Throughput optimization for micro-factories subject to failures

Anne Benoit, Alexandru Dobrila, Jean-Marc Nicod, Laurent Philippe

► To cite this version:

Anne Benoit, Alexandru Dobrila, Jean-Marc Nicod, Laurent Philippe. Throughput optimization for micro-factories subject to failures. ISPDC'2009, 8th Int. Symposium on Parallel and Distributed Computing, 2009, Portugal. pp.11–18. hal-00563300

HAL Id: hal-00563300
https://hal.science/hal-00563300v1

Submitted on 4 Feb 2011

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

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

Throughput optimization for micro-factories subject to failures

Anne Benoit
ENS Lyon, Université de Lyon, France
LIP laboratory (ENS, CNRS, INRIA, UCBL)
Anne.Benoit@ens-lyon.fr

Alexandru Dobrila, Jean-Marc Nicod and Laurent Philippe
Laboratoire d'Informatique de Franche-Comté,
Université de Franche-Comté, France
adobrila,jmnicod,lphilippe@lifc.univ-fcomte.fr

***Abstract*—In this paper, we study the problem of optimizing the throughput for micro-factories subject to failures. The challenge consists in mapping several tasks onto a set of machines. The originality of our approach is the failure model for such applications in which tasks are subject to failures rather than machines. If there is exactly one task per machine in the mapping, then we prove that the optimal solution can be computed in polynomial time. However, the problem becomes NP-hard if several tasks can be assigned to the same machine. Several polynomial time heuristics are presented for the most realistic *specialized* setting, in which tasks of a same type can be mapped onto the same machine. Experimental results show that the best heuristics obtain a good throughput, much better than the throughput obtained with a random mapping. Moreover, we obtain a throughput close to the optimal solution in the particular cases on which the optimal throughput can be computed.**

I. Introduction

Distributed systems provide a support to tolerate faults but their correct management also implies to take faults into account. Standard distributed systems mainly focus on processor dependent faults. We commonly assume that faults are generated by the execution platform and thus that the fault model must be linked to the processors, or more generally to the resources that perform the tasks. In this case, a stochastic fault model that defines the fault probability is usually attached to the processor. This model fits distributed computing environments such as parallel platforms where failures come from the nodes of the platform.

If we look however at a more general definition of a distributed system, we can note that this model does not always fit. In some distributed platforms the fault model may be attached to the task rather than to the processor or node. For example, in production systems, a task may be complex to perform - for instance due to some hard manipulation - with an impact on its success ratio. If the same robot is able to perform different tasks, it may generate less faults on simple tasks than on difficult ones.

In this paper we are interested in studying the impact of a fault model linked to the tasks. The application context is more a production system than a distributed computing system. Our specific use case is a micro-factory but the results presented in this paper are more generally applicable to distributed production systems or to distributed systems where the fault probability is attached to tasks instead of resources.

Micro-factories are production units designed to produce pieces composed of micro-metric elements [**?**]. Today's micro-factories are composed of micro-robots able to carry out basic operations through elementary actuators as piezo-electric beams (e.g. for gripping), stick-slip systems, etc. As these robots are usually teleoperated by a human operator only simple tasks can be done. To perform more complex operations and to improve their efficiency, micro-factories need to be automated and robots grouped in cells. Then cells will be put together and they will cooperate to produce complex assembled pieces, as it is done for macroscopic productions. Due to the pieces, actuators and cells size, it is however impossible for human operators to directly interfere with the physical system. So it needs a highly automated command. The complexity of this command makes it mandatory to develop a distributed system to support this control. So, the cell group results in a distributed system that is very similar to a distributed computing platform. However, at this scale the physical constraints are not totally controlled so there is a need to take faults into account in the automated command. As previously explained, the fault model that we consider in this work differs from the standard fault model used in computing systems.

The main issue for fault tolerant systems [1] is to overcome the failure of a node, a machine or a processor. To deal with those faulty machines the most common method used in distributed systems is to replicate [2] the data. Those models assume that failures are attached to a machine. So the probability to get one product as a result is highly increased when the task is replicated on several machines. Once all the replicated jobs are done, a vote algorithm [3] is often used to decide which result is the right one. In real time systems, another model called Window-Constrained [4] model can be used. In this model one considers that, for y messages, only x ($x \leq y$) of them will reach their destination. The y value is called the Window. The looses are not considered as a failure but as a guarantee: for a given network a Window-Constrained Scheduling [5], [6] can guarantee that no more than x messages will be lost for every y sent messages. In this paper, the Window-Constrained based failure model is adapted to a distributed system, the micro-factory. So the issue is to guarantee the output of a given number of products. With failures attached to tasks, we can compute the number

of products needed as input of the system and guarantee the output for the desired number of products.

The paper is organized as follows. Section II gives the characteristics and a more formal presentation of the context of micro-factories and of the failure model. Section III presents the optimization problems tackled in the paper. The complexity study and results are given in Section IV. Heuristics to solve the problem are proposed in Section V and simulation results for these heuristics are given and commented in Section VI. Finally, we conclude in Section VII.

II. Framework

We outline in this section the characteristics of the applicative framework and target platform. Finally, we describe and motivate the failure model that we use in this work.

