Optimizing the reliability of pipelined applications under throughput constraints

Anne Benoit, Hinde Lilia Bouziane, Yves Robert

▶ **To cite this version:**

Anne Benoit, Hinde Lilia Bouziane, Yves Robert. Optimizing the reliability of pipelined applications under throughput constraints. 2010. hal-00457207

HAL Id: hal-00457207
https://hal.science/hal-00457207

Preprint submitted on 17 Feb 2010

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.

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Optimizing the reliability of pipelined applications under throughput constraints

Anne Benoit,
Hinde Lilia Bouziane,
Yves Robert

January 2010

Research Report N° 2010-06

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`




Optimizing the reliability of pipelined applications under throughput constraints

Anne Benoit, Hinde Lilia Bouziane, Yves Robert

January 2010

Abstract

Mapping a pipelined application onto a distributed and parallel platform is a challenging problem. The problem becomes even more difficult when multiple optimization criteria are involved, and when the target resources are heterogeneous (processors and communication links) and subject to failures. This report investigates the problem of mapping pipelined applications, consisting of a linear chain of stages executed in a pipeline way, onto such platforms. The objective is to optimize the reliability under a performance constraint, i.e., while guaranteeing a threshold throughput. In order to increase reliability, we replicate the execution of stages on multiple processors. We present complexity results, proving that this bi-criteria optimization problem is NP-hard. We then propose some heuristics, and present extensive experiments evaluating their performance.

Keywords: pipelined applications, interval mapping, throughput, reliability, heterogeneous platforms, bi-criteria optimization, complexity, heuristics.

Résumé

Le problème du placement d'applications sur des infrastructures matérielles distribuées et/ou parallèles représente un grand défi. Ce problème est encore plus difficile lorsque plusieurs critères à optimiser sont à prendre en compte et lorsque ces plates-formes sont hétérogènes (processeurs et liens de communication). Dans ce rapport, nous étudions le problème du placement d'applications pipelinées, composées d'une suite linéaire d'étages exécutés dans un mode pipeline, sur ce type de plates-formes. Notre objectif est d'optimiser la fiabilité sous contrainte de performance, c.-à-d, en assurant un débit minimum à l'exécution. Pour rendre l'application plus fiable, nous utilisons un mécanisme de réplication qui consiste à exécuter un étage de l'application sur plus d'un processeur. Nous présentons les résultats de complexité qui montrent que le présent problème d'optimisation bi-critère est NP-difficile. Nous proposons ensuite un ensemble d'heuristiques et présentons les résultats expérimentaux qui évaluent leurs performances.

Mots-clés: applications pipelinées, placement, débit, fiabilité, plates-formes hétérogènes, optimisation bi-critère, complexité, heuristiques.

1 Introduction

Mapping applications onto distributed and parallel platforms is a difficult problem. The problem becomes even more difficult if we consider state-of-the-art heterogeneous platforms, like clusters or grids. These platforms are typically composed of processors with different speeds and interconnected through networks with different link capacities. In addition, they are subject to failures, which implies adding replication mechanisms to provide a more reliable application execution, and which in turn introduces another level of difficulty to the mapping problem.

A large number of programming models are proposed to design and develop distributed and parallel applications. These models offer means to deal with the complexity of such applications as well as with the complexity of the execution platforms, while attempting to ensure efficient execution and resource usage. In particular, algorithmic skeletons have been introduced to support typical parallel patterns [6, 10]. A skeleton follows a precise structure and execution behavior, like pipelined or farmed computations. It thus provides information that can help the realization of an efficient mapping. In this report, we focus on the widely used *pipeline* skeleton. A pipeline is a linear chain structure in which a stream of data enters a stage and progresses from stage to stage until the final result is computed. Each stage reads an input data produced by the previous stage, processes the data and outputs a result to the next stage. Finally, the pipeline operates in a synchronous mode: after an initialization delay, a new data in the stream is processed every period.

The mapping problem for a pipelined application can informally be stated as to choose which processor is assigned which stage. This choice may be done according to one or multiple optimization criteria. We focus on two key metrics: period (inverse of throughput) and reliability. The period of a mapping is defined as the longest cycle time of a processor. Under a bounded multiport platform model with overlap [8], i.e., in which a processor can simultaneously receive, compute and send data, the cycle time is the maximum among the times spent to perform these operations for all processed data. The reliability of an application is the probability that all computations will be successfully performed. The reliability is increased by replicating each stage on a set of processors. Thus, the application fails to be executed only if processors involved in the execution of a same stage all fail during the whole execution.

This report is a follow-on of a thread of papers aiming at period and/or reliability optimization. Computing a mapping minimizing the period has been studied in [12, 13] onto homogeneous platforms and later in [3] onto heterogeneous platforms. A first attempt to solve the period/reliability problem can be found in [2]. The work in [2] addresses a similar but different problem, that of replicating both for performance (assigning different data sets to different processors, to decrease the period) and for reliability (assigning the same data sets to different processors, to increase the reliability), at the price of a simplified model without any communication cost. In this report, we take communication costs into account, at the price on concentrating on the sole replication for reliability. Indeed, the impact of communications turns out to have dramatic consequences on the difficulty of the mapping problem. This report is the first attempt to deal with the induced combinatorial complexity of deciding for message originators and orchestrating communications in the period/reliability problem.

We follow an interval-mapping approach, where a processor is assigned a single interval of consecutive stages. We consider platforms with different-speed processors having either identical failure probabilities (*failure homogeneous* platforms) or different failure probabilities

(*fully heterogeneous* platforms). The objective is to maximize the reliability under a period constraint onto such platforms.

The rest of the report is organized as follows. Section 2 illustrates the target optimization problem through an example, and further motivates this work. Section 3 formally details the mapping problem, and Section 4 establishes the complexity results. Next in Section 5, we introduce a linear program to solve the mapping problem onto *failure homogeneous* platforms. We also introduce several polynomial-time heuristics in Section 6 to propose practical solutions for the more general case of *fully heterogeneous* platforms. Section 7 is devoted to experimental results; we evaluate the absolute performance of the heuristics with respect to the linear program on small problem instances, before comparing their relative performance on larger instances (whose solution is inaccessible to the linear program). Finally, Section 8 provides some concluding remarks and directions for future work.

2 The mapping problem through an example

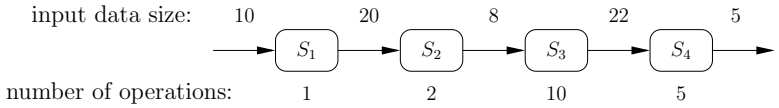

Figure 1: Example of a pipelined application with 4 stages ($S_1..S_4$).

This section aims at outlining the difficulty of mapping a pipelined application on a given execution platform. We consider the application example shown in Figure 1. For each stage of the pipeline, we specify the number of computations, and the size of input/output data.

Figure 2 shows the operations performed for the first six data sets $d_1..d_6$ of the (possibly infinite) input stream. The mapping is assumed to be known: S_1 and S_2 are both assigned to same processors, while S_3 and S_4 are assigned to distinct processors. The notation $S_i \rightarrow S_{i+1}$ corresponds to a remote communication from the processor assigned to S_i to the one assigned to S_{i+1}. For a given data set, the stages are executed in a sequential way, while they are executed in pipelined fashion for different data sets. As soon as the first result res_1 is produced, the pipeline reaches a steady state and periodically produces a result. In the example, provided the threshold 5.5 for the the period of the mapping, i.e., the inverse of the throughput, is 5.5: a new data set enters in the pipeline every 5.5 time units.

The execution platform is shown in Figure 3. In this platform, processors are heterogeneous (different speeds and network card capacities) and are interconnected through homogeneous network links (identical bandwidth $b = 5$ data units per time unit). Such a platform may well represent a heterogeneous cluster. In addition, we assume that processors are subject to unrecoverable failures, where each processor has a probability f to fail during the whole application execution.

To map the application onto the platform, we need to define some rules. First, we assume that a processor can be assigned at most one set of *consecutive* stages. Such a set is named an *interval*. This rule is reasonable, as it allows to better exploit processor capabilities, and it may avoid costly communications. Second, we must handle failures to provide a failure-tolerant execution scheme. For that, we adopt the well known replication principle, which consists in performing redundant executions of some, or all, application intervals on different

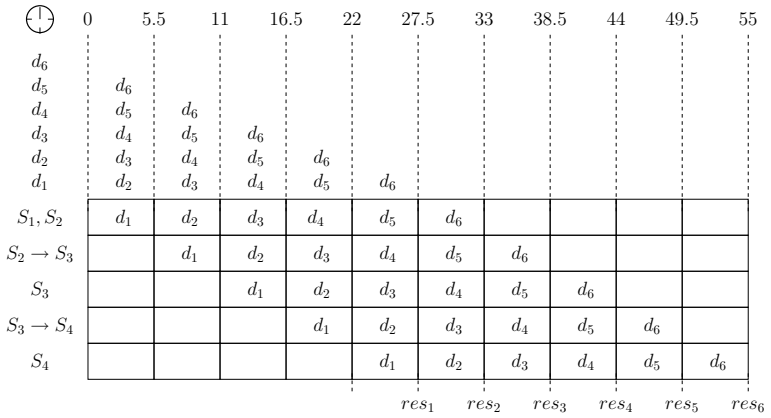

Figure 2: Periodic execution of the application shown in Figure 1; $d_1, \ldots, d_6$ are the first six data sets input to the pipeline, and $res_1, \ldots, res_6$ are the corresponding results.

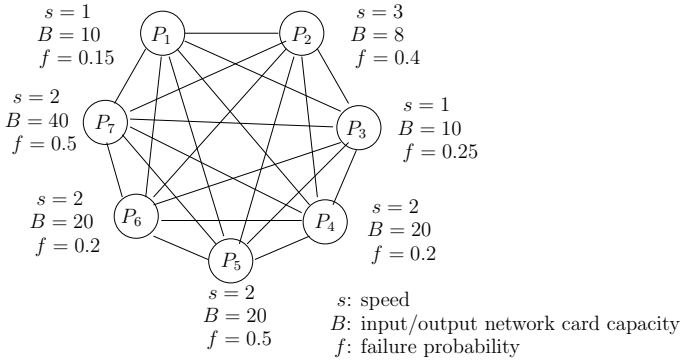

Figure 3: Target execution platform. The links between each processor pair have a bandwidth $b = 5$.

processors. Therefore, a result will be produced (the one of the last stage in the interval) even if a processor fails. In order to avoid producing redundant results for the interval if two or more processors do not fail, we enforce a consensus protocol. This protocol is applied after the process of each data set for a given stage interval. The protocol elects one surviving processor as the sender of the output result of the interval to all processors executing the next interval. This election amounts to choosing the surviving processor allowing for the fastest output communications. We point out that communication links are assumed to be reliable: once a communication is initiated by a processor, it is achieved successfully, and no message is lost.

