Workload Balancing and Throughput Optimization for Heterogeneous Systems Subject to Failures

Anne Benoit[1], Alexandru Dobrila[2,⋆], Jean-Marc Nicod[2], and Laurent Philippe[2]

[1] ENS Lyon, Université de Lyon, LIP laboratory (ENS, CNRS, INRIA, UCBL), France
[2] Université de Franche-Comté, LIFC laboratory, (UFC), France
adobrila@lifc.univ-fcomte.fr

Abstract. In this paper, we study the problem of optimizing the throughput of streaming applications for heterogeneous platforms subject to failures. The applications are linear graphs of tasks (pipelines), and a type is associated to each task. The challenge is to map tasks onto the machines of a target platform, but machines must be specialized to process only one task type, in order to avoid costly context or setup changes. The objective is to maximize the throughput, i.e., the rate at which jobs can be processed when accounting for failures. For identical machines, we prove that an optimal solution can be computed in polynomial time. However, the problem becomes NP-hard when two machines can compute the same task type at different speeds. Several polynomial time heuristics are designed, and simulation results demonstrate their efficiency.

1 Introduction

Most of the distributed environments are subject to failures, and each component of the environment has its own failure rate. Assuming that a failure may be tolerated, as for instance in asynchronous systems [1] or production systems, the failures have an impact on the system performance. When scheduling an application onto such a system, either we can account for failures to help improve the performance in case of failures, or ignore them. In some environments, such as computing grids, this failure rate is so high that we cannot ignore failures when scheduling applications that last for a long time as a batch of input data processed by pipelined tasks for instance. This is also the case for micro-factories where a production is composed of several instances of the same micro-component that must be processed by cells.

In this paper, we deal with scheduling and mapping strategies for *coarse-grain* workflow applications [18,19]. The applications are linear graphs of tasks (pipelines), and a type is associated to each task. The target platform is a set

⋆ Corresponding author.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 242–254, 2011.

of execution resources (generically called *machines*), such as a grid or a micro-factory, on which the tasks must be mapped. A series of jobs enters the workflow and progresses from task to task until the final result is computed. Once a task is mapped onto a set of dedicated resources (known in the literature as *multi-processor tasks* [3,7]), the computation requirements and the failure rates for each machine when processing one job of the workflow are known. After an initialization delay, a new job is completed every period, where the period is the inverse of the throughput. It is defined as the longest cycle-time of a machine. Note that we target coarse-grain applications and platforms on which the cost of communications is negligible in comparison to the cost of computations.

In the distributed computing system context, a use case of a streaming application is for instance an image processing application where images are processed in batches, on a SaaS (Software as a service) platform. In this context, failures may occur because of the nodes, but they also may be impacted by the complexity of the service [9]. On the production side, a use case is a micro-factory [13,5,12] composed of several cells that provides functions as assembly or machining. But, at this scale, the physical constraints are not totally controlled and it is mandatory to take failures into account in the automated command. A common property of these systems is that we cannot use replication, as for instance in [4,14,10], to overcome the failures. For streaming applications, it may impact the throughput to replicate each task. For a production which deals with physical objects, replication is not possible. Fortunately, losing a few jobs may not be a big deal; for instance, the loss of some images in a stream will not alter the result, as far as the throughput is maintained, and losing some micro-products is barely more costly than the occupation of the processing resources that have been dedicated to it. The failure model is based on the Window-Constrained [16] model, often used in real-time environment. In this model, only a fraction of the messages will reach their destination. The losses are not considered as a failure but as a guarantee: for a given network, a Window-Constrained scheduling [15,17] can guarantee that no more than x messages will be lost for every y sent messages.

In this paper, we therefore solely concentrate on the problem of period minimization (i.e., throughput maximization), where extra jobs are processed to account for failures. For instance, if there is a single task, mapped on a single machine, with a failure rate of $1/2$, a throughput of x jobs per unit time will be achieved if the task processes $2 \times x$ jobs per time unit.

The paper is organized as follows. Section 2 presents the framework and formalizes the optimization problems tackled in the paper. An exhaustive complexity study is provided in Section 3: we exhibit some particular polynomial problem instances, and prove that the remaining problem instances are NP-hard problems. In Section 4, we design a set of polynomial-time heuristics to solve the most general problem instance, building upon complexity results, and in particular linear program formulations to solve sub-problems. Moreover, we conduct extensive simulations to assess the relative and absolute performance of the heuristics. Finally, we conclude in Section 5.

2 Framework and Optimization Problems

Applicative framework. The application consists of a linear chain of n tasks, $T_1, T_2, \ldots, T_n$. A type is associated to each task: we have a set of p task types with $n \geq p$, and a function t which returns the type of a task. Hence, $t(i)$ is the type of task T_i. A series of jobs enters the workflow and progresses from task to task until the final result is computed, and x_i is the average number of jobs processed by task T_i to output one job out of the system. Note that x_{i+1} depends on x_i and on the failure rate of the machine processing T_i (see below).