A. Applicative framework

We consider a set $\mathcal{N}$ of n tasks: $\mathcal{N} = \{T_1, T_2, \dots, T_n\}$. Each task T_i ($1 \leq i \leq n$) is applied successively on a set of products, numbered from 1 to x_{in}. We wish to produce x_{out} products as an output, and the total number of products x_{in} being processed may depend on the allocation ($x_{in} \geq x_{out}$, losses due to failure as explained later in Section II-C). Note that all products are identical. When the context is not ambiguous, we may also design task T_i by i for clarity, as for instance in the figures.

A type is associated to each task as the same operation may be applied several times to the same product. Thus, we have a set $\mathcal{T}$ of p task types with $n \geq p$ and a function $t : [1..n] \rightarrow \mathcal{T}$ which returns the type of a task: $t(i)$ is the type of task T_i, for $1 \leq i \leq n$.

The application is a directed acyclic graph (DAG) in which the vertices are tasks, and edges represent dependencies between tasks. An example of application with $n = 5$ tasks is represented on Figure 1. In the top branch of the DAG, we need to finish the processing of task T_1 on one product before proceeding to task T_2. The join to task T_4 corresponds to the merge of two products, which produces a new unit of product composed of the two. Typically one instance of product from each predecessor in the graph is required to process with the joining task. Note that forks cannot be considered in this context as the output of one task is a physical component that cannot be split in two. Unlike data that can be easily replicated at every step of a DAG, an instance of a physical component is the result of all the preceding tasks and cannot be duplicated as it is material.

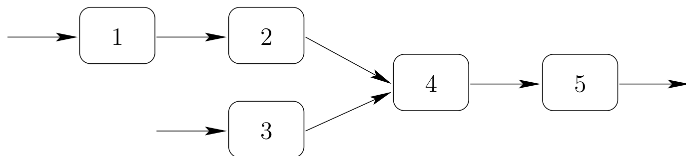

Figure 1. Example of application.

B. Target platform

The platform consists in a set $\mathcal{M}$ of m machines: $\mathcal{M} = \{M_1, M_2, \dots, M_m\}$. All machines can be interconnected: the platform graph is a complete graph. A machine handles some of the tasks at a given speed: machine M_u can perform the task T_i onto one product in a time $w_{i,u}$. We also consider that tasks of the same type have the same execution time on a given machine, since they correspond to the same action to be performed on the products. Thus, we have:
$\forall i, i' \in [1, n], \forall u \in [1, m], t(i) = t(i') \Rightarrow w_{i,u} = w_{i',u}$

We neglect the communication time required to transfer a product from one machine to another. If a communication may not be negligible, we can always model it as a particular task with a dedicated machine (the machine responsible of the transfer of the product).

We are interested in producing the desired number of products rather than producing a particular instance of a product. So we consider that products are not identified: two products, on which the same sequence of tasks has been done, are exactly similar and we can use one or the other indifferently for further operations.

C. Failure model

An additional characteristic of our framework is that tasks are subject to failure. It may happen that a product is lost or damaged while a task is being executed on this product. For instance electrostatic strength may be accumulated on the actuator, and thus the piece will be pushed away rather than caught. Indeed, we work at a scale such that these electrostatic strengths are stronger than gravity.

Due to our application setting, we deal only with transient failures, as defined in [7]. The tasks are failing for some of the products, 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 products to be processed and the unability to finish them.

One classical technique used to deal with failures is replication [2]. However, while replication is very useful for hardware failures of machines, we cannot use it in our framework since the products are not a data such as a numerical image that we need to process, but it is a physical object. Thus, the only solution consists in processing more products than needed, so that at the end, the required number of finished products are output.

The failure rate of task T_i is characterized by the percentage of failure for this task. More precisely, the failure is denoted $f_i = \frac{a_i}{b_i}$, where a_i is the number of products that fail each time b_i products have been processed. $r_i = b_i - a_i$ is the number of successful products, and b_i is also called *period* of task T_i.

We enforce that two tasks of similar type are likely to fail at the same rate with the following equation:

$$\forall i, i' \in [1, n] \quad t(i) = t(i') \Rightarrow f_i = f_{i'}.$$

Since we advocate the computation of more products than needed, we explain in the following how to compute the

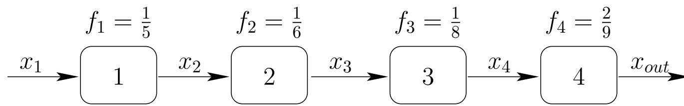

Figure 2. Example of a linear chain application with failure.

number x_{in} of products that should be processed in order to get x_{out} products as an output, and we illustrate it on the example of Figure 2. For instance task T_2 has one failure every 6 products that are being processed by this task. Given these failure rates, the number of products that should be given as an input to task T_i in order to have x_{out} products out of the system is denoted x_i. Thus, $x_{in} = \max_{1 \leq i \leq n} x_i$.

The computation of x_i is done backward: if we know the number of products that should be output by task T_i and its failure rate, we can compute the number of input products that should be given to this task to guarantee this output. As we work only with linear chain and in-trees, each task has an unique outgoing edge, thus the number of products to be output by T_i is x_{i+1} (or x_{out} for T_4 in the example).

