objectives. Power consumption has become a very important concern, both for economic and environmental reasons, and it is important to account for it when designing replica placement strategies.

Even though the optimal algorithms have a high worst-case complexity, we have successfully implemented all of them, including the most time-consuming scheme capable of optimizing power while enforcing a bounded cost that includes pre-existing servers. We were able to process trees with a reasonable number of nodes.

As future work, we plan to design polynomial-time heuristics with a lower complexity than the optimal solution. The idea would be to perform some local optimizations to better load-balance the number of requests per replica, with the goal of minimizing the power consumption. These heuristics should be tuned for

dedicated applications, and should (hopefully!) build upon the fundamental results (complexity and algorithms) that we have provided in this section. Finally, it would be interesting to add more parameters in the model, such as the cost of routing, or the introduction of quality of service constraints.

# 5 Checkpointing strategies

In this section, we give a motivating example of the use of the CONTINUOUS energy model introduced in Section 2. We aim at minimizing the energy consumption when executing a divisible workload under a bound on the total execution time, while resilience is provided through checkpointing. We discuss several variants of this multi-criteria problem. Given the workload $W$, we need to decide how many chunks to use, what are the sizes of these chunks, and at which speed each chunk is executed. Furthermore, since a failure may occur during the execution of a chunk, we also need to decide at which speed a chunk should be re-executed in the event of a failure. Using more chunks leads to a higher checkpoint cost, but smaller chunks imply less computation loss (and less re-execution) when a failure occurs. We assume that a chunk can fail only once, i.e., we re-execute each chunk at most once. Indeed, the probability that a fault would strike during both the first execution and the re-execution is negligible. The accuracy of this assumption is discussed in [3].

Due to the probabilistic nature of failure hits, it is natural to study the expectation $\mathbb{E}(E)$ of the energy consumption, because it represents the average cost over many executions. As for the bound $D$ on execution time (the deadline), there are two relevant scenarios: either we enforce that this bound is a *soft deadline* to be met in expectation, or we enforce that this bound is a *hard deadline* to be met in the worst case. The former scenario corresponds to flexible environment where task deadlines can be viewed as average response times [11], while the latter scenario corresponds to real-time environments where task deadlines are always strictly enforced [41]. In both scenarios, we have to determine the number of chunks, their sizes, and the speed at which to execute (and possibly re-execute) every chunk. The different models are then compared through an extensive set of experiments.

## 5.1 Framework

First we formalize this important multi-objective problem. The general problem consists of finding $n$, the number of chunks, as well as the speeds for the execution and the re-execution of each chunk, both for soft and hard deadlines. We identify and discuss two important sub-cases that help tackling the most general problem instance: (i) a single chunk (the task is atomic); and (ii) re-execution speed is

always identical to the first execution speed. The main notations are as follows: $W$ is the total amount of work; $s$ is the processor speed for first execution; $\sigma$ is the processor speed for re-execution; $T_C$ is the checkpointing time; and $E_C$ is the energy spent for checkpointing.

### 5.1.1 Model

Consider first the case of a single chunk (or atomic task) of size $W$, denoted as SINGLECHUNK. We execute this chunk on a processor that can run at several speeds. We assume continuous speeds, i.e., the speed of execution can take an arbitrary positive real value. The execution is subject to failure, and resilience is provided through the use of checkpointing. The overhead induced by checkpointing is twofold: execution time $T_C$, and energy consumption $E_C$.

We assume that failures strike with uniform distribution, hence the probability that a failure occurs during an execution is linearly proportional to the length of this execution. Consider the first execution of a task of size $W$ executed at speed $s$: the execution time is $T_{\text{exec}} = W/s + T_C$, hence the failure probability is $P_{\text{fail}} = \lambda T_{\text{exec}} = \lambda(W/s+T_C)$, where $\lambda$ is the instantaneous failure rate. If there is indeed a failure, we re-execute the task at speed $\sigma$ (which may or may not differ from $s$); the re-execution time is then $T_{\text{reexec}} = W/\sigma + T_C$ so that the expected execution time is

$$\begin{aligned}\mathbb{E}(T) &= T_{\text{exec}} + P_{\text{fail}} T_{\text{reexec}} \\ &= (W/s + T_C) + \lambda(W/s + T_C)(W/\sigma + T_C)\ . \end{aligned} \tag{9}$$

Similarly, the worst-case execution time is

$$\begin{aligned} T_{wc} &= T_{\text{exec}} + T_{\text{reexec}} \\ &= (W/s + T_C) + (W/\sigma + T_C)\ . \end{aligned} \tag{10}$$

Remember that we assume success after re-execution, so we do not account for second and more re-executions. Along the same line, we could spare the checkpoint after re-executing the last task in a series of tasks, but this unduly complicates the analysis. In [3], we show that this model with only a single re-execution is accurate up to second order terms when compared to the model with an arbitrary number of failures that follows an Exponential distribution of parameter $\lambda$.