Considering these rules, mapping stages $S_1, \ldots, S_4$ on $P_1, \ldots, P_7$ raises the following questions: how to partition the stages into intervals and how to partition the processors over the intervals? The objective function consists in maximizing the reliability of the application, given a threshold on the period that should not be exceeded. It is a bi-criteria optimization problem: the goal is to find a mapping which maximizes the reliability with a constraint on the period.

Figure 4 shows an optimal mapping for the example, when the threshold period set to $P_{\max} = 5.5$. In this mapping, stages are partitioned into 3 intervals $[S_1, S_2]$, $[S_3, S_3]$ and

$[S_4, S_4]$. The period of the mapping is defined by the maximum cycle time of all processors $P_1, .., P_7$. This cycle time is deduced from the periodic behavior illustrated by Figure 2, which assumes an overlap of communications and computations. For instance, the cycle time of processor P_1 is computed as follows:

$$CT_1 = \max\left(\max\left(\frac{10}{10}, \frac{10}{5}\right), \frac{1+2}{1}, \max\left(\frac{8*3}{10}, \frac{8}{5}\right)\right) = 3.$$

The first term corresponds to input data (size 10, input network card 10, link bandwidth 5), the second to computations (sum of stage weights $1+2$ divided by speed 1), and the third to output data (size 8, sent 3 times, output network card 10, link bandwidth 5). Similarly, the cycle time of other processors are $CT_2 = 5.5$, $CT_3 = 5$, $CT_4 = 5$, $CT_5 = 5$, $CT_6 = 4.4$ and $CT_7 = 2$. Note that P_2 is the critical resource with the largest cycle time. This cycle time determines the period reachable in the case where processors P_4 and P_5 fail (Figure 5).

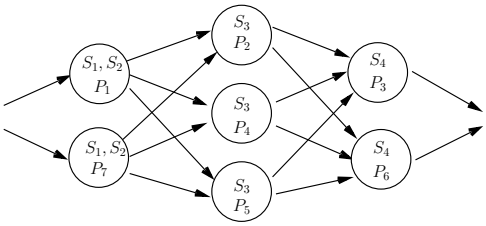

Figure 4: A mapping of the pipelined application of Figure 1 on the platform of Figure 3.

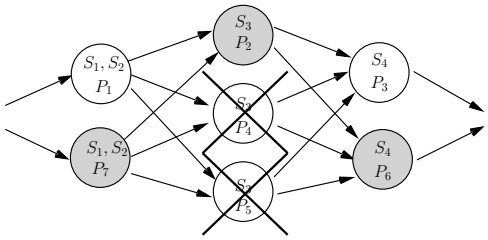

Figure 5: A failure configuration reaching the worst case period. P_4 and P_5 failed. The colored processors are those elected for sending the results.

Finally, the failure probability is computed as 1 minus the probability that the execution is successful, which happens if and only if all intervals are successful (hence a product in the formula); next a given interval fails if and only if all its assigned processors fail. In the mapping example we derive that

$$\mathcal{F} = 1 - (1 - 0.15 * 0.5) * (1 - 0.4 * 0.2 * 0.5) * (1 - 0.25 * 0.2) = 0.1564.$$

This latter probability turns out to be the minimum that can be obtained by all possible mapping solutions of the application example onto the current platform. To prove this, we have enumerated all possible mapping solutions. However, the number of these solutions is exponential. While it may be possible to evaluate all of them for small instances, this is not conceivable for real-life problems. To the best of our knowledge, there is no solution in the literature for this challenging bi-criteria reliability and period optimization problem, on a fully heterogeneous platform with communication costs. The following section presents more formally the applicative framework as well as the mapping problem.

3 Framework

3.1 Applicative framework

This work focuses on pipelined applications. A pipeline is composed of n ordered stages S_i, $1 \leq i \leq n$. These stages continuously operate on a stream of data. When input data are fed

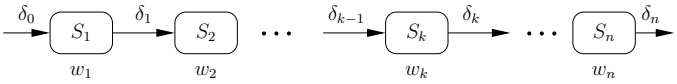

Figure 6: Overview of a pipelined application.

into the pipeline, they are processed from stage to stage, until they exit the last stage S_n. In other words, each stage S_i receives an input data, of size δ_{i-1}, from the previous stage S_{i-1}, performs a computation composed of w_i operations, and produces an output data, of size δ_i. The computation of a stage is periodically repeated on each input data in the pipeline stream. The input (respectively output) of the stream is initially produced (finally consumed) by an extra stage S_0 (respectively S_{n+1}). A graphical representation of a pipelined application is shown in Figure 6.

3.2 Target platform

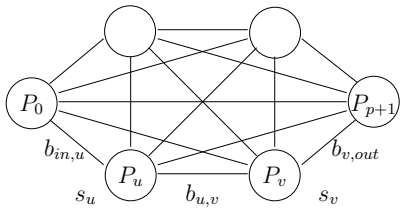

Figure 7: The target platform.

The target platform is composed of $p+2$ processors: p computing processors P_u ($1 \leq u \leq p$) are dedicated to host stages S_i ($1 \leq i \leq n$), while P_0 (also denoted as P_{in}) and P_{p+1} (also denoted as P_{out}) are special processors devoted to host the extra stages S_0 and S_{n+1}. Therefore, P_{in} is dedicated to store initial input data sets of the pipeline and P_{out} to receive and store the final results. Each processor P_u ($1 \leq u \leq p$) has a speed denoted as s_u. That means P_u takes X/s_u time units to execute X operation units. The speed may be identical for all processors ($s_u = s$ for $1 \leq u \leq p$). In this case, the platform is said to be *SpeedHom* (homogeneous in speed). On the opposite, the platform is *SpeedHet* when processors have different speeds.

As shown in Figure 7, all processors are interconnected as a virtual clique. A link between any two processors P_u, P_v ($0 \leq u, v \leq p+1$) is bidirectional and has a bandwidth denoted as $b_{u,v}$. Note that a physical link between any processor pair is not required. Instead, the connection of P_u to P_v may be done through a switch or a path composed of several physical links. In this latter case, $b_{u,v}$ is the bandwidth of the slowest physical link in the path. When the links are identical ($b_{u,v} = b$ for all $0 \leq u, v \leq p+1$), the platform is said to be *LinkHom*. This is the case for instance in parallel machines. Alternatively, the platform is *LinkHet*, like in grid infrastructures.

In addition to link bandwidths, the total communication capacity of a processor is limited by its own input/output network card capacity. Formally, we denote by B_u^i and B_u^o the input and output card capacity of processor P_u. Thus, P_u cannot receive more than B_u^i data

units nor send more than B_u^o data units per time unit. When all processors have same card capacities ($B_u^i = B^i, B_u^o = B^o$, for all $1 \leq u \leq p$), the platform is said to be *CardHom*. This is often true when processors are identical (parallel machines, homogeneous clusters). Otherwise, the platform is said to be *CardHet*.

The platform is assumed to be subject to failures. We consider only fail-silent processor failures without recovering. Thus, a processor can only perform correct actions before eventually crashing (no transient errors). In addition, communication links are assumed to be reliable, hence no data is lost. For the mapping optimization problem, we need to measure the reliability of used processors. This is given by the failure probability f_u ($0 < f_u < 1$) of each processor P_u ($1 \leq u \leq p$). This failure probability is assumed to be constant, i.e., the same at any time during the execution of a pipelined application. This is because we target a steady-state execution, for instance a scenario with resources loaning/renting. In such a scenario, resources could be suddenly reclaimed by their owners, as during an episode of cycle-stealing [1, 5, 11]. Also, there is no time upper bound for the execution of a streaming application which may involve an arbitrary number of data sets, so the failure probability cannot depend upon execution time. As a consequence, the failure probability should thus be seen as a global indicator of the reliability of a processor. We consider platforms with two failure models. The first model, *FailHom*, assumes identical failure probabilities for all processors ($f_u = f$ for all $1 \leq u \leq p$), while the second model, *FailHet*, assumes different failure probabilities.

Finally, we classify a target platform according to different combinations of processors and links properties. In particular, we consider five classes:

- ***Fully Homogeneous* platforms (*FullHom*)**: these platforms are both *SpeedHom*, *CardHom*, *LinkHom* and *FailHom*.
- ***Failure Homogeneous* platforms**: these platforms are *FailHom*, without any homogeneity constraints on processors speeds, network cards and communication links.
- ***Speed Heterogeneous* platforms**: these platforms are *SpeedHet*, but failures, network cards and communication links are homogeneous.
- ***Failure Heterogeneous* platforms**: these platforms are *FailHet*, but speeds, network cards and communication links are homogeneous.
- ***Fully Heterogeneous* platforms (*FullHet*)**: these platforms are *SpeedHet*, *CardHet*, *LinkHet* and *FailHet*.

This classification is relevant from both theoretical and practical perspectives for the mapping optimization problem.

3.3 Communication model

Communications between processors follow the *bounded multi-port model* [8]. In this model, multiple communications can take place simultaneously on a same communication link. This assumes the ability to initiate multiple concurrent incoming and outgoing communications, and to share the link bandwidth. This can be done by using multi-threaded communication libraries such as MPICH2 [9]. The *bounded* characteristic of simultaneous communications is related to the fact that each communication is allotted a bandwidth fraction of the network

card, and the sum of these fractions cannot exceed the total capacity of the card. Moreover, we assume full *overlap* of communications and computations: a stage can simultaneously receive, compute and send independent data. This assumption is reasonable as most state-of-the-art processors are running multi-threaded operating systems capable of such an overlap.

3.4 Mapping problem

Mapping a pipelined application is the process of allocating target execution processors to the pipeline stages. To decide how application stages are assigned to processors, different rules may be adopted. For instance, in *one-to-one mappings*, each stage is assigned to a distinct processor, and thus each processor processes only one single stage. A less restrictive class of mappings, *interval mappings*, are such that a processor may be processing a consecutive subset of stages. In this report, we focus on such interval mappings, which have been widely studied [3, 4, 12, 13].

In the following, we formally define interval mappings and the adopted replication model to deal with processors failures. Then, we express the period and failure probability of a pipelined application, once given a mapping.