Target platform. The target platform is distributed and heterogeneous. It consists of a set of m machines (a cell in the micro-factory or a host in a grid platform), $M_1, M_2, \ldots, M_m$. The task processing time depends on the machine that performs it: it takes $w_{i,u}$ units of time to machine M_u to execute task T_i on one job. Each machine is able to process all the task types. However, to avoid costly context or setup changes during execution, the machines may be specialized to process only one task type. Note that we do not take communication times into account as we consider that the processing time is much greater than the communication time (coarse-grain applications).

Failure model. It may happen that a job (or product) is lost (or damaged) while a task is being executed on this job. For instance, an electrostatic charge may be accumulated on an actuator and a piece will be pushed away rather than caught, or a message will be lost due to network contention. Note that we deal only with transient failures, as defined in [8]: the tasks are failing for some jobs, but we do not consider a permanent failure of the machine responsible of the task, as this would lead to a failure for all the remaining jobs to be processed and the inability to finish them. In order to deal with failures, we process more jobs than needed, so that at the end, the required throughput is reached. The failure rate of task T_i performed onto machine M_u is the percentage of failure for this task and it is denoted $f_{i,u}$.

Objective function. Our goal is to assign tasks to machines so as to optimize some key performance criteria. A task can be allocated to several machines, and $q(i,u)$ is the quantity of task T_i executed by machine M_u; if $q(i,u) = 0$, T_i is not assigned to M_u. Recall that x_i is the average number of jobs processed by task T_i to output one job out of the system. We must have, for each task T_i, $\sum_{u=1}^{m} q(i,u) = x_i$, i.e., enough jobs are processed for task T_i in the system.

The objective function is to maximize the number of jobs that exit the system per time unit, making abstraction of the initialization and clean-up phases. This objective is important when a large number of jobs must be processed. Actually, we deal with the equivalent optimization problem that minimize the *period*, the inverse of the throughput. One challenge is that we cannot compute the number x_i of jobs that must be processed by task T_i before allocating tasks to machines, since x_i depends on the failure rates incurred by the allocation. However, each task T_i has a unique successor task T_{i+1}, and x_{i+1} is the amount of jobs needed by T_{i+1} as input. Since T_i is distributed on several machines

with different failure rates, we have $\sum_{u=1}^{m}(q(i,u)\times(1-f_{i,u}))=x_{i+1}$, where $q(i,u)\times(1-f_{i,u})$ represents the amount of jobs output by the machine M_u if $q(i,u)$ jobs are treated by that machine. For each task, we sum all the instances treated by all the machines. We are now ready to define the cycle-time ct_u of machine M_u: it is the time needed by M_u to execute all tasks T_i with $q(i,u)>0$: $ct_u=\sum_{i=1}^{n}q(i,u)\times w_{i,u}$. The objective function is to minimize the maximum cycle-time, which corresponds to the period of the system: $\min\max_{1\leq u\leq m}ct_u$.

Rules of the game. Different rules of the game may be enforced to define the allocation, i.e., the $q(i,u)$ values. For *one-to-many* mappings, we enforce that a single task must be mapped onto each machine: $\forall i,i':1\leq i,i'\leq n\ s.t.\ i\neq i'$, $q(i,u)>0\Rightarrow q(i',u)=0$. This kind of mapping is quite restrictive because we must have at least as many machines as tasks. Note that a task can be allocated to several machines. We relax this rule to allow for *specialized* mappings, in which several tasks of the same type can be mapped onto the same machine: $\forall i,i':1\leq i,i'\leq n\ s.t.\ t(i)\neq t(i'),q(i,u)>0\Rightarrow q(i',u)=0$. Note that if each task has a different type, the specialized mapping and the one-to-many mapping are equivalent. Finally, *general* mappings have no constraints: any task (no matter the type) can be mapped on any machine.

Problem definition. For the optimization problem that we consider, the three important parameters are: (i) the rules of the game (*one-to-many (o2m)* or *specialized (spe)* or *general (gen)* mapping); (ii) the failure model (f if failures are all identical, f_i if the failure for a same task is identical on two different machines, f_u if the failure rate depends only on the machine, and the general case $f_{i,u}$); and (iii) the computing time (w if the processing times are all identical, w_i if it differs only from one task to another, w_u if it depends only on the machine, and $w_{i,u}$ in the general case). We are now ready to define the optimization problem:

Definition 1. MinPer(R,F,W)*: Given an application and a target platform, with a failure model* $F=\{f|f_i|f_u|f_{i,u}|*\}$ *and computation times* $W=\{w|w_i|w_u|w_{i,u}|*\}$*, find a mapping (i.e., values of* $q(i,u)$ *such that for each task* T_i *with* $1\leq i\leq n$, $\sum_{u=1}^{m}q(i,u)=x_i$*) following rule* $R=\{o2m|spe|gen|*\}$*, which minimizes the period of the application,* $\max_{1\leq u\leq m}\sum_{i=1}^{n}q(i,u)\times w_{i,u}$.