What is the expected energy consumed during execution? The energy consumed during the first execution at speed $s$ is $Ws^2 + E_C$, where $E_C$ is the energy consumed during a checkpoint. The energy consumed during the second execution at speed $\sigma$ is $W\sigma^2 + E_C$, and this execution takes place with probability

$P_{\text{fail}} = \lambda T_{\text{exec}} = \lambda(W/s + T_C)$, as before. Hence the expectation of the energy consumed is

$$\mathbb{E}(E) = (Ws^2 + E_C) + \lambda\left(W/s + T_C\right)\left(W\sigma^2 + E_C\right). \tag{11}$$

With multiple chunks (MULTIPLECHUNKS model), the execution times (worst case or expected) are the sum of the execution times for each chunk, and the expected energy is the sum of the expected energy for each chunk (by linearity of expectations).

We point out that the failure model is coherent with respect to chunking. Indeed, assume that a divisible task of weight $W$ is split into two chunks of weights $w_1$ and $w_2$ (where $w_1 + w_2 = W$). Then the probability of failure for the first chunk is $P^1_{\text{fail}} = \lambda(w_1/s + T_C)$ and that for the second chunk is $P^2_{\text{fail}} = \lambda(w_2/s + T_C)$. The probability of failure $P_{\text{fail}} = \lambda(W/s + T_C)$ with a single chunk differs from the probability of failure with two chunks only because of the extra checkpoint that is taken; if $T_C = 0$, they coincide exactly. If $T_C > 0$, there is an additional risk to use two chunks, because the execution lasts longer by a duration $T_C$. Of course this is the price to pay for a shorter re-execution time in case of failure: Equation (9) shows that the expected re-execution time is $P_{\text{fail}} T_{\text{reexec}}$, which is quadratic in $W$. There is a trade-off between having many small chunks (many $T_C$'s to pay, but small re-execution cost) and a few larger chunks (fewer $T_C$'s, but increased re-execution cost).

### 5.1.2 Optimization problems

The optimization problem is stated as follows: given a deadline $D$ and a divisible task whose total computational load is $W$, the problem is to partition the task into $n$ chunks of size $w_i$, where $\sum_{i=1}^{n} w_i = W$, and choose for each chunk an execution speed $s_i$ and a re-execution speed $\sigma_i$ in order to minimize the expected energy consumption:

$$\mathbb{E}(E) = \sum_{i=1}^{n} (w_i s_i^2 + E_C) + \lambda\left(\frac{w_i}{s_i} + T_C\right)\left(w_i\sigma_i^2 + E_C\right),$$

subject to the constraint that the deadline is met either in expectation or in the worst case:

$$\text{EXPECTED-DEADLINE} \quad \mathbb{E}(T) = \sum_{i=1}^{n}\left(\frac{w_i}{s_i} + T_C + \lambda\left(\frac{w_i}{s_i} + T_C\right)\left(\frac{w_i}{\sigma_i} + T_C\right)\right) \le D$$

$$\text{HARD-DEADLINE} \quad T_{wc} = \sum_{i=1}^{n}\left(\frac{w_i}{s_i} + T_C + \frac{w_i}{\sigma_i} + T_C\right) \le D$$

The unknowns are the number of chunks $n$, the sizes of these chunks $w_i$, the speeds for the first execution $s_i$ and the speeds for the second execution $\sigma_i$. We consider two variants of the problem, depending upon re-execution speeds:

- SINGLESPEED: in this simpler variant, the re-execution speed is always the same as the speed chosen for the first execution. We then have to determine a single speed for each chunk: $\sigma_i = s_i$ for all $i$.
- MULTIPLESPEEDS: in this more general variant, the re-execution speed is freely chosen, and there are two different speeds to determine for each chunk.

We also consider the variant with a single chunk (SINGLECHUNK), i.e., the task is atomic and we only need to decide for its execution speed (in the SINGLESPEED model), or for its execution and re-execution speeds (in the MULTIPLESPEEDS model). We start the study in section 5.2 with this simpler problem.

## 5.2 With a single chunk

In this section, we consider the SINGLECHUNK model: given a non-divisible workload $W$ and a deadline $D$, find the values of $s$ and $\sigma$ that minimize

$$\mathbb{E}(E) = (Ws^2 + E_C) + \lambda \left(\frac{W}{s} + T_C\right) \left(W\sigma^2 + E_C\right),$$

subject to

$$\mathbb{E}(T) = \left(\frac{W}{s} + T_C\right) + \lambda \left(\frac{W}{s} + T_C\right) \left(\frac{W}{\sigma} + T_C\right) \leq D$$

in the EXPECTED-DEADLINE model, and subject to

$$\frac{W}{s} + T_C + \frac{W}{\sigma} + T_C \leq D$$

in the HARD-DEADLINE model. We first deal with the SINGLESPEED model, where we enforce $\sigma = s$, before moving on to the MULTIPLESPEEDS model.

Note that the formal proofs of this section can be found in [3].

### 5.2.1 SINGLESPEED model

In this section, we express $\mathbb{E}(E)$ as functions of the speed $s$. That is, $\mathbb{E}(E)(s) = (Ws^2+E_C)(1+\lambda(W/s+T_C))$. The following result is valid for both EXPECTED-DEADLINE and HARD-DEADLINE models.