Interval mappings: in an interval mapping (with replication), stages S_i $(1 \leq i \leq n)$ are partitioned into $m \leq n$ intervals, and each interval is assigned to a distinct set of processors. This consists in partitioning the interval of stages indices $[1..n]$ into m intervals $I_j = [d_j, e_j]$, where $d_j \leq e_j$ for $1 \leq j \leq m$, $d_1 = 1$, $d_{j+1} = e_j + 1$ for $1 \leq j \leq m-1$ and $e_m = n$. Each interval I_j is mapped to one set of processors whose indices belong to $alloc(d_j)$. In such a mapping, $alloc(i) = alloc(d_j)$ for $d_j \leq i \leq e_j$. In addition, a processor cannot process two distinct intervals, i.e., $alloc(d_j) \cap alloc(d_{j'}) = \emptyset$ for $1 \leq j, j' \leq m$, $j \neq j'$.

Replication model: as discussed in Section 3.2, processors are subject to failures. To deal with such failures, we adopt an *active* replication protocol. In more details, all processors P_u $(u \in alloc(d_j), 1 \leq j \leq m)$ perform the same assigned interval computations (*active*) on the same input data. Therefore, the output data of an interval I_j has to be sent to all processors P_v with $v \in alloc(d_{j+1})$. To avoid redundant input data, a consensus protocol [14] is executed by surviving processors P_u $(u \in alloc(d_j))$ after each execution of interval I_j on an input data. The consensus aims at electing a processor as the sole one that will send the output data of I_j to all surviving processors P_v. This protocol is illustrated in Figure 8, where all processors are surviving. It allows to pay at most $|alloc(d_{j+1})|$ outgoing communications by the elected processor (according to the *bounded multi-port* communication model) and only one incoming communication by P_v. In the scope of this report, we assume that communications intrinsic to the consensus have a negligible overhead. Hence, only the multiple outgoing communications executed by an elected processor are accounted for in the performance model.

Period: the period $\mathcal{P}$ of an interval mapping with replication on the most general *FullHet* platforms is expressed as:

$$\mathcal{P} = \max_{1 \leq j \leq m} \max_{u \in alloc(d_j)} \max \left\{ \frac{\delta_{d_j - 1}}{\min_{v \in alloc(d_j - 1)} (b_{v,u}, B^i_u)}, \frac{\sum_{i=d_j}^{e_j} w_i}{s_u}, \frac{\delta_{e_j}}{\min_{v \in alloc(d_{j+1})} (b_{u,v})}, \frac{|alloc(d_{j+1})| \delta_{e_j}}{B^o_u} \right\}. \quad (1)$$

This formula considers the worst case scenario, where only one processor P_u allocated to interval I_j $(u \in alloc(d_j))$ is surviving, while all processors allocated to the next interval

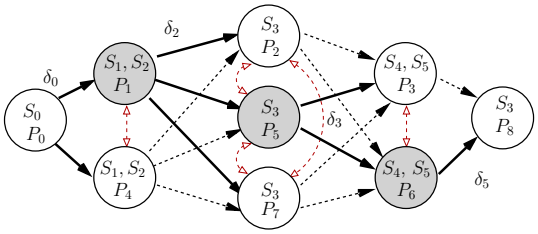

Figure 8: Replication model. Each processor periodically receives input data from one predecessor (on the plain incoming arrow), executes all assigned interval stages, exchanges extra messages (on dashed vertical arrows) with processors allocated to the same interval, agrees upon which processor (filled circle) has to send the result to all its successors (on plain outgoing arrows).

I_{j+1} are alive. The formula for P_u accounts for input data (one communication from the slowest processor assigned to interval I_{j-1}, hence the minimum taken on link and network card bandwidths), for computations, and for output data (constraint on each communication link, on the network card, and there is a total of $|alloc(d_{j+1})|$ communications). There remains to take the maximum of these cycle times for $u \in alloc(d_j)$, and then a global maximum over all intervals.

Failure probability: the failure probability $\mathcal{F}$ of a pipelined application in the most general situation (*FailHet*) is computed by the following formula:

$$\mathcal{F} = 1 - \prod_{1 \leq j \leq m} \left(1 - \prod_{u \in alloc(d_j)} f_u \right). \tag{2}$$

This formula is obtained from the fact that the execution of an application is successful if and only if there remains at least one surviving processor per set $alloc(d_j)$ of processors allocated to each interval of stages, i.e., for $1 \leq j \leq m$.

Finally, the target optimization problem is to determine the best interval mapping that minimizes the failure probability $\mathcal{F}$ given a threshold period $P_{\max}$: the mapping must be such that $\mathcal{P} \leq P_{\max}$.

4 Complexity results

We first prove that the problem can be solved in polynomial time on a fully homogeneous (*FullHom*) platform (*SpeedHom*, *LinkHom*, *CardHom* and *FailHom*), using dynamic programming. However, the problem becomes NP-hard as soon as we add one level of heterogeneity (*SpeedHet* or *FailHet*).

Theorem 1. *For* fully homogeneous *platforms, the optimal interval mapping which minimizes the failure probability under a fixed period threshold can be determined in polynomial time* $O(n^2p^3)$.

Proof. We exhibit here a dynamic programming algorithm which computes the optimal mapping. Let $\mathcal{P}$ denote the threshold period. We recursively compute the value of $R(i, q, q_{succ})$, which is the optimal value of reliability, i.e., $1 - \mathcal{F}$, that can be achieved by any interval

mapping of stages S_1 to S_i, using exactly q processors, and given that the following interval (starting with stage S_{i+1}) is using exactly q_{succ} processors. The goal is to determine $\max_{1\leq q\leq p} R(n,q,1)$, since the final interval has only one succeeding processor, P_{out}. Maximizing the reliability amounts to minimizing the failure probability. The recurrence relation can be expressed as:

$$R(i,q,q_{succ}) = \max_{0\leq j<i,\ 1\leq q'\leq q} \begin{cases} R(j,q-q',q') \times (1-f^{q'}) & \text{if } \begin{cases} \frac{\delta_j}{\min(b,B^i)} \leq \mathcal{P}, \text{ and} \\ \frac{\sum_{k=j+1}^{i} w_k}{s} \leq \mathcal{P}, \text{ and} \\ \max\left(\frac{\delta_i}{b}, \frac{q_{succ}\times\delta_i}{B^o}\right) \leq \mathcal{P}; \end{cases} \\ 0 \quad \text{otherwise,} \end{cases}$$

for $1 \leq i \leq n$, $1 \leq q, q_{succ} \leq p$, with the initialization

$$R(0,q,q_{succ}) = 1 \quad \text{for } 1 \leq q, q_{succ} \leq p,$$

$$R(i,0,q_{succ}) = 0 \quad \text{for } 1 \leq i \leq n, 1 \leq q_{succ} \leq p.$$

The recurrence is easy to justify: to compute $R(i,q,q_{succ})$, we create an interval from stages S_{j+1} to S_i, for $0 \leq j < i$, and we allocate it onto q' processors. Since everything is homogeneous, the contribution to the reliability from this interval is $(1-f^{q'})$, and the reliability is the product of reliabilities of all intervals. The solution is valid only if the constraint on the period is satisfied, hence the condition checking whether input communications, computations and output communications satisfy the bound. The parameter q_{succ} is needed to compute the time required by output communications. In the recursive call, the new value of q_{succ} is thus q', and we consider stages S_1 to S_j.

For the initialization, if we have already mapped all stages, the contribution to the reliability is 1 ($R(0,q,q_{succ})$), while if we used all processors and we still have stages that remain not allocated, then the reliability is set to 0 in order to indicate that this solution is not valid (the failure probability becomes 1).

The complexity of this dynamic programming algorithm is bounded by $O(n^2p^3)$: we need to compute $O(np^2)$ values of $R(i,q,q_{succ})$, and it takes a time $O(np)$ to compute one value in the worst case (maximum over j and q'). □

Theorem 2. *For* SpeedHet *or* FailHet *platforms, finding the optimal interval mapping which minimizes the failure probability under a fixed period threshold is NP-hard, even with no communication cost.*

Proof. We consider the associated decision problem INT-PF: given a period P and a failure probability F, is there a mapping of period less than P and of failure probability less than F?

It is clear that INT-PF is in NP: given a period, a failure probability and a mapping, it is easy to check in polynomial time that it is valid by computing its period and failure probability.

The completeness for *SpeedHet* platforms comes directly from [3], in which we prove that minimizing the period with different speed processors is NP-hard (this is the heterogeneous chains-on-chains problem, denoted CoC-HET). Thus, starting from an instance of CoC-HET, we create an instance of INT-PF with no communication, F set to 1, and the same period as

in CoC-HET. There is no need to replicate in this case: it can only decrease the period, and we do not care about failures since $F = 1$. The two problems are thus fully identical.

To establish the completeness for *FailHet* platforms, we use a reduction from 3-PARTITION [7], which is NP-complete in the strong sense. We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given a set $\{a_1, ..., a_{3m}\}$ of $3m$ integers, and an integer B such that $\sum_{1 \leq j \leq 3m} a_j = mB$, does there exist m independent subsets $B_1, \cdots, B_m$ of $\{1, \cdots, 3m\}$ such that for all $1 \leq i \leq m$, $\sum_{j \in B_i} a_j = B$?

We build the following instance $\mathcal{I}_2$ of our problem: the pipeline is composed of $n = m$ stages with $w = 1$, and $p = 3m$ processors with speeds $s = 1$, and failure probabilities $f_u = 2^{-a_u}$, for $1 \leq u \leq p$. We then set $K = 1$ and $F = 1 - (1 - 2^{-B})^m$.

Note that the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. Indeed, since 3-PARTITION is NP-complete in the strong sense, we could encode $\mathcal{I}_1$ in unary, and thus the size of the instance would be in $O(mB)$. Moreover, the values of f_u and F can be encoded in binary and thus their size is polynomial in the size of $\mathcal{I}_1$.

Now we show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ has a solution. Suppose first that $\mathcal{I}_1$ has a solution. For $1 \leq i \leq m$, stage S_i is mapped onto the processors of subset B_i, thus respecting the period of 1, and this stage is successful with a probability $1 - \prod_{j \in B_i} f_j = 1 - \prod_{j \in B_i} 2^{-a_j} = 1 - 2^{\sum_{j \in B_i} -a_j} = 1 - 2^{-B}$. Since there are m intervals of one stage in the mapping, the total failure probability is $1 - (1 - 2^{-B})^m$, which means that $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. A processor cannot handle more than one stage, otherwise its period becomes greater than $K = 1$ (it would be at least $2w/s = 2$). Let T_i be the set of indices of processors working onto stage S_i, for $1 \leq i \leq m$. The failure probability is thus $1 - \prod_{i=1}^{m} (1 - 2^{-\sum_{j \in T_i} a_j})$. This quantity is always strictly larger than $1 - \prod_{i=1}^{m} (1 - 2^{-B})$ unless when $\sum_{j \in T_i} a_j = B$ for $1 \leq i \leq m$, as was proved by Lemma 2 in [2]. Thus, the processor indices of the mapping correspond to a solution of $\mathcal{I}_1$, which concludes the proof. □

5 A mixed integer linear problem

This section deals with the problem of maximizing reliability under period constraints. As stated in Section 4, this problem is NP-hard for interval mappings on heterogeneous platforms. In this section, we introduce a mixed integer linear program which computes the optimal interval mapping on such platforms, but restricting to *FailHom* processors.