Note that $*$ is used to express the problem with any variant of the corresponding parameter; for instance, MinPer$(*,f_{i,u},w)$ is the problem of minimizing the period with any mapping rule, where failure rates are general, while execution times are all identical.

3 Complexity Results

We assess the complexity of the different instances of the MinPer(R,F,W) problem. First we provide the complexity of the problems with $F=f_i$, and then we discuss the most general problems with $F=f_{i,u}$. Even though the general problem is NP-hard, we show that once the allocation of tasks to machines is known, we can optimally decide how to share tasks between machines, in polynomial time. Also, we give an integer linear program to solve the problem.

3.1 Complexity of the MinPer$(*, f_i, *)$ Problems

We first show how the MinPer$(*, f_i, *)$ problems can be simplified. Indeed, in this case, the number of products that should be computed for task T_i at each period, x_i, is independent of the allocation of tasks to machines. We can therefore ignore the failure probabilities, and focus on the computation of the period of the application. The following Lemma 1 allows us to further simplify the problem: tasks of similar type can be grouped and processed as a single *equivalent* task.

Lemma 1. *For* MinPer$(*, f_i, w_i)$ *or* MinPer$(*, f_i, w_u)$*, there exists an optimal solution in which all tasks of the same type are executed onto the same set of machines, in equal proportions:* $\forall i, j : 1 \leq i, j \leq n$ *with* $t(i) = t(j)$*,*

$$\exists \alpha_{i,j} \in \mathbb{Q} \text{ s.t. } \forall u : 1 \leq u \leq m,\ q(i,u) = \alpha_{i,j} \times q(j,u)\ . \tag{1}$$

The proof consists in building an optimal solution which follows Equation (1), from an existing one. We redistribute the work and define the $\alpha_{i,j}$ values for each problem instance. The detailed proof is available in the companion research report [2].

Corollary 1. *For* MinPer$(*, f_i, w_i)$ *or* MinPer$(*, f_i, w_u)$*, we can group all tasks of same type* t *as a single* equivalent *task* $T_t^{(eq)}$*, s.t.* $x_t^{(eq)} = \sum_{1 \leq i \leq n | t(i) = t} x_i$*. Then, we can solve this problem with the one-to-many rule, and deduce the solution of the initial problem.*

Proof. Following Lemma 1, we search for the optimal solution which follows Equation (1). Since all tasks of the same type are executed onto the same set of machines in equal proportions, we can group them as a single equivalent task. The amount of work to be done by the set of machines corresponds to the total amount of work of the initial tasks, i.e., for a type t, $\sum_{1 \leq i \leq n | t(i) = t} x_i$.

The one-to-many rule decides on which set of machines each equivalent task is mapped, and then we share the initial tasks in equal proportions to obtain the solution to the initial problem: if task T_i is not mapped on machine M_u, then $q(i,u) = 0$, otherwise $q(i,u) = \frac{x_i}{x_{t(i)}^{(eq)}} \times \frac{P}{w_{i|u}}$, where $w_{i|u} = \{w_i \mid w_u\}$.

We are now ready to establish the complexity of the MinPer$(*, f_i, *)$ problems. Recall that n is the number of tasks, m is the number of machines, and p is the number of types. We start by providing polynomial algorithms for one-to-many and specialized mappings with w_i (Theorem 1 and Corollary 2). Then, we discuss the case of general mappings, which can also be solved in polynomial time (Theorem 2). Finally, we tackle the instances which are NP-hard (Theorem 3).

Theorem 1. MinPer$(o2m, f_i, w_i)$ *can be solved in time* $O(m \times \log n)$*.*

Proof. First, note that solving this one-to-many problem amounts to decide on how many machines each task is executed (since machines are identical), and then split the work evenly between these machines to minimize the period. Hence, if T_i is executed on k machines, $q(i,u) = \frac{x_i}{k}$, where M_u is one of these k machines, and the corresponding period is $\frac{x_i}{k} \times w_i$.