**Lemma 3.** *$\mathbb{E}(E)$ is convex on $\mathbb{R}_+^\star$. It admits a unique minimum $s^\star$ which can be computed numerically.*

**EXPECTED-DEADLINE:** In the SINGLESPEED EXPECTED-DEADLINE model, we denote $\mathbb{E}(T)(s) = (W/s + T_C)(1 + \lambda(W/s + T_C))$ the constraint on the execution time.

**Lemma 4.** *For any $D$, if $T_C + \lambda T_C^2 \geq D$, then there is no solution. Otherwise, the constraint on the execution time can be rewritten as $s \in (s_0, +\infty($, where*

$$s_0 = W \frac{1 + 2\lambda T_C + \sqrt{4\lambda D + 1}}{2(D - T_C(1 + \lambda T_C))}. \tag{12}$$

**Proposition 4.** *In the* SINGLESPEED *model, it is possible to numerically compute the optimal solution for* SINGLECHUNK *as follows:*

1. *If $T_C + \lambda T_C^2 \geq D$, then there is no solution;*
2. *Else, the optimal speed is $\max(s_0, s^\star)$.*

**HARD-DEADLINE:** In the HARD-DEADLINE model, the bound on the execution time can be written as $2\left(\frac{W}{s} + T_C\right) \leq D$.

**Lemma 5.** *In the* SINGLESPEED HARD-DEADLINE *model, for any $D$, if $2T_C \geq D$, then there is no solution. Otherwise, the constraint on the execution time can be rewritten as $s \in \left[\frac{W}{\frac{D}{2} - T_C}; +\infty\right($.*

**Proposition 5.** *Let $s^\star$ be the solution indicated in Lemma 3. In the* SINGLESPEED HARD-DEADLINE *model if $2T_C \geq D$, then there is no solution. Otherwise, the minimum is reached when $s = \max\left(s^\star, \frac{W}{\frac{D}{2} - T_C}\right)$.*

### 5.2.2 MULTIPLESPEEDS model

In this section, we consider the general MULTIPLESPEEDS model. We use the following notations:

$$\mathbb{E}(E)(s, \sigma) = (Ws^2 + E_C) + \lambda(W/s + T_C)(W\sigma^2 + E_C).$$

**EXPECTED-DEADLINE:** The execution time in the MULTIPLESPEEDS EXPECTED-DEADLINE model can be written as

$$\mathbb{E}(T)(s, \sigma) = (W/s + T_C) + \lambda(W/s + T_C)(W/\sigma + T_C).$$

We start by giving a useful property, namely that the deadline is always tight in the MULTIPLESPEEDS EXPECTED-DEADLINE model:

**Lemma 6.** *In the* MULTIPLESPEEDS EXPECTED-DEADLINE *model, in order to minimize the energy consumption, the deadline should be tight.*

This lemma allows us to express $\sigma$ as a function of $s$:

$$\sigma = \frac{\lambda W}{\frac{D}{\frac{W}{s}+T_C} - (1+\lambda T_C)}.$$

Also we reduce the bi-criteria problem to the minimization problem of the single-variable function:

$$s \mapsto Ws^2 + E_C + \lambda \left(\frac{W}{s} + T_C\right) \left(W \left(\frac{\lambda W}{\frac{D}{\frac{W}{s}+T_C} - (1+\lambda T_C)}\right)^2 + E_C\right), \quad (13)$$

which can be solved numerically.

**HARD-DEADLINE:** In this model we have similar results as with EXPECTED-DEADLINE. The constraint on the execution time writes: $\frac{W}{s} + T_C + \frac{W}{\sigma} + T_C \leq D$.

**Lemma 7.** *In the* MULTIPLESPEEDS EXPECTED-DEADLINE *model, in order to minimize the energy consumption, the deadline should be tight.*

This lemma allows us to express $\sigma$ as a function of $s$:

$$\sigma = \frac{W}{(D-2T_C)s - W} s$$

Finally, we reduce the bi-criteria problem to the minimization problem of the single-variable function:

$$s \mapsto Ws^2 + E_C + \lambda \left(\frac{W}{s} + T_C\right) \left(W \left(\frac{W}{(D-2T_C)s - W} s\right)^2 + E_C\right), \quad (14)$$

which can be solved numerically.

## 5.3 Several chunks

In this section, we deal with the general problem of a divisible task of size $W$ that can be split into an arbitrary number of chunks. We divide the task into $n$ chunks of size $w_i$ such that $\sum_{i=1}^{n} w_i = W$. Each chunk is executed once at speed $s_i$, and

re-executed (if necessary) at speed $\sigma_i$. The problem is to find the values of $n$, $w_i$, $s_i$ and $\sigma_i$ that minimize

$$\mathbb{E}(E) = \sum_i \left(w_i s_i^2 + E_C\right) + \lambda \sum_i \left(\frac{w_i}{s_i} + T_C\right)\left(w_i \sigma_i^2 + E_C\right),$$

subject to

$$\sum_i \left(\frac{w_i}{s_i} + T_C\right) + \lambda \sum_i \left(\frac{w_i}{s_i} + T_C\right)\left(\frac{w_i}{\sigma_i} + T_C\right) \leq D$$

in the EXPECTED-DEADLINE model, and subject to

$$\sum_i \left(\frac{w_i}{s_i} + T_C\right) + \sum_i \left(\frac{w_i}{\sigma_i} + T_C\right) \leq D$$

in the HARD-DEADLINE model. We first deal with the SINGLESPEED model, where we enforce $\sigma_i = s_i$, before dealing with the MULTIPLESPEEDS model.