To determine x_i, we need to sum both the number of products which will be successfully processed (i.e., x_{i+1}), and the number of products which fail during the processing phase. Thus we must compute the number of periods of the task T_i, which is an integer number being greater than the number of output products divided by the number of successful products computed each period: $\left\lceil \frac{x_{i+1}}{r_i} \right\rceil$. In the worst case, failures occur for the first a_i products of the period, thus the number of products to be computed and which will fail is $a_i \times \left\lceil \frac{x_{i+1}}{r_i} \right\rceil$.

Finally we can deduce the total number of products that should be computed:

$$x_i = x_{i+1} + a_i \times \left\lceil \frac{x_{i+1}}{r_i} \right\rceil \quad (1)$$

As an extension of this formula, we can also deduce the completion time $L_{x_{i+1},i}$ needed to exit x_{i+1} products out of task T_i, once it has been assigned to a machine, M_u. The completion time $L_{x_{i+1},i}$ is the maximum between the time needed to compute x_i products on task T_i (i.e., $x_i \times w_{i,u}$) and the sum of the time to output x_i product out of the task T_{i-1} (i.e., $L_{x_i,i-1}$) and the completion time of the last product on task T_i (i.e., $w_{i,u}$).

$$L_{x_{i+1},i} = \max \left(L_{x_i,i-1} + w_{i,u}, x_i \times w_{i,u} \right) \quad (2)$$

III. Optimization problems

Now that the framework has been clarified, we formalize in this section the various optimization problems that we wish to solve. Our goal is to assign tasks to machines so as to optimize some key performance criteria. The solution to a problem is thus an allocation function $a : [1..n] \to [1..m]$ which returns for each task the machine on which it is executed. Thus, if $a(i) = u$, task T_i is executed on machine M_u, and the processing of one product for this task takes a time $w_{i,u}$.

We first discuss the objective criteria that we want to optimize. Then we introduce the different rules of the game that can be used in the definition of the allocation function a. Finally, Section III-C gives a summary of all problem variants, combining framework characteristics and rules of the game. The complexity of these various problems is discussed in Section IV.

A. *Objective function*

In our framework, several objective functions could be optimized. For instance, one may want to produce a mapping of the tasks on the machines as reliable as possible, i.e., minimize the total number of products to input in the system, x_{in}. Rather, we consider that products are cheap, and we focus on a performance criteria, the throughput. The goal is to maximize the number of products processed per time unit, making abstraction of the initialization and clean-up phases. This objective is important when a large number of products must be produced.

Rather than maximizing the throughput of the application, we rather deal with the *period*, which is the inverse of the throughput. First we need to introduce the fractional number $\overline{x_i}$, which is the average number of products required to output one product out of the system for task T_i. Similarly to the computation of the x_i performed in Section II-C, we can compute the $\overline{x_i}$ recursively for any application DAG, setting the number of final products $x_{out} = 1$. Indeed, if task T_i needs to output $\overline{x_{i+1}}$ products, then $\overline{x_i} = \frac{b_i}{r_i} \times \overline{x_{i+1}}$ (the fraction represents the number of products needed per successful product). Starting from the nodes with no successor (and thus $\overline{x_{i+1}} = x_{out} = 1$), we can then compute $\overline{x_i}$ for all i. Note that $\overline{x_i} \leq x_i$ since x_i is an upper integer part which accounts for the worst case failures.

The computation of $\overline{x_i}$ and x_i for the example of Figure 2 is illustrated in Table I. For instance, $\overline{x_4} = 9/(9-2) = 9/7 \simeq 1.3$.

Table I
Values of x_i and $\overline{x_i}$ for the example of Figure 2, with $x_{out} = 1$.

Task number	1	2	3	4
x_i	7	5	4	3
$\overline{x_i}$	$\simeq 2.2$	$\simeq 1.8$	$\simeq 1.5$	$\simeq 1.3$

We are now ready to define the *period* of a machine: it is the time needed by a machine to execute all the tasks allocated onto this machine in order to produce one final product out of the system. Formally, we have

$$period(M_u) = \sum_{a(i)=u} \overline{x_i} w_{i,u} \quad (3)$$

The period of machine M_u is the sum, for each task allocated to that machine, of the average number of products ($\overline{x_i}$) needed to output one product, multiplied by the speed ($w_{i,u}$) of that task onto that machine. The slowest machine will slow down the whole application, thus we aim at minimizing the largest machine period. The machines realizing this maximum

are called *critical machines*. If M_c is a critical machine, then $period = period(M_c) = \max_{M_u \in \mathcal{M}} period(M_u)$.
Note that minimizing the period is similar to maximizing the throughput.

B. Rules of the game

In this section, we classify several variants of the optimization problem that has been introduced. For *one-to-one* mappings, we enforce that a single task must be mapped onto each machine. Then we consider the case of specialized machines: several tasks of the same type can be mapped onto the same machine; such mappings are called *specialized* mappings. Finally, *general* mappings have no constraints: any task (no matter the type) can be mapped on any machine.

1) One-to-one mappings: In this first class of problems, a single task is mapped on each machine. This rule of the game is enforced with the following constraint, meaning that a machine cannot compute two different tasks:

$$\forall 1 \leq i, i' \leq n \quad i \neq i' \Rightarrow a(i) \neq a(i')$$

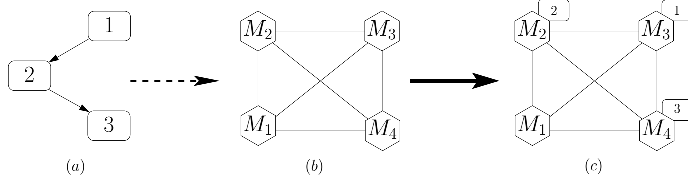

Figure 3. One-to-one mapping.