We provide a greedy algorithm to solve the problem. The idea is to assign initially one machine per task (note that there is a solution only if $m \geq n$), sort the tasks by non-increasing period, and then iteratively add a machine to the task whose machine(s) have the greater period, while there are some machines available. Let g_i be the current number of machines assigned to task T_i: the corresponding period is $\frac{x_i}{g_i} \times w_i$. At each step, we insert the task whose period has been modified in the ordered list of tasks, which can be done in $O(\log n)$ (binary search). The initialization takes a time $O(n \log n)$ (sorting the tasks), and then there are $m - n$ steps of time $O(\log n)$. Since we assume $m \geq n$, the complexity of this algorithm is in $O(m \times \log n)$. To prove that this algorithm returns the optimal solution, let us assume that there is an optimal solution of period P_{opt} that has assigned o_i machines to task T_i, while the greedy algorithm has assigned g_i machines to this same task, and its period is $P_{greedy} > P_{opt}$. Let T_i be the task which enforces the period in the greedy solution (i.e., $P_{greedy} = x_i w_i / g_i$). The optimal solution must have given at least one more machine to this task, i.e., $o_i > g_i$, since its period is lower. This means that there is a task T_j such that $o_j < g_j$, since $\sum_{1 \leq i \leq n} o_i \leq \sum_{1 \leq i \leq n} g_i = m$ (all machines are assigned with the greedy algorithm). Then, note that since $o_j < g_j$, because of the greedy choice, $x_j w_j / o_j \geq x_i w_i / g_i$ (otherwise, the greedy algorithm would have given one more machine to task T_i). Finally, $P_{opt} \geq x_j w_j / o_j \geq x_i w_i / g_i = P_{greedy}$, which leads to a contradiction, and concludes the proof.

Corollary 2. MinPer(spe, f_i, w_i) *can be solved in time* $O(n + m \times \log p)$.

Proof. For the specialized mapping rule, we use Corollary 1 to solve the problem: first we group the n tasks by types, therefore obtaining p equivalent tasks, in time $O(n)$. Then, we use Theorem 1 to solve the problem with p tasks, in time $O(m \times \log p)$. Finally, the computation of the mapping with equal proportions is done in $O(n)$, which concludes the proof.

Theorem 2. MinPer$(gen, f_i, *)$ *can be solved in polynomial time.*

Proof. We exhibit a linear program to solve the problem for the general case with $w_{i,u}$. Note however that the problem is trivial for w_i or w_u: we can use Corollary 1 to group all tasks as a single equivalent task, and then share the work between machines as explained in the corollary.

In the general case, we solve the following (rational) linear program, where the variables are P (the period), and $q(i,u)$, for $1 \leq i \leq n$ and $1 \leq u \leq m$.

$$\begin{array}{l} \text{Minimize } P, \text{ subject to} \\ \text{(i) } q(i,u) \geq 0 \text{ for } 1 \leq i \leq n, 1 \leq u \leq m \\ \text{(ii) } \sum_{1 \leq u \leq m} q(i,u) = x_i \text{ for each task } T_i \text{ with } 1 \leq i \leq n \\ \text{(iii) } \sum_{1 \leq i \leq n} q(i,u) \times w_{i,u} \leq P \text{ for each machine } M_u \text{ with } 1 \leq u \leq m \end{array} \tag{2}$$

The size of this linear program is polynomial in the size of the instance, all $n \times m + 1$ variables are rational. Therefore, it can be solved in polynomial time [11].

Finally, we prove that the remaining problem instances are NP-hard (one-to-many or specialized mappings, with w_u or $w_{i,u}$). Since MinPer$(o2m, f_i, w_u)$ is

a special case of all other instances, it is sufficient to prove the NP-completeness of the latter problem.

Theorem 3. *The* MinPer$(o2m, f_i, w_u)$ *problem is NP-hard in the strong sense.*

Proof. We consider the following decision problem: given a period P, is there a one-to-many mapping whose period does not exceed P? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time whether it is valid or not. The NP-completeness is obtained by reduction from 3-PARTITION [6], which is NP-complete in the strong sense.

We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3n$ positive integers $a_1, a_2, \ldots, a_{3n}$ such that for all $i \in \{1, \ldots, 3n\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{n} a_i = nB$, does there exist a partition $I_1, \ldots, I_n$ of $\{1, \ldots, 3n\}$ such that for all $j \in \{1, \ldots, n\}$, $|I_j| = 3$ and $\sum_{i \in I_j} a_i = B$? We build the following instance $\mathcal{I}_2$ of our problem with n tasks, such that $x_i = B$, and $m = 3n$ machines with $w_u = 1/a_u$. The period is fixed to $P = 1$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. For $1 \leq i \leq n$, we assign task T_i to the machines of I_i: $q(i, u) = a_u$ for $u \in I_i$, and $q(i, u) = 0$ otherwise. Then, we have $\sum_{1 \leq u \leq m} q(i, u) = \sum_{u \in I_i} a_u = B$, and therefore all the work for task T_i is done. The period of machine M_u is $\sum_{1 \leq i \leq n} q(i, u) \times w_u = a_u / a_u = 1$, and therefore the period of 1 is respected. We have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. Task T_i is assigned to a set of machines, say I_i, such that $\sum_{u \in I_i} q(i, u) = B$, and $q(i, u) \leq a_u$ for all $u \in I_i$. Since all the work must be done, by summing over all tasks, we obtain $q(i, u) = a_u$, and the solution is a 3-partition, which concludes the proof.