Note that the formal proofs of this section can be found in [3].

### 5.3.1 Single speed model

**EXPECTED-DEADLINE:** In this section, we deal with the SINGLESPEED EXPECTED-DEADLINE model and consider that for all $i$, $\sigma_i = s_i$. Then:

$$\mathbb{E}(T)(\cup_i(w_i, s_i, s_i)) = \sum_i \left(\frac{w_i}{s_i} + T_C\right) + \lambda \sum_i \left(\frac{w_i}{s_i} + T_C\right)^2$$

$$\mathbb{E}(E)(\cup_i(w_i, s_i, s_i)) = \sum_i \left(w_i s_i^2 + E_C\right)\left(1 + \lambda\left(\frac{w_i}{s_i} + T_C\right)\right)$$

**Theorem 8.** *In the optimal solution to the problem with the* SINGLESPEED EXPECTED-DEADLINE *model, all $n$ chunks are of equal size $W/n$ and executed at the same speed $s$.*

Thanks to this result, we know that the problem with $n$ chunks can be rewritten as follows: find $s$ such that

$$n\left(\frac{W}{ns} + T_C\right) + n\lambda\left(\frac{W}{ns} + T_C\right)^2 = \frac{W}{s} + nT_C + \frac{\lambda}{n}\left(\frac{W}{s} + nT_C\right)^2 \leq D$$

in order to minimize

$$n\left(\frac{W}{n}s^2 + E_C\right) + n\lambda\left(\frac{W}{ns} + T_C\right)\left(\frac{W}{n}s^2 + E_C\right) = \left(Ws^2 + nE_C\right)\left(1 + \frac{\lambda}{n}\left(\frac{W}{s} + nT_C\right)\right).$$

One can see that this reduces to the SINGLECHUNK problem with the SINGLESPEED model (Section 5.2.1) up to the following parameter changes:

- $\lambda \leftarrow \frac{\lambda}{n}$ - $T_C \leftarrow nT_C$ - $E_C \leftarrow nE_C$

If the number of chunks $n$ is given, we can express the minimum speed such that there is a solution with $n$ chunks:

$$s_0(n) = W \frac{1 + 2\lambda T_C + \sqrt{4\frac{\lambda D}{n} + 1}}{2(D - nT_C(1 + \lambda T_C))}. \tag{15}$$

We can verify that when $D \leq nT_C(1 + \lambda n)$, there is no solution, hence obtaining an upper bound on $n$. Therefore, the two variables problem (with unknowns $n$ and $s$) can be solved numerically.

**HARD-DEADLINE:** In the HARD-DEADLINE model, all results still hold, they are even easier to prove since we do not need to introduce a second speed.

**Theorem 9.** *In the optimal solution to the problem with the* SINGLESPEED HARD-DEADLINE *model, all $n$ chunks are of equal size $W/n$ and executed at the same speed $s$.*

### 5.3.2 Multiple speeds model

**EXPECTED-DEADLINE:** In this section, we still deal with the problem of a divisible task of size $W$ that we can split into an arbitrary number of chunks, but using the more general MULTIPLESPEEDS model. We start by proving that all re-execution speeds are equal:

**Lemma 8.** *In the* MULTIPLESPEEDS *model, all re-execution speeds are equal in the optimal solution: $\exists \sigma, \forall i, \sigma_i = \sigma$, and the deadline is tight.*

We can now redefine

$$\mathbb{E}(T)(\cup_i(w_i, s_i, \sigma_i)) = T(\cup_i(w_i, s_i), \sigma)$$
$$\mathbb{E}(E)(\cup_i(w_i, s_i, \sigma_i)) = E(\cup_i(w_i, s_i), \sigma)$$

**Theorem 10.** *In the* MULTIPLESPEEDS *model, all chunks have the same size $w_i = \frac{W}{n}$, and are executed at the same speed $s$, in the optimal solution.*

Thanks to this result, we know that the $n$ chunks problem can be rewritten as follows: find $s$ such that

- $\frac{W}{s} + nT_C + \frac{\lambda}{n}\left(\frac{W}{s} + nT_C\right)\left(\frac{W}{\sigma} + nT_C\right) = D$
- in order to minimize $Ws^2 + nE_C + \frac{\lambda}{n}\left(\frac{W}{s} + nT_C\right)\left(W\sigma^2 + nE_C\right)$

One can see that this reduces to the SINGLECHUNK MULTIPLESPEEDS EXPECTED-DEADLINE task problem where:

- $\lambda \leftarrow \frac{\lambda}{n}$ $\quad$ • $T_C \leftarrow nT_C$ $\quad$ • $E_C \leftarrow nE_C$

and allows us to write the problem to solve as a two-parameter function:

$$(n,s) \mapsto Ws^2 + nE_C + \frac{\lambda}{n}\left(\frac{W}{s} + nT_C\right)\left(W\left(\frac{\frac{\lambda}{n}W}{\frac{D}{\frac{W}{s}+nT_C} - (1+\lambda T_C)}\right)^2 + nE_C\right), \tag{16}$$

which can be minimized numerically.