On Figure 3, we have an application graph (a) that must be mapped on a platform graph (b). The result is shown in (c) where we can see that one machine can handle only one task. Thus this mapping is quiet restrictive because we must have at least as many machines as tasks.

2) Specialized mappings: We have dedicated machines that can realize only one type of tasks. But task types are not dedicated to machines, so two machines may compute different tasks of the same type.

For instance, let us consider five tasks T_1, T_2, T_3, T_4, T_5 with the following types: $t(1) = t(3) = t(5) = 1$ and $t(2) = t(4) = 2$. If the machine M_3 computes task T_1, it could also execute T_3 and T_5 but not T_2 and T_4. As types are not dedicated to machines, T_5 could also be assigned to another machine, for instance M_1. This situation is described on Figure 4.

The following constraint expresses the fact that a machine cannot compute two tasks of different types:

$$\forall 1 \leq i, i' \leq n \quad t(i) \neq t(i') \Rightarrow a(i) \neq a(i')$$

3) General mappings: A machine can compute any task regardless of its type, thus there are no constraints.

An example of this case is shown on Figure 5.

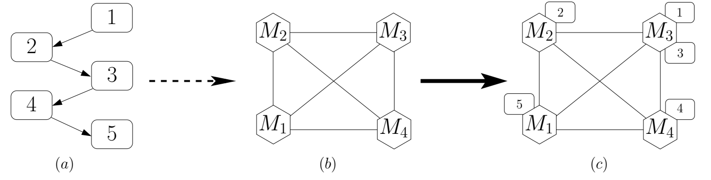

Figure 4. One machine can do different tasks if they are of the same type. Here the type of tasks are the following : t(1)=t(3)=t(5)=1 and t(2)=t(4)=2.

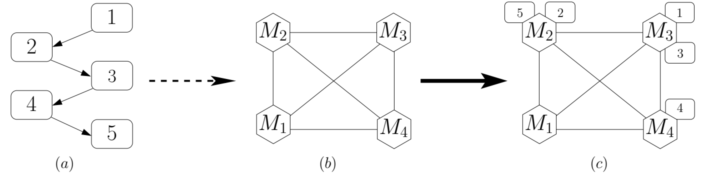

Figure 5. One machine can handle any task. Here the type of tasks are the following : t(1)=t(3)=1, t(2)=t(4)=2 and t(5) = 3.

C. Problem classification

We summarize in this section the optimization problems which arises from our application. The two important parameters of a problem are :

- the rules of the game (*one-to-one* or *specialized* or *general* mapping);
- and the degree of heterogeneity of machines and tasks: the time to compute one product of task T_i on machine M_u may be identical for each task/machine (w), depend only on the task (w_i) or the machine (w_u), or be fully general (w_{iu}).

IV. Complexity results

Complexity results are classified depending on the mapping rules. We start with one-to-one mappings, then we focus on specialized and general ones. Finally, we compare one-to-one mappings with general and specialized ones.

A. Complexity of one-to-one mappings

Theorem 1. *Given an application and a set of machines, finding the one-to-one mapping which maximizes the throughput can be done in polynomial time.*

Proof: We can compute the average number of products $\overline{x_i}$ needed to output one product out of task T_i, as explained in Section III-A. Since the mapping is required to be one-to-one, we create a bipartite graph with one node per task on one side, one node per machine on the other side. The cost of an edge from task T_i to machine M_u is then set to $\overline{x_i} w_{i,u}$, which corresponds to the period of machine M_u if task T_i is assigned to this machine. Since the period of the mapping is the maximum of the periods of each machine, the problem is equivalent to a maximum weight matching in bipartite graphs, which can be found in polynomial time, for instance using the Hungarian method [8], [9].

■

B. Complexity of specialized and general mappings

Theorem 2. *Finding the optimal specialized or general mapping is NP-hard, even with constant processing costs w.*

Proof: We consider the following decision problems: given a period K, is there a general/specialized mapping whose period does not exceed K? 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 2-PARTITION [10]. Let $\mathcal{I}_1$ be an instance of 2-PARTITION: given a set $\{a_1, ..., a_n\}$ of n integers, does it exist a subset I such that $\sum_{i \in I} a_i = \frac{1}{2}\sum_{1 \leq j \leq n} a_j$? We construct the instance $\mathcal{I}_2$ with n tasks ordered as a linear chain, 2 machines, and $w = 1$. All tasks are of the same type, thus there is no difference between general and specialized mappings, and both problems are tackled simultaneously. We assume that $a_1 \geq a_2 \geq ... \geq a_n$ (the sort can be done in polynomial time), and then we fix:

- $f_n = \frac{a_n - 1}{a_n}$; $\forall 1 \leq i \leq n-1, f_i = \frac{a_i - a_{i+1}}{a_i}$;
- $K = \frac{1}{2}\sum_{1 \leq j \leq n} a_j$;

First we prove by induction that $\overline{x_i} = a_i$ for $1 \leq i \leq n$. For $i = n$, we have $\overline{x_n} = 1 \times b_n / r_n = a_n$. For $1 \leq i \leq n-1$, if $\overline{x_j} = a_j$ for $j > i$, then $\overline{x_i} = \overline{x_{i+1}} \times b_i / r_i = a_{i+1} \times a_i / a_{i+1} = a_i$.