3.2 Complexity of the MinPer$(*, f_{i,u}, *)$ Problems

When we consider problems with $f_{i,u}$ instead of f_i, we do not know in advance the number of jobs to be computed by each task in order to have one job exiting the system, since it depends upon the machine on which the task is processed. However, we are still able to solve the problem with general mappings, as explained in Theorem 4. For one-to-many and specialized mappings, the problem is NP-hard with w_u, since it was already NP-hard with f_i in this case (see Theorem 3). We prove that the problem becomes NP-hard with w_i in Theorem 5, which illustrates the additional complexity of dealing with $f_{i,u}$ rather than f_i.

Theorem 4. MinPer$(gen, f_{i,u}, *)$ *can be solved in polynomial time.*

Proof. We modify the linear program (2) of Theorem 2 to solve the case with general failure rates $f_{i,u}$. Indeed, constraint (ii) is no longer valid, since the x_i are not defined before the mapping has been decided. It is rather replaced by constraints (iia) and (iib):

(iia) $\sum_{1 \leq u \leq m} q(n, u) \times (1 - f_{n,u}) = 1$;
(iib) $\sum_{1 \leq u \leq m} q(i, u) \times (1 - f_{i,u}) = \sum_{1 \leq u \leq m} q(i+1, u)$ for each $T_i (1 \leq i < n)$.

Constraint (iia) states that the final task must output one job, while constraint (iib) expresses the number of jobs that should be processed for task T_i, as a function of the number for task T_{i+1}. There are still $n \times m + 1$ variables which are rational, and the number of constraints remains polynomial, therefore this linear program can be solved in polynomial time [11].

Theorem 5. *The* MinPer$(o2m, f_{i,u}, w_i)$ *problem is NP-hard.*

The proof of this theorem is quite involved, and we refer to the companion research report [2] for the details.

However, if the allocation of tasks to machines is known, then we can optimally decide how to share tasks between machines, in polynomial time. We build upon the linear program of Theorem 4, and we add a set of parameters: $a_{i,u} = 1$ if T_i is allocated to M_u, and $a_{i,u} = 0$ otherwise (for $1 \leq i \leq n$ and $1 \leq u \leq m$). The variables are still the period P, and the amount of task per machine $q(i, u)$. The linear program writes:

$$\begin{array}{l} \text{Minimize } P, \text{ subject to} \\ \text{(i) } q(i,u) \geq 0 \text{ for } 1 \leq i \leq n, 1 \leq u \leq m \\ \text{(iia) } \sum_{1 \leq u \leq m} q(n,u) \times (1 - f_{n,u}) = 1 \\ \text{(iib) } \sum_{1 \leq u \leq m} q(i,u) \times (1 - f_{i,u}) = \sum_{1 \leq u \leq m} q(i+1,u) \text{ for } 1 \leq i < n \\ \text{(iii) } \sum_{1 \leq i \leq n} q(i,u) \times w_{i,u} \leq P \text{ for } 1 \leq u \leq m \\ \text{(iv) } q(i,u) \leq a_{i,u} \times F_{\max} \text{ for } 1 \leq i \leq n \text{ and } 1 \leq u \leq m \end{array} \tag{3}$$

We have added constraint (iv), which states that $q(i, u) = 0$ if $a_{i,u} = 0$, i.e., it enforces that the fixed allocation is respected. $F_{\max} = \prod_{1 \leq i \leq n} \max_{1 \leq u \leq m} f_{i,u}$ is an upper bound on the $q(i, u)$ values, it can be pre-computed before running the linear program. The size of this linear program is clearly polynomial in the size of the instance, all $n \times m + 1$ variables are rational, and therefore it can be solved in polynomial time [11].

The linear program of Equation (3) allows us to find the solution in polynomial time, once the allocation is fixed. We also propose an integer linear program (ILP), which computes the solution to the MinPer$(spe, f_{i,u}, w_{i,u})$ problem, even if the allocation is not known. However, because of the integer variables, the resolution of this program takes an exponential time. Note that this ILP can also solve the MinPer$(o2m, f_{i,u}, w_{i,u})$: one just needs to assign a different type to each task. We no longer have the $a_{i,u}$ parameters, and therefore we suppress constraint (iv). Rather, we introduce a set of Boolean variables, $x(u, t)$, for $1 \leq u \leq m$ and $1 \leq t \leq p$, which is set to 1 if machine M_u is specialized in type t, and 0 otherwise. We then add the following constraints:

$$\begin{array}{ll} \text{(iva)} & \sum_{1 \leq t \leq p} x(u,t) \leq 1 \text{ for each machine } M_u \text{ with } 1 \leq u < m \text{ ;} \\ \text{(ivb)} & q(i,u) \leq x(u,t_i) \times F_{\max} \text{ for } 1 \leq i \leq n \text{ and } 1 \leq u \leq m \text{ .} \end{array}$$

Constraint (iva) states that each machine is specialized into at most one type, while constraint (ivb) enforces that $q(i, u) = 0$ when machine M_u is not specialized in the type t_i of task T_i. This ILP has $n \times m + 1$ rational variables, and $m \times p$ integer variables. The number of constraints is polynomial in the size of the instance. Note that this ILP can be solved for small problem instances with ILOG CPLEX (`www.ilog.com/products/cplex/`).