**HARD-DEADLINE:** In this section, the constraint on the execution time can be written as:

$$\sum_i \left(\frac{w_i}{s_i} + T_C + \frac{w_i}{\sigma_i} + T_C\right) \leq D.$$

**Lemma 9.** *In the* MULTIPLESPEEDS HARD-DEADLINE *model with divisible chunk, the deadline should be tight.*

**Lemma 10.** *In the optimal solution, for all $i, j$, $\lambda\left(\frac{w_i}{s_i} + T_C\right)\sigma_i^3 = \lambda\left(\frac{w_j}{s_j} + T_C\right)\sigma_j^3$.*

**Lemma 11.** *If we enforce the condition that the execution speeds of the chunks are all equal, and that the re-execution speeds of the chunks are all equal, then all chunks should have same size in the optimal solution.*

We have not been able to prove a stronger result than Lemma 11. However we conjecture the following result:

**Conjecture 1.** *In the optimal solution of* MULTIPLESPEEDS HARD-DEADLINE*, the re-execution speeds are identical, the deadline is tight. The re-execution speed is equal to $\sigma = \frac{W}{(D-2nT_C)s-W}s$. Furthermore the chunks should have the same size $\frac{W}{n}$ and should be executed at the same speed $s$.*

This conjecture reduces the problem to the SINGLECHUNK MULTIPLESPEEDS problem where

- $\lambda \leftarrow \frac{\lambda}{n}$ $\quad$ • $T_C \leftarrow nT_C$ $\quad$ • $E_C \leftarrow nE_C$

and allows us to write the problem to solve as a two-parameter function:

$$(n,s) \mapsto Ws^2 + nE_C + \frac{\lambda}{n}\left(\frac{W}{s} + nT_C\right)\left(W\left(\frac{W}{(D-2nT_C)s-W}s\right)^2 + nE_C\right) \tag{17}$$

which can be solved numerically.

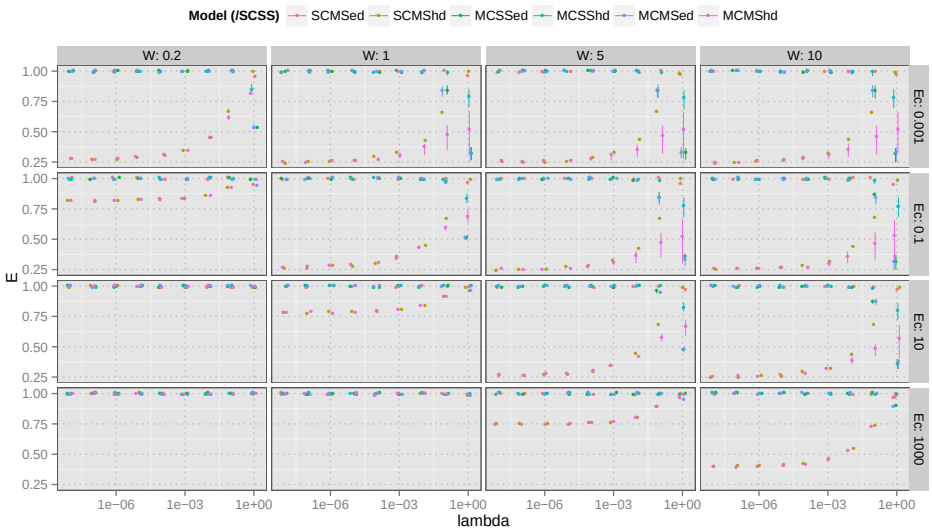



Figure 6: Comparison with single chunk single speed.

## 5.4 Simulations

### 5.4.1 Simulation settings

We performed a large set of simulations in order to illustrate the differences between all the models studied in this paper, and to show to which extent each additional degree of freedom improves the results, i.e., allowing for multiple speeds instead of a single speed, or for multiple smaller chunks instead of a single large chunk. All these simulations are conducted under both constraint types, expected and hard deadlines.

We envision reasonable settings by varying parameters within the following ranges:

- $\frac{W}{D} \in [0.2, 10]$;
- $\frac{T_C}{D} \in [10^{-4}, 10^{-2}]$;
- $E_C \in [10^{-3}, 10^{3}]$;
- $\lambda \in [10^{-8}, 1]$.

In addition, we set the deadline to 1. Note that since we study $\frac{W}{D}$ and $\frac{T_C}{D}$ instead