The size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. Suppose that $\mathcal{I}_1$ has a solution I. We construct the allocation function a such that: $\forall i, a(i) = 1 \iff i \in I$. Since $w = 1$ and $\overline{x_i} = a_i$ for all i, the period of the mapping is thus $P = \max\{\sum_{i \in I} a_i, \sum_{i \notin I} a_i\}$, that means $P = K$ and $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. Let $I = \{a_i | a(i) = 1\}$. By hypothesis, we have $\sum_{i \in I} a_i \leq K$ and $\sum_{i \notin I} a_i = 2K - \sum_{i \in I} a_i \leq K$. We can conclude that $\sum_{i \in I} a_i = \frac{1}{2}\sum_{1 \leq j \leq n} a_j$. Then, $\mathcal{I}_1$ has a solution. This concludes the proof. ■

C. Comparison of mapping rules

In this section, we compare the three mapping strategies, namely one-to-one, specialized and general mappings. The first thing that we want to point out is that one-to-one mappings are a particular case of specialized mappings, which are themselves a particular case of general mapping. Thus, an optimal one-to-one mapping cannot be better than an optimal specialized mapping, which itself cannot be better than a general mapping.

Why not restrict to general mappings? The problem of these general mappings is that they are not realistic, because if a machine is processing tasks of different types, one needs to reconfigure the machine between operations, and this cost is unaffordable in most micro-factories. Thus, in the following, an emphasis is given to one-to-one and specialized mappings.

Since the optimal one-to-one mapping can be found in polynomial time (see Theorem 1), why not restrict to such mappings? The problem arises when $m \leq n$, i.e., there are many tasks and not so many machines. In such cases, it is mandatory to execute several tasks on the same machine. When there are enough machines ($m \geq n$), one-to-one allocations are a good way to tackle the problem (see the following theorem), but they can be arbitrarily worse than a specialized allocation in the general case.

Theorem 3. *If $m \geq n$, and for problems with w_i ($w_{i,u} = w_i$ for $1 \leq u \leq m$), there is an optimal specialized or general mapping which performs a one-to-one allocation of tasks onto machines. In other words, one-to-one mappings are dominant in this case.*

Proof: The proof is simply done by an exchange argument. Suppose that there is an optimal mapping which is not a one-to-one mapping. For instance, tasks T_i and T_j are mapped onto the same machine, M_u. Since $m \geq n$, there is at least one free machine, say M_v, and the period can be decreased from $\overline{x_i}w_i + \overline{x_j}w_j$ to $\max(\overline{x_i}w_i, \overline{x_j}w_j)$ if task T_j is assigned to M_v instead of M_u. This concludes the proof. ■

Note that this is not true if the completion time also depends on the processor. For instance, consider a problem with w_u ($w_{i,u} = w_u$ for $1 \leq i \leq n$) in which there is one machine with $w_1 = 1$ and a second one with $w_2 = K$, where K is arbitrary large. If the application consists in two tasks of same type with no failures, then the optimal throughput can be obtained by mapping both tasks onto machine 1, resulting in a period of $1 + 1 = 2$, while a one-to-one mapping must use machine 2 and thus its period cannot be better than K, which can be arbitrarily greater than 2.

V. HEURISTICS

As explained in Section IV-C, general mappings are not realistic in the context of micro-factories, because of the unaffordable reconfiguration costs. When the number m of machines is greater than the number p of task types, it is always possible to find a specialized mapping, since each machine is able to process all the tasks of a same type. The key point is thus to find m (or less) groups of tasks of the same type to be assigned to the m machines of the platform. The best solution may be a one-to-one mapping (cases in which such mappings are optimal, see Theorem 3).

As shown before, finding the optimal specialized mapping is NP-hard (see Theorem 2). Thus, we present in the following five heuristics that returns a mapping, by grouping tasks of same type onto machines.

H1: *Random heuristic* — The m groups are made by using a random assignment function. We randomly choose p tasks, such that $t(i) \neq t(i')$ for all chosen tasks T_i and $T_{i'}$, and we randomly assign them to p machines of the platform (recall that p is the number of task types). Then we can randomly assign the rest of the tasks T_j either on a machine which is free or already specialized to the same task type $t(j)$.

H2: *Task group heuristic* — p groups are made by assigning all the tasks of the same type to the same group. While the number of groups is less than m, the number of machines, the group which consists in the larger number of tasks is divided into two

groups to balance the workload. Then, an assignment of groups to machines is performed using the *one-to-one* mapping algorithm.