4 Heuristics and Simulations

From the complexity study, we are able to find an optimal general mapping. In this section, we provide practical solutions to solve MinPer($spe, f_{i,u}, w_{i,u}$), which is NP-hard. Indeed, general mappings are not feasible in some cases, since it involves reconfiguring the machines between the execution of two tasks whose type is different. This additional setup time may be unaffordable. We design in Section 4.1 a set of polynomial time heuristics which return a specialized mapping, building upon the complexity results of Section 3. Finally, we present exhaustive simulation results in Section 4.2.

4.1 Polynomial Time Heuristics

Since we are able to find the optimal solution once the tasks are mapped onto machines, the heuristics are building such an assignment, and then we run the linear program of Equation (3) to obtain the optimal solution in terms of $q(i, u)$. The first heuristic is random, and serves as a basis for comparison. Then, the next three heuristics (H2, H3 and H4) are based on an iterative allocation process in two stages. In the first *top-down* stage, the machines are assigned from task T_1 to task T_n depending on their speed $w_{i,u}$: the machine with the best $w_{1,u}$ is assigned to T_1 and so on. The motivation is that the workload of the first task is larger than the last task because of the job failures that arise along the pipeline. In the second *bottom-up* stage, the remaining machines are assigned from task T_n to task T_1 depending on their reliability $f_{i,u}$: the machine with the best $f_{n,u}$ is assigned to T_n and so on. The motivation is that it is more costly to lose a job at the end of the pipeline than at the beginning, since more execution time has been devoted to it. We iterate until all the machines have at least one task to perform. Finally, H5 performs only a *top-down* stage, repetitively. The heuristics are described below.

H1: Random heuristic. The first heuristic randomly assigns each task to a machine when the allocation respects the task type of the chosen machine.

H2: Without any penalization. The *top-down* stage assigns each task to the fastest possible machine. At the end of this stage, each task of the same type is assigned onto the same machine, the fastest. Then, the already assigned machines are discarded from the list. In the same way, the *bottom-up* stage assigns each task of the same type to the same machine starting from the more reliable one. We iterate on these two steps until all machines are specialized.

H3: Workload penalization. The difference with H2 is in the execution of the *top-down* stage. Each time a machine is assigned to a task, this machine is penalized to take the execution of this task into account and its $w_{i,u}$ is changed to $w_{i,u} \times (k+1)$ where k is the number of tasks already mapped on the machine M_u. This implies that several machines can be assigned to the same task type in this phase of the algorithm: if a machine is already loaded by several tasks then we

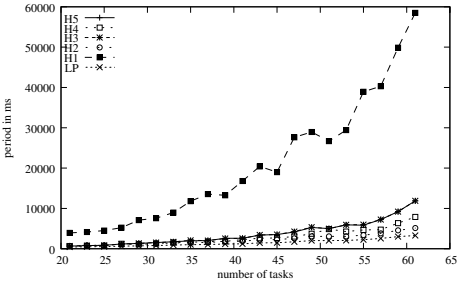

Fig. 1. $m = 20$, $p = 5$.
Heuristics against the linear program.

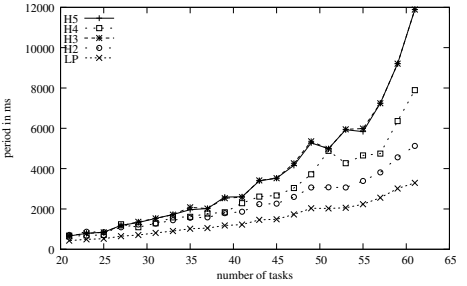

Fig. 2. $m = 20$, $p = 5$.
Without H1.

may find a faster machine and assign it to this task type. The *bottom-up* stage has the same behavior as for H2.

H4: Cooperation work. In this heuristic, a new machine is assigned to each task, depending on its speed, during the *top-down* stage; then the *bottom-up* stage has the same behavior as the heuristic H2.

H5: Focus on speed. The heuristic H5 focuses only on the speed by repeating the *top-down* stage of heuristic H3, until all the machines are allocated to at least one task.