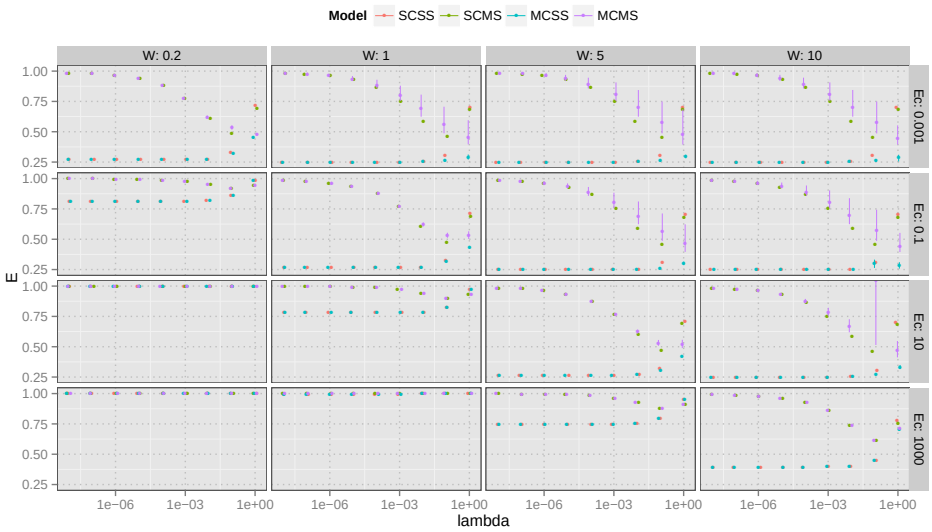



Figure 7: Comparison hard deadline versus expected deadline.

of $W$ and $T_C$, we do not need to study how the variation of the deadline impacts the simulation, this is already taken into account.

We use the Maple software to solve numerically the different minimization problems. Results are showed from two perspectives: on the one hand (Figure 6), for a given constraint (HARD-DEADLINE or EXPECTED-DEADLINE), we normalize all variants according to SINGLESPEED SINGLECHUNK, under the considered constraint. For instance, on the plots, the energy consumed by MULTIPLECHUNKS MULTIPLESPEEDS (denoted as MCMS) for HARD-DEADLINE is divided by the energy consumed by SINGLECHUNK SINGLESPEED (denoted as SCSS) for HARD-DEADLINE, while the energy of MULTIPLECHUNKS SINGLESPEED (denoted as MCSS) for EXPECTED-DEADLINE is normalized by the energy of SINGLECHUNK SINGLESPEED for EXPECTED-DEADLINE.

On the other hand (Figure 7), we study the impact of the constraint hardness on the energy consumption. For each solution form (SINGLESPEED or MULTIPLESPEEDS, and SINGLECHUNK or MULTIPLECHUNKS), we plot the ratio energy consumed for EXPECTED-DEADLINE over energy consumed for HARD-DEADLINE.

Note that for each figure, we plot for each function different values that depend on the different values of $T_C/D$ (hence the vertical intervals for points where $T_C/D$ has an impact). In addition, the lower the value of $T_C/D$, the lower the

energy consumption.

### 5.4.2 Comparison with single speed

At first, we observe that the results are identical for any value of $W/D$, up to a translation of $E_C$ (see $(W/D = 0.2, E_C = 10^{-3})$ vs. $(W/D = 5, E_C = 1000)$ or $(W/D = 1, E_C = 10^{-3})$ vs. $(W/D = 5, E_C = 0.1)$ on Figure 6, for instance).

Then the next observation is that for EXPECTED-DEADLINE, with a small $\lambda$ ($< 10^{-2}$), MULTIPLECHUNKS or MULTIPLESPEEDS models do not improve the energy ratio. This is due to the fact that, in both expressions for energy and for execution time, the re-execution term is negligible relative to the execution one, since it has a weighting factor $\lambda$. However, when $\lambda$ increases, if the energy of a checkpoint is small relative to the total work (which is the general case), we can see a huge improvement (between 25% and 75% energy saving) with MULTIPLECHUNKS.

On the contrary, as expected, for small $\lambda$'s, re-executing at a different speed has a huge impact for HARD-DEADLINE, where we can gain up to $75\%$ energy when the failure rate is low. We can indeed run at around half speed during the first execution (leading to the $1/2^2 = 25\%$ saving), and at a high speed for the second one, because the very low failure probability avoids the explosion of the expected energy consumption. For both MULTIPLECHUNKS and SINGLECHUNK, this saving ratio increases with $\lambda$ (the energy consumed by the second execution cannot be neglected any more, and both executions need to be more balanced), the latter being more sensitive to $\lambda$. But the former is the only configuration where $T_C$ has a significant impact: its performance decreases with $T_C$; still it remains strictly better than SINGLECHUNK MULTIPLESPEEDS.

### 5.4.3 Comparison between EXPECTED-DEADLINE and HARD-DEADLINE

As before, the value of $W/D$ does not change the energy ratios up to translations of $E_C$. As expected, the difference between the EXPECTED-DEADLINE and HARD-DEADLINE models is very important for the SINGLESPEED variant: when the energy of the re-execution is negligible (because of the failure rate parameter), it would be better to spend as little time as possible doing the re-execution in order to have a speed as slow as possible for the first execution, however we are limited in the SINGLESPEED HARD-DEADLINE model by the fact that the re-execution time is fully taken into account (its speed is the same as the first execution, and there is no parameter $\lambda$ to render it negligible).

Furthermore, when $\lambda$ is minimum, MULTIPLESPEEDS consumes the same energy for EXPECTED-DEADLINE and for HARD-DEADLINE. Indeed, as expected, the $\lambda$ in the energy function makes it possible for the re-execution speed to be

maximal: it has little impact on the energy, and it is optimal for the execution time; this way we can focus on slowing down the first execution of each chunk. For HARD-DEADLINE, we already run the first execution at half speed, thus we cannot save more energy, even considering EXPECTED-DEADLINE instead. When $\lambda$ increases, speeds of HARD-DEADLINE cannot be lowered but the expected execution time decreases, making room for a downgrade of the speeds in the EXPECTED-DEADLINE problems.