We failed to derive a program with a polynomial number of variables for *FailHet* platforms. The reason can be seen from Equation 2: we would have needed to create a variable for all possible processor subsets S, and to pre-compute the corresponding product $\prod_{u \in S} f_u$. Instead, with *FailHom* platforms, we succeed to keep a polynomial number of variables; the key observation is that we only need to record the number of processors assigned to each interval.

Recall that a pipelined application is composed of n stages and the target platform of p processors, plus two fictitious extra stages S_0 and S_{n+1} respectively assigned to two extra processors P_0 and P_{p+1}. We start by defining the program parameters and variables, then we describe the linear constraints of the problem:

Parameters:

- n: number of application stages, except S_0, S_{n+1}.
- p: number of target platform processors, except P_0, P_{p+1}.
- δ_i ($i \in [0..n]$): the size of output data of stage S_i.
- w_i ($i \in [1..n]$): the workload of stage S_i.
- s_u ($u \in [1..p]$): the speed of processor P_u.
- B_u^i ($u \in [1..p]$): the input network card capacity of processor P_u.
- B_u^o ($u \in [1..p]$): the output network card capacity of processor P_u.
- $b_{u,v}$ ($u, v \in [0..p+1], u \neq v$): the bandwidth of link $P_u \leftrightarrow P_v$.
- Λ_k ($k \in [1..p]$): double equal to $log(1 - f^k)$, where f is the failure probability of all processors (*FailHom* platform).
- $P_{\max}$: the constrained maximum period.

Decision variables:

- $R_{\log}$: the logarithm of the reliability probability to maximize.
- $x_{i,u}$ ($i \in [0..n+1], u \in [0..p+1]$): a boolean variable equal to 1 if stage S_i is assigned to processor P_u. Hypothesis: $x_{0,0} = x_{n+1,p+1} = 1$, $x_{i,0} = 0$ for $i \geq 1$, $x_{0,u} = 0$ for $u \geq 1$, $x_{i,p+1} = 0$ for $i \leq n$ and $x_{n+1,u} = 0$ for $u \leq p$.
- y_i ($i \in [0..n]$): a boolean variable equal to 0 if stages S_i and S_{i+1} belong to a same interval. Hypothesis: $y_0 = y_n = 1$.
- $z_{i,u,v}$ ($i \in [0..n], u, v \in [0..p+1]$): a boolean variable equal to 1 if stage S_i is assigned to P_u and stage S_{i+1} is assigned to P_v. When $u \neq v$, S_i is not assigned to P_v and S_{i+1} is not assigned to P_u (S_i and S_{i+1} are in distinct intervals). Hypothesis: $z_{i,0,v} = 0$ for $i \neq 0$ and all v, and $z_{i,u,p+1} = 0$ for $i \neq n$ and all u.
- $first_u$ and $last_u$ ($u \in [1..p]$): integer variables which denotes, in order, the first and last stages S_{first_u} and S_{last_u} assigned to processor P_u. Thus P_u is assigned the interval $[first_u, last_u]$. Hypothesis: $1 \leq first_u \leq last_u \leq n$.
- nbP_i ($i \in [0..n+1]$): integer variable which denotes the number of processors allocated to stage S_i. Hypothesis: $1 \leq nbP_i \leq p$ and $nbP_0 = nbP_{n+1} = 1$.
- $PperINT_{i,k}$ ($i \in [1..n], k \in [1..p]$): boolean variable equal to 1 if stages S_i and S_{i+1} are assigned to different processors and if S_i is assigned to exactly k processors.

Constraints:

- Stages assignment on processors:
  - If stage S_i is assigned to P_u and not to P_v and stage S_{i+1} is assigned to P_v and not to P_u, then $z_{i,u,v} = 1$:

$$\forall i \in [0..n], \forall u, v \in [0..p+1], u \neq v, \qquad x_{i,u} + x_{i+1,v} + (1 - x_{i,v}) \leq 2 + z_{i,u,v}$$

  - If stages S_i and S_{i+1} are both assigned to P_u, then $z_{i,u,u} = 1$:

$$\forall i \in [0..n], \forall u \in [0..p+1], \qquad x_{i,u} + x_{i+1,u} \leq 1 + z_{i,u,u}$$

  - If $z_{i,u,v} = 1$, then stage S_i is assigned to P_u and stage S_{i+1} to P_v. In addition, when $u \neq v$, S_i is not assigned to P_v nor S_{i+1} to P_u:

$$\begin{aligned}
\forall i \in [0..n], \forall u, v \in [0..p+1], &\qquad z_{i,u,v} \leq x_{i,u} \\
\forall i \in [0..n], \forall u, v \in [0..p+1], &\qquad z_{i,u,v} \leq x_{i+1,v} \\
\forall i \in [0..n], \forall u, v \in [0..p+1], u \neq v, &\qquad z_{i,u,v} \leq 1 - x_{i,v}
\end{aligned}$$

  - If stages S_i and S_{i+1} are both assigned to P_u, then $y_i = 0$:

$$\forall i \in [0..n], \qquad \sum_{u \in [0..p+1]} \sum_{v \in [0..p+1], u \neq v} z_{i,u,v} \geq y_i$$

  - If stages S_i and S_{i+1} are assigned to different processors, then $y_i = 1$:

$$\forall i \in [0..n], \forall u \in [0..p+1], \forall v \in [0..p+1], u \neq v, \qquad z_{i,u,v} \leq y_i$$

  - If stage S_i is assigned to P_u, then $z_{i,u,u}$ is the inverse of y_i:

$$\begin{aligned}
\forall i \in [0..n], \forall u \in [1..p], &\qquad z_{i,u,u} \leq 1 - y_i \\
\forall i \in [0..n], \forall u \in [1..p], &\qquad x_{i,u} - z_{i,u,u} \leq y_i
\end{aligned}$$

- The bounds of an interval:
  - If stage S_i is assigned to P_u, then $first_u \leq i \leq last_u$:

$$\begin{aligned}
\forall i \in [1..n], \forall u \in [1..p], &\qquad first_u \leq i * x_{i,u} + n * (1 - x_{i,u}) \\
\forall i \in [1..n], \forall u \in [1..p], &\qquad last_u \geq i * x_{i,u}
\end{aligned}$$

  - If stage S_i is assigned to P_u and stage S_{i+1} to P_v ($v \neq u$), i.e., $z_{i,u,v} = 1$, then $last_u \leq i$ and $first_v \geq i + 1$ since we consider intervals:

$$\begin{aligned}
\forall i \in [1..n-1], \forall u \in [1..p], \forall v \in [1..p], v \neq u, &\qquad last_u \leq i * z_{i,u,v} + n * (1 - z_{i,u,v}) \\
\forall i \in [1..n-1], \forall u \in [1..p], \forall v \in [1..p], v \neq u, &\qquad first_v \geq (i+1) * z_{i,u,v}
\end{aligned}$$

  - If a processor P_u is not used, then $last_u$ and $first_u$ are forced to be equal to 1:

$$\begin{aligned}
\forall u \in [1..p], &\qquad first_u \leq \sum_{i \in [1..n]} x_{i,u} * n + 1 \\
\forall u \in [1..p], &\qquad last_u \leq \sum_{i \in [1..n]} x_{i,u} * n + 1
\end{aligned}$$

- The number of processors allocated to a stage/interval:
  - Each stage is assigned exactly nbP_i processors:

$$\forall i \in [0..n+1], \sum_{u \in [0..p+1]} x_{i,u} = nbP_i$$

  - If stages S_i and S_{i+1} are both assigned to a same processor, then $nbP_i = nbP_{i+1}$:

$$\forall i \in [0..n], \quad nbP_i - nbP_{i+1} \leq y_i * p$$
$$\forall i \in [0..n], \quad nbP_i - nbP_{i+1} \geq -y_i * p$$

  - If $y_i = 1$, then exactly one value of $k \in [1..p]$ corresponds to the number of processors allocated to S_i, in which case, $PperINT_{i,k} = 1$, otherwise $PperINT_{i,k} = 0$ for all i and all k:

$$\forall i \in [1..n], \quad \sum_{k \in [1..p]} PperINT_{i,k} = y_i$$

  - If $PperINT_{i,k} = 1$, then $k = nbP_i$:

$$\forall i \in [1..n], \quad nbP_i - \sum_{k \in [1..p]} k * PperINT_{i,k} \leq (1 - y_i) * p$$
$$\forall i \in [1..n], \quad nbP_i - \sum_{k \in [1..p]} k * PperINT_{i,k} \geq (1 - y_i)$$

- Cycle-time of a processor:
  - The workload of a processor is expressed as:

$$\forall u \in [1..p], \quad \sum_{i \in [1..n]} \frac{w_i}{s_u} x_{i,u} \leq P_{\max}$$

  - Incoming communications[1] of a processor are expressed as:

$$\forall u \in [1..p], \forall t \in [0..p], t \neq u, \quad \sum_{i \in [1..n]} \frac{\delta_{i-1}}{B^i_u} z_{i-1,t,u} \leq P_{\max}$$
$$\forall u \in [1..p], \forall t \in [0..p], t \neq u, \quad \sum_{i \in [1..n]} \frac{\delta_{i-1}}{b_{t,u}} z_{i-1,t,u} \leq P_{\max}$$

  - Outgoing communications[1] of a processor are expressed as:

$$\forall u \in [1..p], \quad \sum_{i \in [1..n]} \sum_{v \in [1..p+1], v \neq u} \frac{\delta_i}{B^o_u} z_{i,u,v} \leq P_{\max}$$
$$\forall u \in [1..p], \forall v \in [1..p+1], v \neq u, \quad \sum_{i \in [1..n]} \frac{\delta_i}{b_{u,v}} z_{i,u,v} \leq P_{\max}$$

[1] Recall that communications follow the multi-port model with overlap.

- Failure probability: the reliability probability (inverse of failure probability) of the whole pipelined application is expressed as:

$$\sum_{i\in[1..n]} \sum_{k\in[1..p]} \Lambda_k * PperINT_{i,k} \geq R_{\log}$$

Objective function: we aim at finding values for each variable in order to maximize $R_{\log}$, given that all constraints are satisfied.