4.2 Simulations

In this section, we evaluate the performance of the five heuristics. The period returned by each heuristic is measured in *ms*. Recall that m is the number of machines, p the number of types, and n the number of tasks. Each point in a figure is an average value of 30 simulations where the $w_{i,u}$ are randomly chosen between 100 and 1000 *ms* (these values are chosen to show the high level of heterogeneity of the machines, and they are randomly chosen since machines and tasks are unrelated), for $1 \leq i \leq n$ and $1 \leq u \leq m$, unless stated otherwise. Similarly, failure rates $f_{i,u}$ are randomly chosen between 0.2 and 10 % unless stated otherwise. Indeed, we observed that failure rates over 10 % do not change the behavior of the heuristics.

Heuristics versus linear program. In this set of simulations, the heuristics are compared to the integer linear program which gives the optimal solution. The platform is such that $m = 20$, $p = 5$ and $21 \leq n \leq 61$. Figure 1 shows that the random heuristic H1 has poor performance. Therefore, for visibility reasons, H1 does not appear in the rest of the figures. Results in Figure 2 show that the heuristics are not far from the optimal. The best heuristics H2 and H4 have a ratio of 1.5 and 2 to the optimal solution. The platform used for these simulations is limited on cases where the integer linear program finds a result. With the same platform but $p = 10$, the percentage of success of the linear program is less than 50% with 61 tasks.

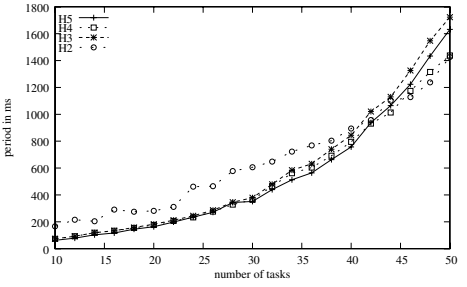

Fig. 3. $m = 50$, $p = 25$.
Heuristics with more machines than tasks.

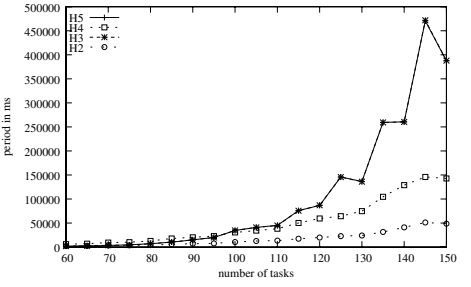

Fig. 4. $m = 50$, $p = 25$.
Heuristics with more tasks than machines.

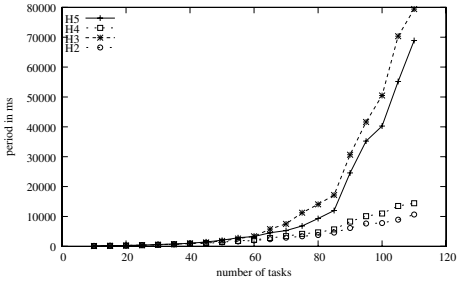

Fig. 5. $m = 40$, $p = 5$.
Small number of types.

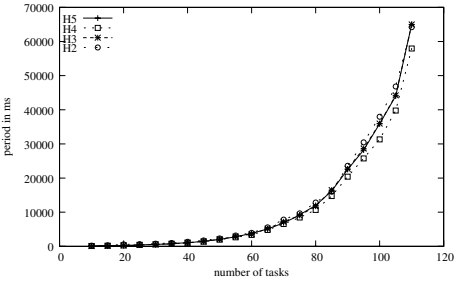

Fig. 6. $m = 40$, $p = 35$.
High number of types.

General behavior of the heuristics. In a second set of simulations, we focus on the behavior of the heuristics alone. First we compare settings with more tasks than machines, or the contrary. In Figure 3, we have $m = 50$, $p = 25$, and $10 \leq n \leq 50$. Results show that H2 is slightly worse than the other heuristics. This lack of performance of H2 becomes even clearer when p is closer to m (see [2] for further results). This is explained by the fact that H2 does not apply any penalization to the machines, thus using a good machine for many tasks of the same type. But when there is less tasks than machines, it is better to dedicate more machines to a given type. However, when the number of tasks is higher than the number of machines, H2 and H4 become clearly the best (see Figure 4). Indeed, at the end of the first stage of allocation, H3 and H5 will almost have used all the machines thus the second stage will not be decisive.

Also, we studied the impact of the number of types, for $m = 40$ and $10 \leq n \leq 110$. In Figure 5, we have $p = 5$, versus $p = 35$ in Figure 6. For $p = 5$, the possibilities to split groups are important. In this case, H2 and H4 are the best heuristics because the workload is shared on a higher number of machines and not only on those efficient for a given task. In the contrary, when the number of types is close to the number of machines ($p = 35$), the number of split tasks decreases. Indeed, each machine must be specialized to one type. In Figure 6, only 5 machines can be used to share the workload once each machine is dedicated to a type, and therefore the performance of the heuristics is pretty much alike.