## 5.5 Concluding remarks

In this section, we have studied the energy consumption of a divisible computational workload on volatile platforms under the CONTINUOUS speed model. In particular, we have studied the expected energy consumption under different deadline constraints: a soft deadline (a deadline for the expected execution time), and a hard deadline (a deadline for the worst case execution time).

As stated in Section 2, the CONTINUOUS speed model is theoretically appealing, and allowed us to show mathematically, for all cases but one, that when using the MULTIPLECHUNKS model, then (i) every chunk should be equally sized; (ii) every execution speed should be equal; and (iii) every re-execution speed should also be equal. This problem remains open in the MULTIPLESPEEDS HARD-DEADLINE variant.

Through a set of extensive simulations we have shown the following: (i) when the fault parameter $\lambda$ is small, for EXPECTED-DEADLINE constraints, the SINGLECHUNK SINGLESPEED model leads to almost optimal energy consumption. This is not true for the HARD-DEADLINE model, which accounts equally for execution and re-execution, thereby leading to higher energy consumption. Therefore, for the HARD-DEADLINE model and for small values of $\lambda$, the model of choice should be the SINGLECHUNK MULTIPLESPEEDS model, and that is not intuitive. When the fault parameter rate $\lambda$ increases, using a single chunk is no longer energy-efficient, and one should focus on the MULTIPLECHUNKS MULTIPLESPEEDS model for both deadline types.

An interesting direction for future work is to extend this study to the case of an application workflow: instead of dealing with a single divisible task, we would deal with a DAG of tasks, that could be either divisible (checkpoints can take place anytime) or atomic (checkpoints can only take place at the end of the execution of some tasks). Again, we can envision both soft or hard constraints on the execution time, and we can keep the same model with a single re-execution per chunk/task, at the same speed or possibly at a different speed. Deriving complexity results and heuristics to solve this difficult problem is likely to be very challenging, but could have a dramatic impact to reduce the energy consumption of many scientific

applications.

# 6 Conclusion

In this chapter, we have discussed several energy-aware algorithms aiming at decreasing the energy consumption in data centers. We have started with a description of various energy models, ranking from the most theoretical model of continuous speeds to the more realistic discrete model. Indeed, processor speeds can be changed thanks to the DVFS technique (Dynamic Voltage and Frequency Scaling), hence decreasing energy consumption when running at a lower speed. Of course, performance should not be sacrificed for energy, and a bound on the performance should always be enforced.

We have first illustrated these models on a task graph scheduling problem where we can reclaim the energy of a schedule by running some non-critical tasks at a lower speed. Depending upon the model, the complexity of the problem varies: while several optimality results can be obtained with continuous speeds, the problem with discrete speeds is NP-hard. Through this study, we have laid the theoretical foundations for a comparative study of energy models.

We have then targeted a problem typical of data centers, namely the replica placement problem. The root server has the original copy of the database but cannot serve all clients directly, so a distribution tree is deployed to provide the clients with a hierarchical and distributed access to replicas of the original data. The problem is to decide where to place replicas, and where to serve each client. We have provided an optimal dynamic programming algorithm that works in a dynamic setting: we assume that client requests can evolve over time, and hence some replicas are already placed in the network. It is more efficient to re-use some of these replicas if possible. We have also added a criterion of power consumption to the problem, and proved the NP-completeness of this problem with a discrete energy model. In addition, some practical solutions have been proposed.

Finally, a rising concern in data centers, apart from energy consumption, is higher failure rate. We have therefore discussed checkpointing strategies, in the case of a divisible workload. Two deadline constraints have been studied: a hard deadline scenario corresponding to real-time environments where task deadlines are always strictly enforced, and a soft deadline scenario corresponding to a more flexible environment, where an average response time must be enforced. We have conducted this study under the continuous model, which enabled us to derive theoretical results: we proved that every chunk should be equally sized, and that every speed should be equal. In case of failure, we re-execute a chunk, and all re-execution speeds should also be equal.

Through these three case studies, we have demonstrated the importance and the complexity of proposing energy-aware solutions to problems that occur in data centers. We have provided a first step towards energy-efficient data centers by first discussing energy models, and then designing energy-efficient algorithms for some typical problems. Several research directions have been opened.

# References


[1] AMD processors. http://www.amd.com.

[2] 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.

[3] G. Aupy, A. Benoit, R. Melhem, P. Renaud-Goud, and Y. Robert. Energy-aware checkpointing of divisible tasks with soft or hard deadlines. In *Proceedings of the International Green Computing Conference (IGCC)*, Arlington, USA, June 2013.

[4] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 113–121. IEEE CS Press, 2003.

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

[6] E. Beigne, F. Clermidy, J. Durupt, H. Lhermet, S. Miermont, Y. Thonnart, T. Xuan, A. Valentian, D. Varreau, and P. Vivet. An asynchronous power aware and adaptive NoC based circuit. In *Proceedings of the 2008 IEEE Symposium on VLSI Circuits*, pages 190–191, June 2008.