H3: *Binary search heuristic 1* — This heuristic aims at optimizing the potential of the machines, i.e., the goal is to assign to each machine a set of tasks for which it is efficient. Thus, we start by sorting, for each machine M_u, the set of $w_{i,u}$, for $1 \leq i \leq n$, in ascending order. Then, $rank_{i,u}$ represents the rank of T_i in the ordered set for M_u.
The heuristic performs a binary search on the period between 0 (best case) and the time required to perform sequentially all the tasks on a machine (worst case). For each value of the search, all tasks are assigned greedily (from T_1 to T_n) to machines. For task T_i, we try to assign it to a machine such that $rank_{i,u}$ is minimum. If the rank equals one, this means that the potential of M_u for this task is optimal. In case of equality (several machines of identical rank for T_i), machines are sorted by non-decreasing values of $w_{i,u}$. Of course, the assignment can be done only if the machine was not already specialized to a type which is different from $t(i)$, and if the fixed period is not exceeded. Otherwise we try to assign T_i to the next machine, according to their priority order for this task. If no machine is able to process T_i, then no assignment is found and we try a larger period. If all tasks can be correctly assigned, we try a smaller period.

H4: *Binary search heuristic 2* — Similarly to the previous heuristic, H4 performs a binary search on the period. However, the greedy assignment procedure is different. For each task T_i, the heuristic tries to assign it to the machine M_u which minimizes $w_{i,u}$, if the machine is not already specialized to a type which is different from $t(i)$, and the period is not exceeded. No ranking is computed, the idea is to forget about the potential of the machines, but simply try to execute each task as fast as possible, thus using efficient machines.

H5: *Binary search heuristic 3* — This heuristic is the same as H4 except that, for the assignment, the machines are sorted by their heterogeneity level in descending order. The idea is to preserve homogeneous machines for the last tasks. The heterogeneity level of M_u is computed as the standard deviation of its $w_{i,u}$ values. Each task is assigned to the most heterogeneous machine capable of handling it. Note that for this heuristic, slow machines may be used instead of powerful ones, because of their heterogeneity level.

The next section presents experimental results that compare these heuristics.

VI. Experiments

In this section, we compare together the 5 heuristics that give sub-optimal solutions to the specialized mapping problem with $w_{i,u}$. The performance of each heuristic is measured by its period in *ms* (see Section III-A). Only the most significant results are presented in this paper. The full set of experiments is available in [11].

Recall that m is the number of machines, p the number of types, and n the number of tasks. Each point in the figures is an average value of 50 simulations where the $w_{i,u}$ are randomly chosen between 100 and 1000 *ms*, for $1 \leq i \leq n$ and $1 \leq u \leq m$. Similarly, failure rates f_i ($1 \leq i \leq n$) are randomly chosen between 0.5 and 2 % (i.e., 1/200 and 1/50).

A. First set of experiments: m and p fixed

In the first set of experiments, m and p are fixed, and we plot the period for each heuristic as a function of the number of tasks n. Figure 6 shows that the random heuristic H1 returns very large periods, compared to the 4 other heuristics. This remains true for all experiments: H1 shows very poor performance. Thus H1 is removed from the curves for readability.

To analyze the impact of the platform heterogeneity ratio, the same experiments ($m = 10$ and $p = 5$) have been run with a smaller duration interval ($w_{i,u}$ between 100 and 200 *ms*) in order to simulate less heterogeneous platforms. Results of these simulations are presented in Figures 7 and 8. In both results, H3 is clearly the best and the performances of H2, H4 and H5 are equivalent. These two results are slightly alike except for the scale. We can though deduce that the heterogeneity ratio of the platform has little influence concerning the comparison of these heuristics. In the following, we thus present results only in the heterogeneous setting with $w_{i,u}$ varying between 100 and 1000.

Figures 9 and 10 show that the performance of H2 is very similar to that of H4 and H5 when the difference between the number of machines (respectively 20 and 100) and the number of types (respectively 18 and 90) is small. Indeed, H2 tries to use all the machines and thus it splits the groups until it has as many groups as machines. In these experiments, the way the groups are split does not influence the performance so much because only 2 (respectively 10) extra groups will be created. H3 is clearly the best heuristic in such a case. With H3, we optimize the potential of each machine so we make the best use of a given platform.

On the contrary, when the number of machines m is much greater than the number of types p, the performance of H2 decreases, as we can see in Figures 11 and 12. In these experiments, H3 and H4 are both reaching a satisfying period, while H5 is slightly less good.

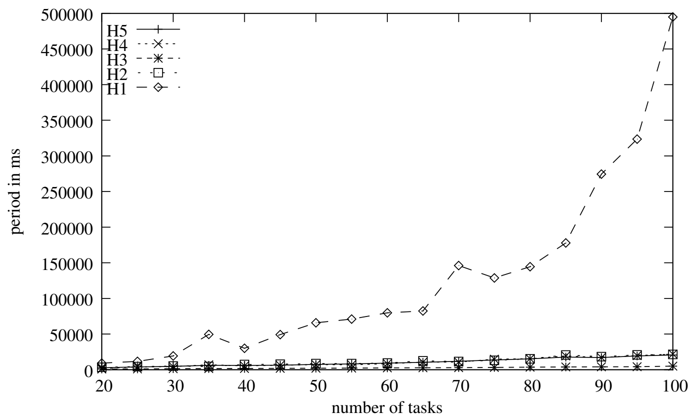

Figure 6. $m = 10$, $p = 5$.
Behavior of H1.