Summary. Even though it is clear that H1 performs really poorly, the other heuristics can all be the most appropriate, depending upon the situation. If the number of tasks is greater than the number of machines, H2 is the best heuristic; otherwise, H4 becomes better than H2. Further simulations are done in [2], in particular to illustrate the impact of the failure rate on the results. Note that the comparison between the heuristics is made easier if the gap between the number of types and the number of machines is big. Indeed, with a small number of types, the tasks can be split many times because more machines are potentially dedicated to a same type. The choices made by a heuristic either to split a task or not have more impact on the result.

5 Conclusion

In this paper, we investigate the problem of maximizing the throughput of coarse-grain pipeline applications where tasks have a type and are subject to failures, with different mapping strategies (one-to-many, specialized or general). A task can be distributed on the platform so as to balance workload between the machines. From a theoretical point of view, an exhaustive complexity study is proposed. We prove that an optimal solution can be computed in polynomial time in the case of general mappings whatever the application/platform parameters, and in the case of one-to-many and specialized mappings when the failure rates only depend on the tasks, while the optimization problem becomes NP-hard in any other cases. Since general mappings do not provide a realistic solution because of unaffordable setup times when reconfiguration occurs, we propose to solve the specialized mapping problem by designing several polynomial heuristics. An exhaustive set of simulations demonstrate the efficiency of the heuristics: some of them return a throughput close to the optimal, while random mappings never give good solutions.

As future work, we plan to investigate other objective functions, such as the mean time to output one job out of the system, or other models: the failure rate associated to the task and/or the machine could be correlated with the time required to perform that task.

Acknowledgment. A. Benoit is with the Institut Universitaire de France. This work was supported in part by the ANR *StochaGrid* and *RESCUE* projects.

References

1. Bahi, J., Contassot-Vivier, S., Couturier, R.: Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In: International Parallel and Distributed Processing Symposium, IPDPS 2003 (April 2003)
2. Benoit, A., Dobrila, A., Nicod, J.M., Philippe, L.: Workload balancing and throughput optimization for heterogeneous systems subject to failures. Research report, INRIA, France (February 2011), `http://graal.ens-lyon.fr/~abenoit/`
3. Blażewicz, J., Drabowski, M., Weglarz, J.: Scheduling multiprocessor tasks to minimize schedule length. IEEE Trans. Comput. 35, 389–393 (1986)

4. Cirne, W., Brasileiro, F., Paranhos, D., Góes, L.F.W., Voorsluys, W.: On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. Parallel Computing 33(3), 213–234 (2007)
5. Descourvières, E., Debricon, S., Gendreau, D., Lutz, P., Philippe, L., Bouquet, F.: Towards automatic control for microfactories. In: IAIA 2007, 5th Int. Conf. on Industrial Automation (2007)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability, a Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York (1979)
7. Gröflin, H., Klinkert, A., Dinh, N.P.: Feasible job insertions in the multi-processor-task job shop. European J. of Operational Research 185(3), 1308–1318 (2008)
8. Jalote, P.: Fault Tolerance in Distributed Systems. Prentice-Hall, Englewood Cliffs (1994)
9. Litke, A., Skoutas, D., Tserpes, K., Varvarigou, T.: Efficient task replication and management for adaptive fault tolerance in mobile grid environments. Future Generation Computer Systems 23(2), 163–178 (2007)
10. Parhami, B.: Voting algorithms. IEEE Trans. on Reliability 43(4), 617–629 (1994)
11. Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency. Algorithms and Combinatorics, vol. 24. Springer, Heidelberg (2003)
12. Tanaka, M.: Development of desktop machining microfactory. Journal RIKEN Rev 34, 46–49 (2001) iSSN:0919-3405
13. Verettas, I., Clavel, R., Codourey, A.: Pocketfactory: a modular and miniature assembly chain including a clean environment. In: 5th Int. Workshop on Microfactories (2006)
14. Weissman, J.B., Womack, D.: Fault tolerant scheduling in distributed networks (1996)
15. West, R., Zhang, Y., Schwan, K., Poellabauer, C.: Dynamic window-constrained scheduling of real-time streams in media servers (2004)
16. West, R., Poellabauer, C.: Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In: Proc. of the 21st IEEE Real-Time Systems Symp., pp. 239–248. IEEE, Los Alamitos (2000)
17. West, R., Schwan, K.: Dynamic Window-Constrained Scheduling for Multimedia Applications. In: ICMCS, vol. 2, pp. 87–91 (1999)
18. Wieczorek, M., Hoheisel, A., Prodan, R.: Towards a general model of the multi-criteria workflow scheduling on the grid. Future Gener. Comput. Syst. 25(3), 237–256 (2009)
19. Yu, J., Buyya, R.: A taxonomy of workflow management systems for grid computing. Research Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia (April 2005)