6 Heuristics

In this section, we propose heuristics for interval mappings with replication on *FullHet* platforms (Section 3.2). The objective is to find a mapping optimizing the failure probability $\mathcal{F}$ under a fixed period bound $P_{\max}$. From Section 4, we know that determining the optimal mapping is NP-hard on heterogeneous platforms. In addition, we know that to satisfy the period bound, it can be necessary to have multiple intervals. Therefore, we propose a large number of stage partitioning techniques to find a good mapping. In particular, we propose two classes of heuristics. In the first class, the partitioning and mapping phases are addressed by different procedures (Section 6.1). In the second class, partitioning and mapping decisions are made on the fly as the heuristics progress (Section 6.2). We intend to explore quite a comprehensive set of mapping solutions, thereby (hopefully) producing a final mapping with satisfyingly small failure probability.

Before presenting the heuristics, we discuss how the allocation of a processor to an interval succeeds during the mapping process.

Processor allocation: during the processor mapping phase, the cycle-time of a candidate processor for assignment to an interval $I_j = [d_j, e_j]$ $(1 \leq j \leq m \leq n)$ is computed to verify the upper fixed period bound $P_{\max}$. According to Formula 1 (computation of the period P of a mapping), this cycle-time depends on the processors allocated to both previous and next intervals (I_{j-1}, I_{j+1}), if they exist. In addition, allocating a processor to an interval changes the cycle-time of these processors and eventually the period of the whole application. However, when attempting an allocation, some intervals may be not yet assigned. Hence, we introduce some rules to be able to return a cycle-time value at each moment during the mapping. These rules are defined through Algorithm 1. This algorithm checks whether a processor may be allocated to a given interval, i.e., preserves $\mathcal{P} \leq P_{\max}$. This verification is essential for the success of the progressive mapping done by the proposed heuristics. In the algorithm, the notation I_0 (respectively I_{m+1}) is used to design the interval composed of the extra stage S_0 (respectively S_{n+1}). The set $alloc(I_j)$ $(1 \leq j \leq m)$ contains the processors allocated to interval I_j before the current application of Algorithm 1.

6.1 Class 1: Heuristics partitioning then mapping

The heuristics presented in this section (Class 1) work in two phases. First, the application is partitioned into a set of intervals. Then we try to map these intervals onto the platform in order to satisfy the bound on the period, and we compute the reliability of the mapping. We try several different partitionings, and keep the solution which returns the most reliable mapping.

Algorithm 1: Potential allocation of processor P_u to interval I_j, checking whether P_u's cycle-time preserves a period $P \leq P_{\max}$ (the period constraint).

begin
  Initialize $B_0^o = B_{p+1}^i = 10^{12}$ for extra processors P_0 and P_{p+1}
  if $|alloc(I_{j-1})| \neq 0$ **then**
    foreach $v \in alloc(I_{j-1})$ **do**
      $sent_v = \frac{|alloc(I_j)|\delta_{j-1}}{B_v^o}$ *// cf. Formula 1*
      if $sent_v > P_{\max}$ **then**
        return *"failure"*
      end
    end
    $comm_u^i = \frac{\delta_{j-1}}{\min_{v \in alloc(I_{j-1})}(b_{v,u}, B_u^i)}$ *// cf. Formula 1*
  end
  else
    $comm_u^i = \frac{\delta_{j-1}}{B_u^i}$
  end
  if $|alloc(I_{j+1})| \neq 0$ **then**
    $comm_u^o = \max(\frac{\delta_j}{\min_{v \in alloc(I_{j+1})} b_{u,v}}, \frac{|alloc(I_{j+1})|\delta_j}{B_u^o})$ *// cf. Formula 1*
  end
  else
    $comm_u^o = \frac{\delta_j}{B_u^o}$ *// There is at least one communication.*
  end
  $load_u = \frac{\sum_{i=d_j}^{e_j} w_i}{s_u}$ *// cf. Formula 1*
  if $\max(comm_u^i, comm_u^o, load_u) > P_{\max}$ **then**
    return *"failure"*
  end
  return *"success"*
end

6.1.1 Partitioning phase

We start with the partitioning phase. Different partitions are returned, by varying the number of target intervals for the partition: the stages are partitioned into k intervals, with $1 \leq k \leq \min(n, p)$. Three criteria are considered for the partitioning phase.

- Communication cost: stages are split at the $(k-1)$ smallest output data sizes (δ_i). Then, longest inter-stage communications are avoided and replaced by local memory accesses on a processor. In the rest of this report, we identify heuristics using this split criteria by a prefix **Partc**.
- Computation cost: stages are split into k intervals such that the computation load of each interval approximates the average $\frac{\sum_{i=1}^{n} w_i}{k}$. Then, costly intervals in terms of computation may be reduced. In the rest of this report, we identify heuristics using this split criteria by a prefix **Partw**.
- Random partitioning: stages are split into k intervals at a random place. In the rest of this report, we identify heuristics using this split criteria by a prefix **Partr**.

These three ways of interval creation seem to be a good trade-off when mapping computation costly, and/or communication costly, stages on various platforms (homogeneous or heterogeneous).

6.1.2 Heuristics for mapping pre-defined intervals

This section presents four main heuristics and derives some variants. These heuristics differ in the way processors are distributed over the pre-defined intervals (computed with one of the previous split criteria), and in the priority to order the assignment of these intervals.

Small: smallest f_u – This greedy heuristic starts by randomly assigning each interval to one processor satisfying the period constraint. Then, it repeatedly assigns the interval with the highest failure probability to the more reliable processor. As soon as a processor is allocated to an interval, it cannot be reused any more. After all processors are considered, the heuristic attempts to improve the global failure probability. For that, it repeatedly merges the interval with the highest failure probability with previous or next intervals (Algorithm 6). The merge process is done as long as the failure probability can be decreased and the period bound is still satisfied. The heuristic is further detailed in Algorithm 2.

Snake: snake allocation of processors – This heuristic assigns each interval to the most reliable processor satisfying the period constraint. At the next step, each interval is assigned to the least reliable processor, and steps are alternated. As soon as a processor is allocated to an interval, it cannot be reused any more. After all intervals/processors are treated, the heuristic attempts to improve the failure probability of the resulting mapping. For that, it performs the same merge step as done by the **Small** heuristic (application of Algorithm 6). The heuristic is further detailed in Algorithm 3. From this heuristic, we can derive some variants, depending upon the order in which intervals are considered for assignment. In the present work, we define two variants **Snake-c** and **Snake-w**. **Snake-c** considers intervals in a decreasing order of their output data size (δ_{e_j}), while **Snake-w** considers intervals in a

Algorithm 2: Heuristic **Small**: greedy mapping of k given intervals to most reliable processors, under a fixed period $P_{\max}$.

begin
  for $j = 1$ **to** k **do**
    Assign interval I_j to a non-used processor randomly selected and satisfying the period $P_{\max}$ (success of Algorithm 1)
    Mark this processor as used
  end
  Order remaining non-used processors P_u by increasing failure probability f_u in list Lp
  foreach $P_u \in Lp$ *in order* **do**
    Allocate P_u to the interval I_j with the highest failure probability and for which P_u satisfies the period $P_{\max}$ (success of Algorithm 1)
    If success, mark P_u as used
  end
  Apply Algorithm 6 (merge) to improve the failure probability of the current mapping
end

decreasing order of their workload ($\sum_{i=d_j}^{e_j} w_i$). Therefore, the mapping priority is given to costly intervals in terms of either their output communications or their workload.

Algorithm 3: Heuristic **Snake**: snake allocation of p processors to k given intervals, under a fixed period $P_{\max}$.

begin
  Order processors $P_u, 1 \leq u \leq p$ by increasing failure probability f_u in list Lp
  Order intervals $I_j = [d_j, e_j]$ $(1 \leq j \leq k)$ by decreasing workload $\sum_{i=d_j}^{e_j} w_i$ in list Li (or decreasing output data size, i.e., δ_{e_j})
  for $i = 1$ **to** $roundUpInt(\frac{p}{k})$ **do**
    foreach $I_j \in Li$ *in order* **do**
      Assign interval I_j to the first processor found in Lp that satisfies the period $P_{\max}$ (success of Algorithm 1)
      Remove this processor from Lp
    end
    Inverse the order of processors in Lp
  end
  Apply Algorithm 6 (merge) to improve failure probability of the resulted mapping
end

BCT: biggest cycle-time – this heuristic repeatedly considers each interval and searches the most critical processor, i.e., with the longest cycle-time satisfying the period $P_{\max}$, and allocates it to this interval. As soon as a processor is allocated to an interval, it cannot be reused any more. After all intervals/processors are treated, the heuristic attempts to improve the failure probability of the computed mapping, with Algorithm 6 similarly to previous heuristics. The **BCT** heuristic is further detailed in Algorithm 4. We can also

derive some variants, depending upon the order in which intervals are treated. As for the **Snake** heuristic, we define two variants **BCT-c** and **BCT-w**. **BCT-c** (respectively **BCT-w**) considers intervals in a decreasing order of their output data size (resp. their workload). We recall that the objective of such variants is to give a priority for mapping costly intervals.

Algorithm 4: Heuristic **BCT**: mapping k given intervals on critical processors, under a fixed period $P_{\max}$.

begin
  Order intervals $I_j = [d_j, e_j]$ $(1 \leq j \leq k)$ by decreasing computation load $\sum_{i=d_j}^{e_j} w_i$ in list Li (or decreasing output data size, i.e. δ_{e_j})
  for $i = 1$ **to** $roundedToUpperInt(\frac{p}{k})$ **do**
    // p is the number of processors.
    foreach $I_j \in Li$ *in order* **do**
      Assign I_j to the non-used processor resulting to biggest cycle-time and satisfying the period $P_{\max}$ (success of Algorithm 1)
      Mark this processor as used
    end
  end
  Apply Algorithm 6 (merge) to improve failure probability of the resulted mapping
end

Bal: balancing failure probabilities – This heuristic assigns each interval I_j to a set of most critical processors, i.e., with the longest cycle-time satisfying the period $P_{\max}$. This set ($alloc(I_j)$) is such that the product of the processors failure probabilities $\prod_{u \in alloc(I_j)} f_u$ approximates the average value $\sqrt[k]{\prod_{u \in [1..p]} f_u}$ (p is the number of all processors). As soon as a processor is allocated to an interval, it can not be reused any more. When all intervals are assigned, the heuristic attempts to improve the failure probability of the computed mapping by applying the intervals merging algorithm (Algorithm 6). The heuristic is further detailed in Algorithm 5. As for **Snake** and **BCT** heuristics, we define two variants of the **Bal** one: **Bal-c** and **Bal-w**. **Bal-c** (respectively **Bal-w**) treats the intervals in a decreasing order of their output data size (resp. their workload). The objective is still the same, i.e., to provide a mapping priority for costly intervals.