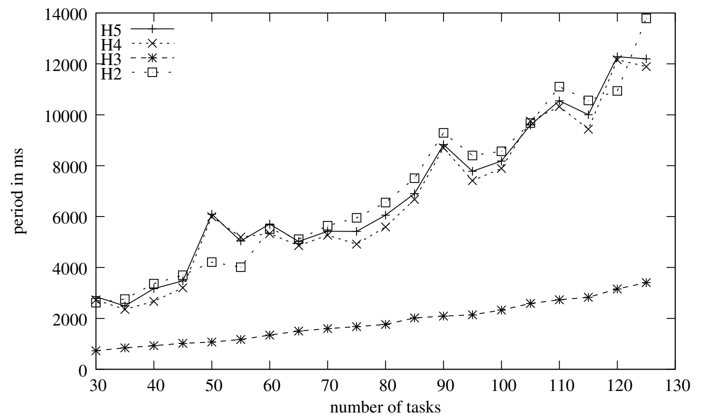

Figure 9. $m = 20$, $p = 18$.
m close to p.

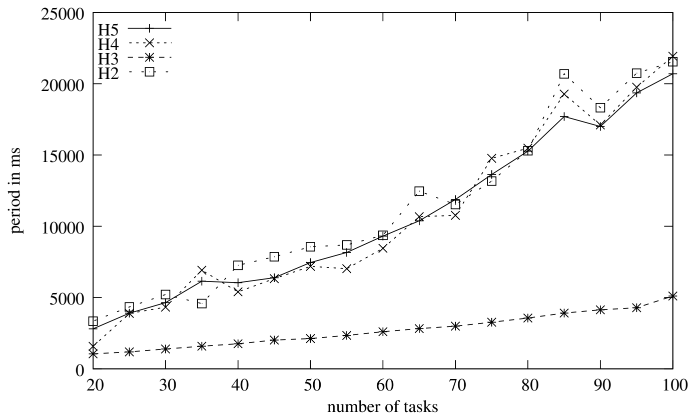

Figure 7. $m = 10$, $p = 5$.
Heterogeneous setting: $100 < w_{i,u} < 1000$.

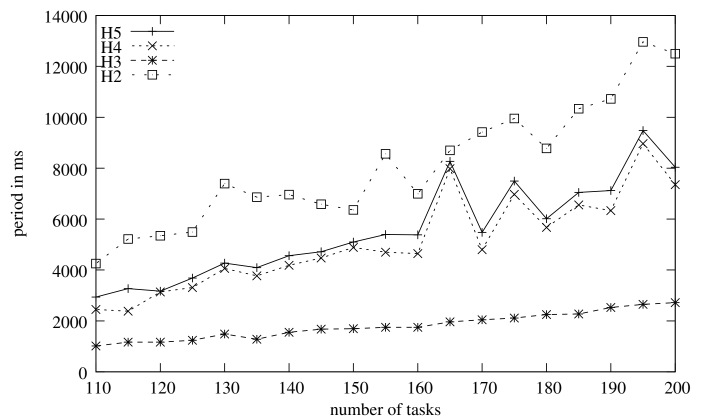

Figure 10. $m = 100$, $p = 90$.
m close to p.

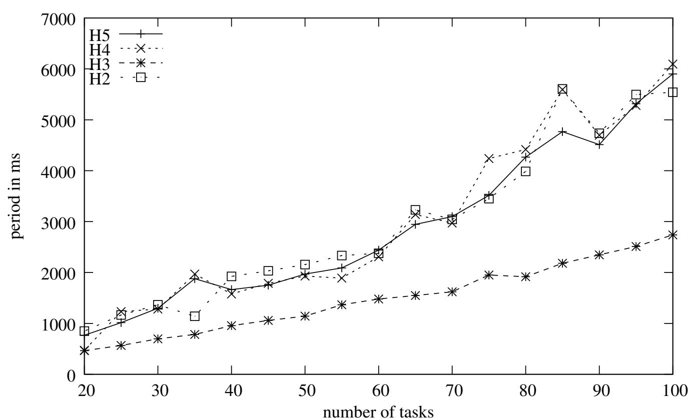

Figure 8. $m = 10$, $p = 5$.
Homogeneous setting: $100 < w_{i,u} < 200$.

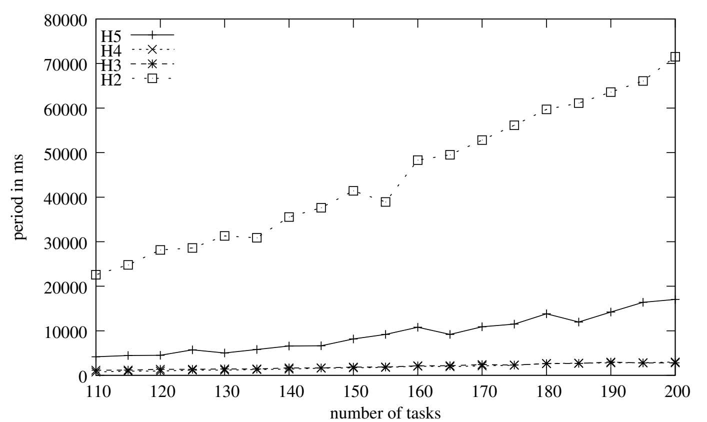

Figure 11. $m = 100$, $p = 5$.
m much greater than p.