[7] E. Beigne, F. Clermidy, S. Miermont, Y. Thonnart, A. Valentian, and P. Vivet. A Localized Power Control mixing hopping and Super Cut-Off techniques within a GALS NoC. In *Proceedings of the IEEE International Conference on Integrated Circuit Design and Technology and Tutorial (ICICDT)*, pages 37–42, June 2008.

[8] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Replica placement and access policies in tree networks. *IEEE Trans. Parallel and Distributed Systems*, 19(12):1614–1627, 2008.

[9] A. Benoit, P. Renaud-Goud, and Y. Robert. Power-aware replica placement and update strategies in tree networks. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Anchorage, USA, May 2011.

[10] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[11] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer series in Computer Science, 2005.

[12] A. P. Chandrakasan and A. Sinha. JouleTrack: A Web Based Tool for Software Energy Profiling. In *Design Automation Conference*, pages 220–225, Los Alamitos, CA, USA, 2001. IEEE Computer Society Press.

[13] G. Chen, K. Malkowski, M. Kandemir, and P. Raghavan. Reducing power with performance constraints for parallel sparse applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 8 pp., Apr. 2005.

[14] J.-J. Chen and C.-F. Kuo. Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications*, pages 28–38, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[15] J.-J. Chen and T.-W. Kuo. Multiprocessor energy-efficient scheduling for real-time tasks. In *Proceedings of International Conference on Parallel Processing (ICPP)*, pages 13–20. IEEE CS Press, 2005.

[16] I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. *Computer Networks*, 40:205–218, 2002.

[17] 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.*, 13:1157–1166, October 2005.

[18] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero. The international exascale software project: a call to cooperative action by the global high-performance community. *Int. J. High Perform. Comput. Appl.*, 23(4):309–322, 2009.

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

[20] R. Ge, X. Feng, and K. W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proceedings of the ACM/IEEE conference on SuperComputing (SC)*, page 34. IEEE Computer Society, 2005.

[21] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277 –1284, Sept. 1996.

[22] P. Grosse, Y. Durand, and P. Feautrier. Methods for power optimization in SOC-based data flow systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14:38:1–38:20, June 2009.

[23] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a pc cluster. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 340, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[24] Intel XScale technology. http://www.intel.com/design/intelxscale.

[25] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202. ACM Press, 1998.

[26] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st annual Design Automation Conference (DAC)*, pages 275–280, New York, NY, USA, 2004. ACM.

[27] K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. Parallel and Distributed Systems*, 12(6):628–637, 2001.

[28] H. Kawaguchi, G. Zhang, S. Lee, and T. Sakurai. An LSI for VDD-Hopping and MPEG4 System Based on the Chip. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, May 2001.

[29] K. H. Kim, R. Buyya, and J. Kim. Power-Aware Scheduling of Bag-of-Tasks Applications with Deadline Constraints on DVS-enabled Clusters. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 541 –548, May 2007.

[30] K. Lahiri, A. Raghunathan, S. Dey, and D. Panigrahi. Battery-driven system design: a new frontier in low power design. In *Proceedings of the 7th Asia and South Pacific Design Automation Conference and the 15th International Conference on VLSI Design (ASP-DAC)*, pages 261 –267, 2002.

[31] P. Langen and B. Juurlink. Leakage-aware multiprocessor scheduling. *J. Signal Process. Syst.*, 57(1):73–88, 2009.

[32] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of DAC'2000, the 37th Conference on Design Automation*, pages 806–809, 2000.

[33] P. Liu, Y.-F. Lin, and J.-J. Wu. Optimal placement of replicas in data grid environments with locality assurance. In *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2006.

[34] 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*, volume 4644 of *Lecture Notes in Computer Science*, pages 556–565. Springer Berlin / Heidelberg, 2007.

[35] M. P. Mills. The internet begins with coal. *Environment and Climate News*, page ., 1999.

[36] T. Okuma, H. Yasuura, and T. Ishihara. Software energy reduction techniques for variable-voltage processors. *IEEE Design Test of Computers*, 18(2):31 – 41, Mar. 2001.

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

[38] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43:67–80, 2008.

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

[40] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer-Verlag, 2003.

[41] J. A. Stankovic, K. Ramamritham, and M. Spuri. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[42] 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 *Proceedings of the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)*, pages 368 –377, May 2010.

[43] J.-J. Wu, Y.-F. Lin, and P. Liu. Optimal replica placement in hierarchical Data Grids with locality assurance. *Journal of Parallel and Distributed Computing (JPDC)*, 68(12):1517–1538, 2008.

[44] R. Xu, D. Mossé, and R. Melhem. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Trans. Comput. Syst.*, 25(4):9, 2007.

[45] L. Yang and L. Man. On-Line and Off-Line DVS for Fixed Priority with Preemption Threshold Scheduling. In *Proceedings of the International Conference on Embedded Software and Systems (ICESS)*, pages 273 –280, May 2009.

[46] 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, Washington, DC, USA, 1995. IEEE Computer Society.

[47] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th annual Design Automation Conference (DAC)*, pages 183–188, New York, NY, USA, 2002. ACM.

[48] D. Zhu, R. Melhem, and D. Mossé. 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.