6.1.3 Partitioning-then-mapping heuristics

Given a partitioning criteria, we map the intervals of a partition using one of the mapping heuristics. With three different partitioning criteria, four mapping strategies and three variants of these strategies (thus a total of seven mapping strategies), we obtain the following 21 heuristics:

- **Partc-Small**, **Partr-Small**, **Partw-Small**,
- **Partc-Snake-c**, **Partr-Snake-c**, **Partw-Snake-c**, **Partc-Snake-w**, **Partr-Snake-w**, **Partw-Snake-w**,
- **Partc-BCT-c**, **Partr-BCT-c**, **Partw-BCT-c**, **Partc-BCT-w**, **Partr-BCT-w**, **Partw-BCT-w**,

Algorithm 5: Heuristic **Bal**: mapping k given intervals with balancing their failure probabilities, under a fixed period $P_{\max}$.

begin
Order intervals $I_j = [d_j, e_j]$ $(1 \leq j \leq k)$ by decreasing computation load $\sum_{i=d_j}^{e_j} w_i$ in list Li (or decreasing output data size, i.e., δ_{e_j})
foreach $I_j \in Li$ *in order* **do**
Assign I_j to a set of non-used processors *procs* of $P_u (1 \leq u \leq p)$ with $\prod_{u \in alloc(I_j)} f_u \approx (\sqrt[k]{\prod_{u \in [1..p]} f_u})$ and which result to the largest cycle-times satisfying the period $P_{\max}$ (success of Algorithm 1)
Mark each processor in *procs* as used
end
Order remaining non-used processors P_u by increasing failure probability f_u in list Lp
foreach $P_u \in Lp$ *in order* **do**
Allocate P_u to the interval I_j with the highest failure probability and for which the period P is satisfied (success of Algorithm 1)
end
Apply Algorithm 6 (merge) to improve failure probability of the resulted mapping
end

- **Partc-Bal-c**, **Partr-Bal-c**, **Partw-Bal-c**, **Partc-Bal-w**, **Partr-Bal-w**, **Partw-Bal-w**.

Algorithm 7 details the **Partc-Small** heuristic, and the others are working in a similar way, with different variants.

6.2 Class 2: Heuristics with progressive creation of intervals

In opposition to the previous heuristics, the heuristics of class 2 compute a mapping solution based on interleaved (and progressive) interval creations and processor allocations. Their principle is common. It consists in splitting stages within an interval into several intervals only if it is necessary, i.e., if no processor can be allocated without transgressing the period constraint. The objective is to minimize the number of intervals, so as to reach the smallest failure probability of a mapping.

In more details, each heuristic in the present class repeatedly attempts to assign an interval I (initially composed of all stages $S_1, .., S_n$) to an initial number q of processors satisfying the period $P_{\max}$. If no processors are found, the heuristic splits interval I into two new intervals, and try recursively to perform such an assignment. The process is repeated until processors are found for the interval, or no further split is possible. Because the final number of intervals is not known, q may be chosen between 1 and p. To increase the probability to find processors to be allocated at each step, we have chosen q between 1 and $\frac{p}{2}$. At the end, the heuristic attempts to assign the remaining non used processors (if any). It also tries to improve the failure probability of the resulted mapping. For that, it repeatedly merges the interval with the highest failure probability with previous or next intervals (Algorithm 6). The merge process is done as long as the failure probability can be decreased and the period bound is still satisfied. Finally, the heuristic explores solutions for several values of q (as

Algorithm 6: Merging intervals of a given mapping with initially k intervals to decrease failure probability $\mathcal{F}$, under a fixed period $P_{\max}$.

begin
  while *it is possible to decrease* $\mathcal{F}$ *and there are at least* 2 *intervals* **do**
    Find interval I_j $(1 \leq j \leq k)$ in the current mapping with the highest failure probability
    // Step 1
    Merge I_j with I_{j+1} $(I_{j+1} \neq [n+1, n+1])$
    Discard processors among those initially assigned to I_j and I_{j+1} and non-used ones that do not satisfy the period after merge. Discarded processors become non-used
    // Step 2
    Merge I_j with I_{j-1} $(I_{j-1} \neq [0, 0])$
    Discard processors among those initially assigned to I_j and I_{j-1} and non-used ones that do not satisfy the period after merge. Discarded processors become non-used
    // Step 3
    if *Step 1 or Step 2 decreases the failure probability of the initial mapping* **then**
      Retain the mapping with the smallest failure probability
    end
    else
      Ignore the merge done in Step 1 and Step 2
    end
  end
  Order remaining non-used processors P_u by increasing failure probability f_u in list Lp
  foreach $P_u \in Lp$ *in order* **do**
    Allocate P_u to the interval I_j with the biggest failure probability and for which P_u satisfies the period $P_{\max}$ (success of Algorithm 1)
  end
end

Algorithm 7: Heuristic **Partc-Small** computing an interval mapping optimizing $\mathcal{F}$, under a fixed period $P_{\max}$.

begin
  Initialize the failure probability of the application $\mathcal{F}$ to 1
  for $k = 1$ **to** $\min(n, p)$ **do**
    // Step 1: create intervals according to communication cost criterion.
    Split the interval $[1..n]$ of all stages $S_1, .., S_n$ into k intervals at stages S_j $(1 \leq j \leq n)$ with the k smallest δ_j, Now S_j and S_{j+1} (if it exists) belong to new different intervals
    // Step 2: compute a mapping for the created intervals.
    Apply Algorithm 2 and compute the failure probability $\mathcal{F}_t$ of the resulted mapping (if an interval is not assigned, set $\mathcal{F}_t$ to 1)
    Accept the mapping with $\min(\mathcal{F}_t, \mathcal{F})$ and set $\mathcal{F}$ to this value
  end
  // Step 3: return a mapping solution.
  if $\mathcal{F}_t = 1$ **then**
    return *"failure"* // one (at least) interval was not assigned.
  end
  Return a mapping solution among the $\min(n, p)$ computed ones with the final failure probability $\mathcal{F}$
  return *"success"*
end

explained above, q varies from 1 to $\frac{p}{2}$) and it retains the solution with the smallest failure probability $\mathcal{F}$.

The way the split is done, as well as the mapping order of the step-by-step computed intervals, determine different heuristic variants in the present class. First, the split can be done at several places within an interval. In this work, we propose a recursive split of an interval into two intervals according to two criteria:

- Communication cost, where the split is done at the stage with the smallest output data size. The objective is to reduce costly communications. In the rest of this report, we identify heuristics using this split criteria by a prefix **Splitc**.

- Random partitioning, where the split is done at a randomly computed place. The objective is to have an intermediate solution between costly communications and costly computations. In the rest of this report, we identify heuristics using this split criteria by a prefix **Splitr**.

Secondly, we determine a mapping order of split intervals. In the present work, we defined two orders. The first order gives a mapping priority to the interval with the biggest output data size. Heuristics using this order are denoted with a suffix **c**. The second order gives the priority to the interval with the biggest workload ($\sum_{i=d_j}^{e_j} w_i$). Heuristics using this order are denoted with a suffix **w**. Therefore, the mapping priority is given to costly intervals in terms of either their output communications or their workload. Finally, we define the following four heuristics (including their variants): **Splitc-c**, **Splitc-w**, **Splitr-c** and **Splitr-w**.

Algorithm 8: Heuristic **Splitc-c** computing an interval-mapping optimizing $\mathcal{F}$ under a fixed period $P_{\max}$.

begin
  Initialize interval $Int = [1, n]$ of all stages $S_1, .., S_n$
  for $q = 1$ **to** $\frac{p}{2}$ **do**
    Apply Algorithm 9 on interval Int with q as the maximum number of processors to allocate to each final interval
    Order remaining non-used processors P_u by increasing failure probability f_u in list Lp
    foreach $P_u \in Lp$ *in order* **do**
      Allocate P_u to the interval I_j with the biggest failure probability and for which the period $P_{\max}$ is satisfied (success of Algorithm 1)
    end
    Apply Algorithm 6 (merge) to improve failure probability of the resulted mapping
  end
  Choose a valid solution (success of Algorithm 9) among q previous ones with the smallest failure probability
  if *non valid solution exists* **then**
    return *"failure"*
  end
  return *"success"*
end

Heuristic **Splitc-c** is detailed in Algorithms 8 and 9. We recall that the heuristic explores solutions for different numbers of processors to be allocated per interval, and retains the solution with the smallest failure probability $\mathcal{F}$.

7 Experiments

This section discusses the performance of the heuristics proposed in Section 6 for different problems sizes. We have simulated several mapping scenarios for randomly generated applications with n from 2 to 120 stages and randomly generated platforms with p from 6 to 100 processors. For all these experiments, the computation load (w) of stages is a random double chosen in the interval $[1, 20]$ and the output data size (δ) or d) is a random integer chosen in the interval $[1, 25]$. As for computing resources, we recall that the heuristics have been designed for *FullHet* platforms. Experiments are done for such platforms, as well as for the restricted case of *Failure Homogeneous* ones. For each processor, the speed (w) is a random double chosen in the interval $[1, 20]$, and the input/output network card capacity is a random double chosen in the interval $[1, 10]$, like the bandwidth b of communication links. At last, the failure probability (f) of processors are either homogeneous, equal to 0.1, or a random value chosen between 0.05 and 0.3.