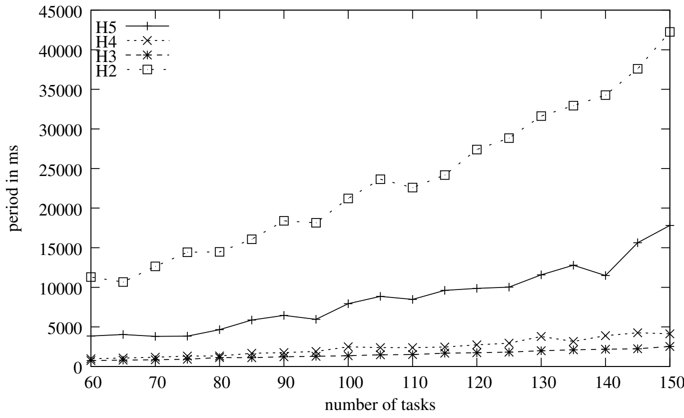

Figure 12. $m = 50$, $p = 5$.
m much greater than p.

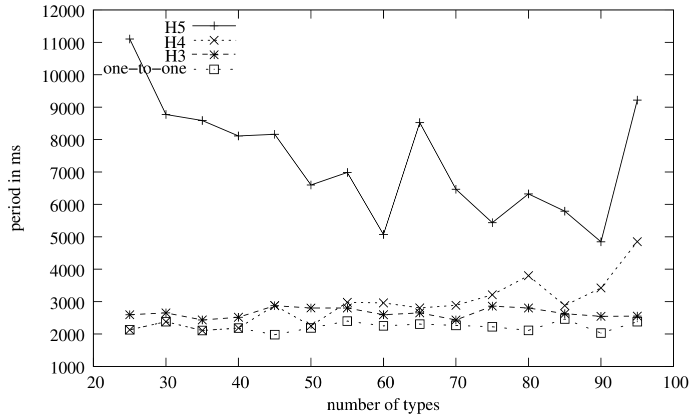

Figure 13. $m = n = 100$, with $w_{i,u} = w_{i,u'}$.
Experiment with fixed m and n.

B. *Second set of experiments: m and n fixed*

In the last experiment (Figure 13), we fix $m = n = 100$, and we plot the period as a function of the number of types p. Moreover, we randomly chose values $w_{i,u}$ such that the duration of a task is machine-independent ($w_{i,u} = w_{i,u'}$ for $1 \leq u, u' \leq m$). In this case, we know that there is an optimal one-to-one mapping (see Theorem 3) and we are able to compute it (see Theorem 1). Thus we are able to assess the absolute performance of the heuristics by computing the optimal period, obtained with a one-to-one mapping (Hungarian algorithm).

C. *Summary*

The results show that H3 and H4 return a mapping whose period is very close to the optimal, which is a very good result. Indeed, we expect this behavior to be similar in a more heterogeneous context, thus assessing the performance of our heuristics. H5 is always returning greater period, thus showing that faster machines must be considered first to find a good mapping (recall that H5 selects machines by heterogeneity level instead of speed, and may use slower machines).

VII. Conclusion

In this paper, we have investigated a throughput optimization problem in the context of micro-factories subject to failures. The problem consists in assigning tasks to machines, either performing a one-to-one mapping (one task per machine), or a specialized mapping (several tasks of the same type per machine), or a general mapping. On the theoretical side, we proved that the optimal one-to-one mapping can be found in polynomial time, while the problem becomes NP-hard for specialized and general mappings. Since general mappings are not usable in practice because of reconfiguration costs, we focused on specialized mappings and proposed several polynomial heuristics to solve the problem. Experimental results suggest that some heuristics return mappings with a throughput close to the optimal, and the sophisticated heuristics return results much better than a random mapping.

As future work, we plan to investigate other mapping rules, as for instance the mapping of one task onto several machines. In such a case, different instances of the task would be handled by different machines. This would allow to obtain a better throughput when a task is time consuming (bottleneck). Also, it would be interesting to consider a failure model in which the failure rate is also machine-dependent (rates $f_{i,u}$ depending on both the task T_i and the machine M_u on which the task is mapped). Finally, other objective functions could be considered, as for instance the total time required to obtain a given number of products, or the average time needed to output one product.

References

[1] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[2] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, and W. Voorsluys, "On the efficacy, efficiency and emergent behavior of task replication in large distributed systems," *Parallel Computing*, vol. 33, no. 3, pp. 213–234, 2007.

[3] B. Parhami, "Voting algorithms," vol. 43, no. 4, Dec 1994, pp. 617–629.

[4] R. West and C. Poellabauer, "Analysis of a window-constrained scheduler for real-time and best-effort packet streams," in *Proc. of the 21st IEEE Real-Time Systems Symp.* IEEE, 2000, pp. 239–248.

[5] R. West, Y. Zhang, K. Schwan, and C. Poellabauer, "Dynamic window-constrained scheduling of real-time streams in media servers," 2004.

[6] R. West and K. Schwan, "Dynamic window-constrained scheduling for multimedia applications," in *ICMCS, Vol. 2*, 1999, pp. 87–91.

[7] P. Jalote, *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001.

[9] H. W. Kuhn, "The hungarian method for the assignment problem," vol. 2, 1955, pp. 83–97.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[11] A. Benoit, A. Dobrila, J.-M. Nicod, and L. Philippe, "Throughput optimization for micro-factories subject to failures," LIP, ENS Lyon, France, Research Report 2009-02, Jan. 2009, available at graal.ens-lyon.fr/~abenoit/.