Heuristics have been developed using the C/C++ language and gcc compiler version 4.3.2. Experiments have been conducted on two machines: one quad-processor machine (64-bit AMD Opteron at 2.3GHz) with 32 GB of RAM and one quad-processor machine (64-bit AMD Opteron at 2.4GHz) with 80 GB of RAM. The whole source code of the heuristics and

Algorithm 9: Heuristic **Splitc-c**-bis for recursive interval-mapping of an interval Int on at most q processors per resulted intervals under a fixed period $P_{\max}$.

begin
  if *there is no more non-used processors* **then**
    return *"failure"*
  end
  Assign the input interval $Int = [d, e]$ to a set of maximum q non-used processors *procs* that result to biggest cycle-times satisfying the period $P_{\max}$ (success of Algorithm 1)
  Mark each processor in *procs* as used
  if $procs \neq \emptyset$ **then**
    return *"success"*
  end
  if $d = e$ **then**
    return *"failure"*
  end
  Split Int into intervals $Int_1 = [d, e_1]$ and $Int_2 = [d_2, e]$ at stage S_i $(i = e1)$, such as S_i has the smallest output data δ_i compared to stages $S_d, ..S_{e-1}$
  Apply the present Algorithm 9 to the interval among Int_1, Int_2 with the biggest output data size
  if *this application fails* **then**
    return *"failure"*
  end
  Apply the present Algorithm 9 to remaining interval
  if *this application fails* **then**
    return *"failure"*
  end
  return *"success"*
end

	Failure probability					Execution time
	min	max	av.	stdv.	best rate	av. (msec)
Partc-Small	0.000	0.729	0.011	0.074	85.29%	11.69
Partr-Small	0.000	0.969	0.033	0.145	76.71%	17.97
Partw-Small	0.000	0.969	0.033	0.139	75.14%	17.12
Partc-Snake-c	0.000	0.729	0.007	0.050	84.14%	3.36
Partc-Snake-w	0.000	0.729	0.007	0.050	85.43%	2.63
Partr-Snake-c	0.000	0.900	0.030	0.132	75.14%	4.66
Partr-Snake-w	0.000	0.900	0.020	0.100	76.14%	6.07
Partw-Snake-c	0.000	0.969	0.032	0.137	73.14%	2.56
Partw-Snake-w	0.000	0.900	0.030	0.130	73.29%	9.42
Partc-BCT-c	0.000	0.891	0.010	0.066	83.86%	4.42
Partc-BCT-w	0.000	0.802	0.011	0.065	83.43%	5.02
Partr-BCT-c	0.000	0.900	0.025	0.116	74.43%	3.12
Partr-BCT-w	0.000	0.890	0.024	0.112	75.71%	5.54
Partw-BCT-c	0.000	0.969	0.030	0.130	71.86%	8.66
Partw-BCT-w	0.000	0.969	0.031	0.133	73.14%	5.88
Partc-Bal-c	0.000	0.900	0.016	0.096	82.43%	8.64
Partc-Bal-w	0.000	0.810	0.015	0.090	83.14%	7.29
Partr-Bal-c	0.000	0.891	0.027	0.123	73.71%	6.57
Partr-Bal-w	0.000	0.890	0.022	0.107	75.29%	3.06
Partw-Bal-c	0.000	0.900	0.034	0.136	70.00%	8.29
Partw-Bal-w	0.000	0.900	0.030	0.128	71.14%	7.33
Splitc-c	0.000	0.891	0.027	0.085	65.14%	3.95
Splitc-w	0.000	0.900	0.034	0.110	64.29%	4.60
Splitr-c	0.000	0.900	0.034	0.110	63.86%	1.63
Splitr-w	0.000	0.890	0.033	0.106	64.29%	2.73
Linear-P	0.000	0.000	0.000	0.000	100.00%	2.78e+05

Table 1: Heuristics vs linear program results on a small *Failure Homogeneous* platform. Failure probabilities: minimum, maximum, average and standard absolute deviation for 700 mapping results. Execution times: average over the 700 executions.

the experiments setup can be found on the Web at: http://graal.ens-lyon.fr/~hbouzian/code/heuristics-FT-P.tgz.

In the following we first evaluate the performance of the heuristics compared to the optimal solution returned by the linear program presented in Section 5. Next, we focus on comparing the heuristics for large problem instances.

7.1 Absolute performance of heuristics

This section compares the results obtained by the heuristics to the optimal mapping solution returned by the linear program. This linear program is solved using the CPLEX Interactive Optimizer version 11.2.0. This version has a support for mixed integer linear programs like in the present case.

As the linear program has been designed for *FailHom* platforms (see Section 5), we limit the comparison with heuristics on such platforms. In addition, the large number of variables in the linear program forces us to limit the experiments to small applications and platforms. We have chosen scenarios with 8 stages and 10 processors. In particular, we selected 14 period bounds between 1.5 and 8.0. For each bound, 50 instances of application-platform pairs have been generated.

Table 1 reports the absolute deviation of the failure probabilities resulting from the heuristics compared to the optimal results. Several conclusions can be drawn. First, from the average (av.) and standard deviation (stdv.) columns, the heuristics based on the communication criteria to partition the stages into intervals (**Partc**-*) approach better the optimal

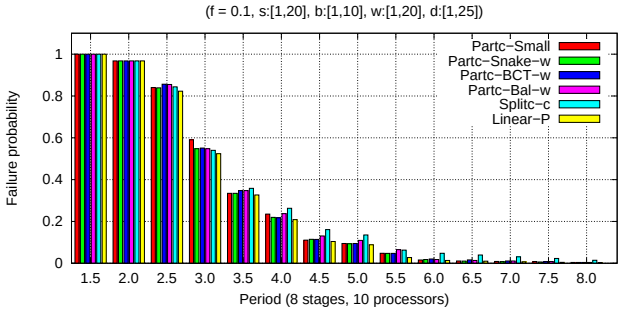

Figure 9: Best heuristics variants vs the linear program on small *Failure Homogeneous* platforms.

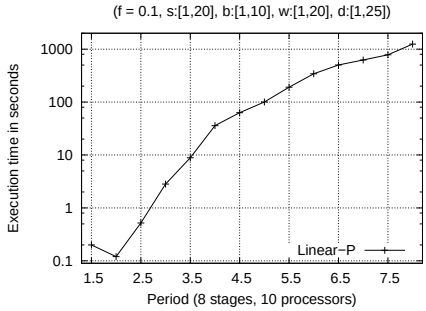

Figure 10: Execution times of the linear program.

solution. This is explained by the fact that small messages between intervals can increase the replication degree of each intervals and then the reliability of the mapping. Second, the best results are obtained by heuristics that repeatedly allocate processors one by one to all intervals. Indeed, the opposite behavior of the variants **Partc-Bal**-{c|w} and those of *class 2* ({**Splitc**|**Splitr**}-{**c**|**w**}) can more frequently limit the possibility to allocate processors. However, experiments presented later report that these heuristics can behave better for larger platforms. Therefore, the current experiment is not sufficient to determine the best heuristics in a general case. Third, from the success rates of heuristics to return the optimal mapping, we can conclude that most of the heuristics give satisfying results. At last, to confirm the above conclusions, Figure 9 shows the evolution of the average behavior of selected heuristics depending on the period bound. It can be noted that for period bounds higher than 2.5, the behavior compared to the linear program varies. The main reason is that stage partitioning and processor distribution are done according to partially known data (communication and/or computation costs). Therefore, it is critical to identify one best heuristic for all period bounds.

Table 1 also reports the execution time of the heuristics and of the linear program. We see that the heuristics are very fast. The biggest average, ≈ 18 msec, is obtained when using the **Small** heuristic variants. This time is explained by the multiple attempts performed to randomly find a processor to be allocated to an interval. From the table, we also see that the linear program requires an average of some minutes (≈ 5) to find the optimal solution. To explain this time, Figure 10 shows its evolution depending on the period bound. We can see that the average time considerably increases with the period bound. Indeed, when this bound is large, the solution space is wider, thus more operations are performed. The maximum resulted time is more than 3 hours, reached for a period bound set to 8. This time is estimated to be very long for a small platform. This represents the main limitation of the linear program. For this reason, we limited experiments with the linear program to small problem instances.

7.2 Comparisons between heuristics

This section presents the results obtained for six sets of experiments. These sets correspond to scenarios with different platform and application sizes. We discuss the impact of varying these parameters on the performance of the heuristics. For each scenario, we also discuss the

impact of the heterogeneity degree of processor failure probabilities. In this direction, each experiment set has been executed on both *FullHet* and *FailHom* platforms. All experiments have been executed on a same machine. In the following figures, each reported experimental value is an average of mapping failure probabilities over 200 application-platform pairs.

Figures 11, 12, 13, 14 and 15 compare the results obtained for the variants of each mapping heuristic presented in Section 6. There are six pairs of plots in each Figure. Each pair of plots corresponds to a set of experiments conduced on *FailHom* (on the left) and *FullHet* platforms (on the right). The first four pairs evaluate the performance of heuristics for fixed sizes of application-platform pairs and different period bounds, while the last two pairs evaluate the performance for fixed sizes of platforms (number of processors) and different applications sizes (number of stages) under a fixed period.

From each figure, we first observe that the behavior of heuristic variants is almost similar for both *FailHom* and *FullHet* platforms. This is explained by the fact that even if the failure probability of a processor is a relevant parameter considered by the mapping process, the respect of the period constraint has a more relevant impact on the result. Second, we observe that the average gap between the results of each heuristic variant becomes more important for medium and large platforms ($p \geq 50$). That is mainly explained by the fact that when the number of processors increases, the order in which processors are allocated varies more, and the probability to allocate a same processor to a same interval decreases. Third, for the same platforms, we observe that heuristics **Partc**-* and **Splitc**-* globally reach better performance. These heuristics partition the stages (prefix **Partc**) or split the intervals (prefix **Splitc**) according to the communication criterion. In this case, the communication of large amount of data is avoided when possible, by mapping the two consecutive stages onto a same processor. For smaller communications, it is then possible to send more copies of the output data to the processor in charge of the next interval, while not exceeding the bound on the period. Therefore, it becomes easier to increase the replication degree of an interval, hence to reach a better reliability. Last, it is interesting to note that the random partitioning done by heuristics of class 1 (with a prefix **Partr**) can behave better than the heuristics that partition stages according to computation costs criterion (with a prefix **Partw**). This often appears when the period bound is not too small (failure probability close to 1) nor too large (failure probability close to 0). This is explained by the fact that an important variation of stage workloads can easily lead to costly intervals when partitioning according to computation costs. Thus, the probability to find a processor matching the period bound may be reduced.

7.3 Summary

Table 2 sums up the performance of all heuristics over all experiments. For each heuristic, we have represented its absolute failure probability, and compared it to that of the heuristic reaching the best (smallest) probability for each experimental value. We have chosen this representation because it defines a meaningful lower bound for comparison. For small platforms, we observe that heuristic **Partc-Small** surpasses all the other heuristics in terms of success rate to give the smallest failure probability. However, from the average and standard deviation columns, other heuristics, like **Splitc-c** with a poor success rate, achieve better performance. The results are different for larger platforms. No large success rate is observed for one particular heuristic, but the rate is dispersed over multiple heuristics. However, we can deduce that heuristics partitioning stages or splitting intervals according to communication costs reach better performance. Among these heuristics, it is no obvious to distinguish a best

	Small platforms (22800 results)					Large platforms (14800 results)				
	min	max	av.	stdv.	best rate	min	max	av.	stdv.	best rate
Partc-Small	0.000	0.969	0.076	0.184	59.79%	0.000	0.958	0.065	0.100	14.77%
Partr-Small	0.000	0.990	0.093	0.191	4.84%	0.000	0.862	0.114	0.123	0.75%
Partw-Small	0.000	0.988	0.093	0.194	3.20%	0.000	0.995	0.215	0.271	0.71%
Partc-Snake-c	0.000	0.961	0.072	0.177	4.86%	0.000	0.958	0.064	0.100	4.97%
Partc-Snake-w	0.000	0.951	0.057	0.154	3.97%	0.000	0.958	0.045	0.087	10.45%
Partr-Snake-c	0.000	0.990	0.089	0.185	1.35%	0.000	0.871	0.120	0.125	0.46%
Partr-Snake-w	0.000	0.990	0.071	0.159	1.75%	0.000	0.886	0.099	0.112	0.88%
Partw-Snake-c	0.000	0.989	0.072	0.162	1.45%	0.000	0.997	0.225	0.266	0.47%
Partw-Snake-w	0.000	0.988	0.064	0.151	1.41%	0.000	0.995	0.209	0.262	0.50%
Partc-BCT-c	0.000	0.951	0.061	0.159	2.34%	0.000	0.958	0.059	0.095	5.47%
Partc-BCT-w	0.000	0.951	0.056	0.153	2.10%	0.000	0.958	0.052	0.090	8.38%
Partr-BCT-c	0.000	0.990	0.081	0.174	1.19%	0.000	0.886	0.113	0.122	0.61%
Partr-BCT-w	0.000	0.900	0.069	0.155	1.17%	0.000	0.862	0.102	0.111	0.52%
Partw-BCT-c	0.000	0.988	0.067	0.154	0.96%	0.000	0.997	0.225	0.265	0.43%
Partw-BCT-w	0.000	0.988	0.061	0.146	0.88%	0.000	0.997	0.221	0.261	0.42%
Partc-Bal-c	0.000	0.951	0.062	0.160	1.04%	0.000	1.000	0.079	0.127	9.93%
Partc-Bal-w	0.000	0.951	0.058	0.154	1.44%	0.000	0.958	0.075	0.118	7.34%
Partr-Bal-c	0.000	0.988	0.082	0.173	0.76%	0.000	1.000	0.129	0.152	1.18%
Partr-Bal-w	0.000	0.988	0.070	0.156	0.92%	0.000	1.000	0.118	0.134	0.75%
Partw-Bal-c	0.000	0.988	0.068	0.154	2.04%	0.000	1.000	0.248	0.289	1.00%
Partw-Bal-w	0.000	0.990	0.062	0.146	0.44%	0.000	1.000	0.240	0.285	0.56%
Splitc-c	0.000	0.931	0.050	0.102	2.34%	0.000	0.777	0.074	0.088	11.48%
Splitc-w	0.000	0.909	0.052	0.106	1.43%	0.000	0.882	0.053	0.076	17.64%
Splitr-c	0.000	0.985	0.076	0.143	0.95%	0.000	0.871	0.181	0.155	0.27%
Splitr-w	0.000	0.990	0.074	0.139	1.23%	0.000	0.871	0.177	0.151	0.28%

Table 2: Heuristics comparison over all done experiments (minimum, maximum, average and standard absolute deviation of failure probabilities for computed mappings).

one. Nevertheless, we observe satisfying results: with not-too-constrained periods, we reach failures probabilities less than 0.2.

8 Conclusion

We have studied the complexity of the mapping problem onto heterogeneous platforms subject to failures. We focused on pipelined applications, composed of consecutive stages executed in a pipeline way. The objective is to find interval mapping solutions for such applications, while maximizing the reliability under a performance (throughput) constraint. A major difficulty is to deal with the impact of communication overheads. To the best of our knowledge, there are no previous results for this important bi-criteria problem, despite the fact that such pipeline workflows are widely encountered in real-life problems.

Our first contribution was to present new complexity results, providing a polynomial algorithm to solve the problem in a fully homogeneous setting, and proving the NP-hardness of the problem when adding one degree of heterogeneity. We also proposed a mixed integer linear programming formulation, which allows us to compute (in exponential time) the optimal solution on *FailHom* platforms. Even for such platforms, the program can take very long time to execute, even for small application/platform pairs, and we could not derive a formulation with a polynomial number of variables for *FullHet* platforms. At last, we have developed polynomial-time heuristics for *fully heterogeneous* platforms. Experimental results showed that for small *FailHom* platforms, the heuristics reach results close to the optimal solution provided by the linear program. Finally, we pointed out that for different problems sizes, multiple heuristics reach quite good results, and it is difficult to identify a particular one with

"the best behavior" in all situations.

We are currently investigating an extension of the results to *general* mappings, where a processor may be assigned multiple intervals of stages. In this context, even computing the worst-case period of a *given* mapping becomes difficult, because it depends upon an exponential number of possible failure configurations. Therefore, much more work is needed before tackling the corresponding optimization problem: in a nutshell, before finding the best mapping, we have to agree on a polynomial approximation of the worst-case period of a given mapping! However, on the practical side, many of the heuristics presented in this report could be extended to this new problem.

References

[1] B. Awerbuch, Y. Azar, A. Fiat, and F. Leighton. Making commitments in the face of uncertainty: how to pick a winner almost every time. In *28th ACM Symp. on Theory of Computing*, pages 519–530. ACM Press, 1996.

[2] A. Benoit, L. Marchal, Y. Robert, and O. Sinnen. Mapping pipelined applications with replication to increase throughput and reliability. Research Report 2009-28, LIP, ENS Lyon, France, Oct. 2009. Available at graal.ens-lyon.fr/~abenoit.

[3] A. Benoit and Y. Robert. Mapping pipeline skeletons onto heterogeneous platforms. *J. Parallel and Distributed Computing*, 68(6):790–808, 2008.

[4] A. Benoit and Y. Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica*, 2009. Available online at http://dx.doi.org/10.1007/s00453-008-9229-4.

[5] S. Bhatt, F. Chung, F. Leighton, and A. Rosenberg. On optimal strategies for cycle-stealing in networks of workstations. *IEEE Trans. Computers*, 46(5):545–557, 1997.

[6] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

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

[8] B. Hong and V. Prasanna. Bandwidth-aware resource allocation for heterogeneous computing systems to maximize throughput. In *Proceedings of the 32th International Conference on Parallel Processing (ICPP'2003)*. IEEE Computer Society Press, 2003.

[9] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *J. Parallel and Distributed Computing*, 63(5):551–563, 2003.

[10] F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, 2002.

[11] A. Rosenberg. Optimal schedules for cycle-stealing in a network of workstations with a bag-of-tasks workload. *IEEE Trans. Parallel and Distributed Systems*, 13(2):179–191, 2002.

[12] J. Subhlok and G. Vondran. Optimal mapping of sequences of data parallel tasks. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[13] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[14] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2000.

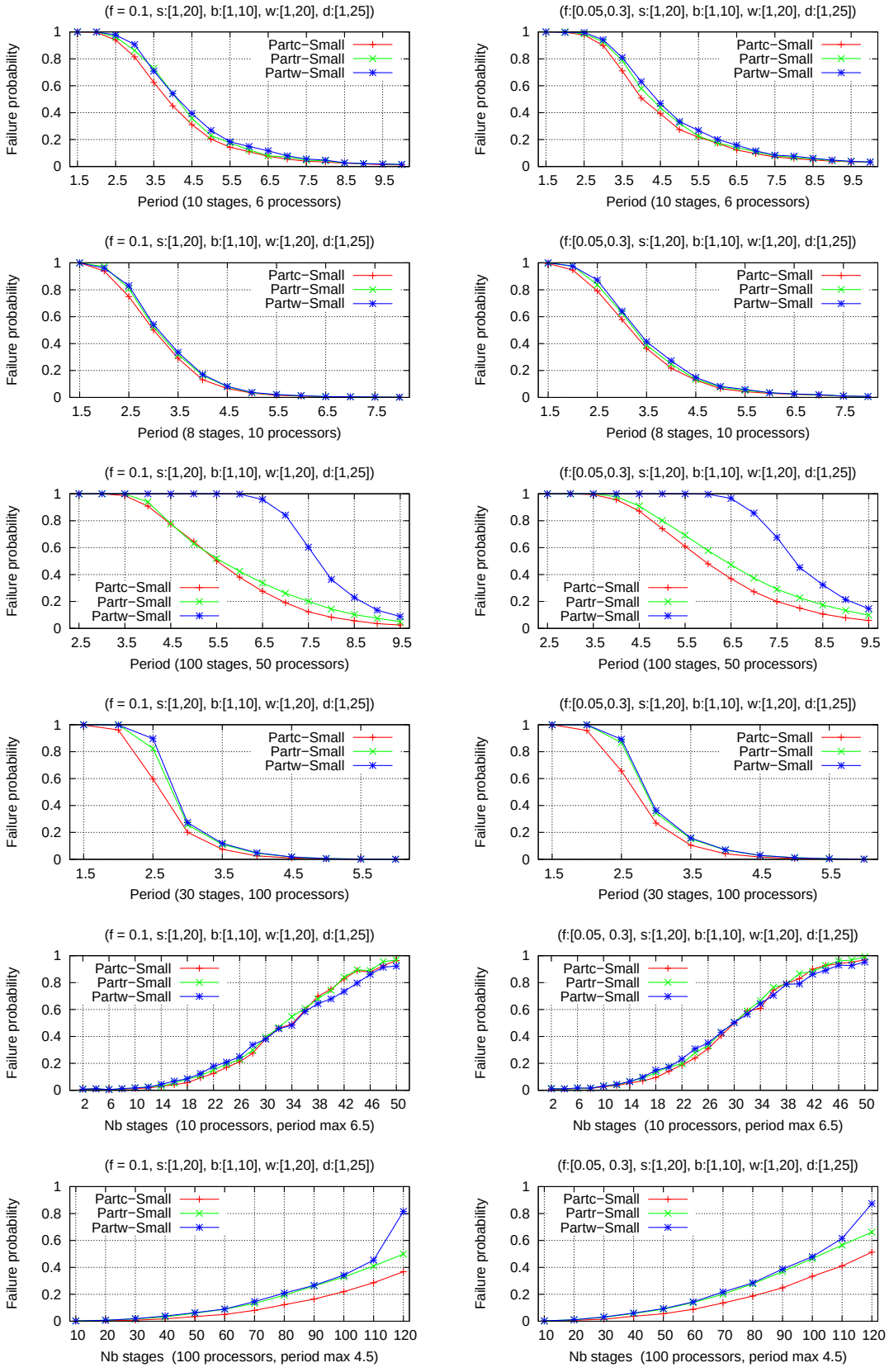

Figure 11: Comparison of {**Partc**|**Partr**|**Partw**}-**Small** heuristic variants on *Failure Homogeneous* (right column) and *FullHet* (left column) platforms.

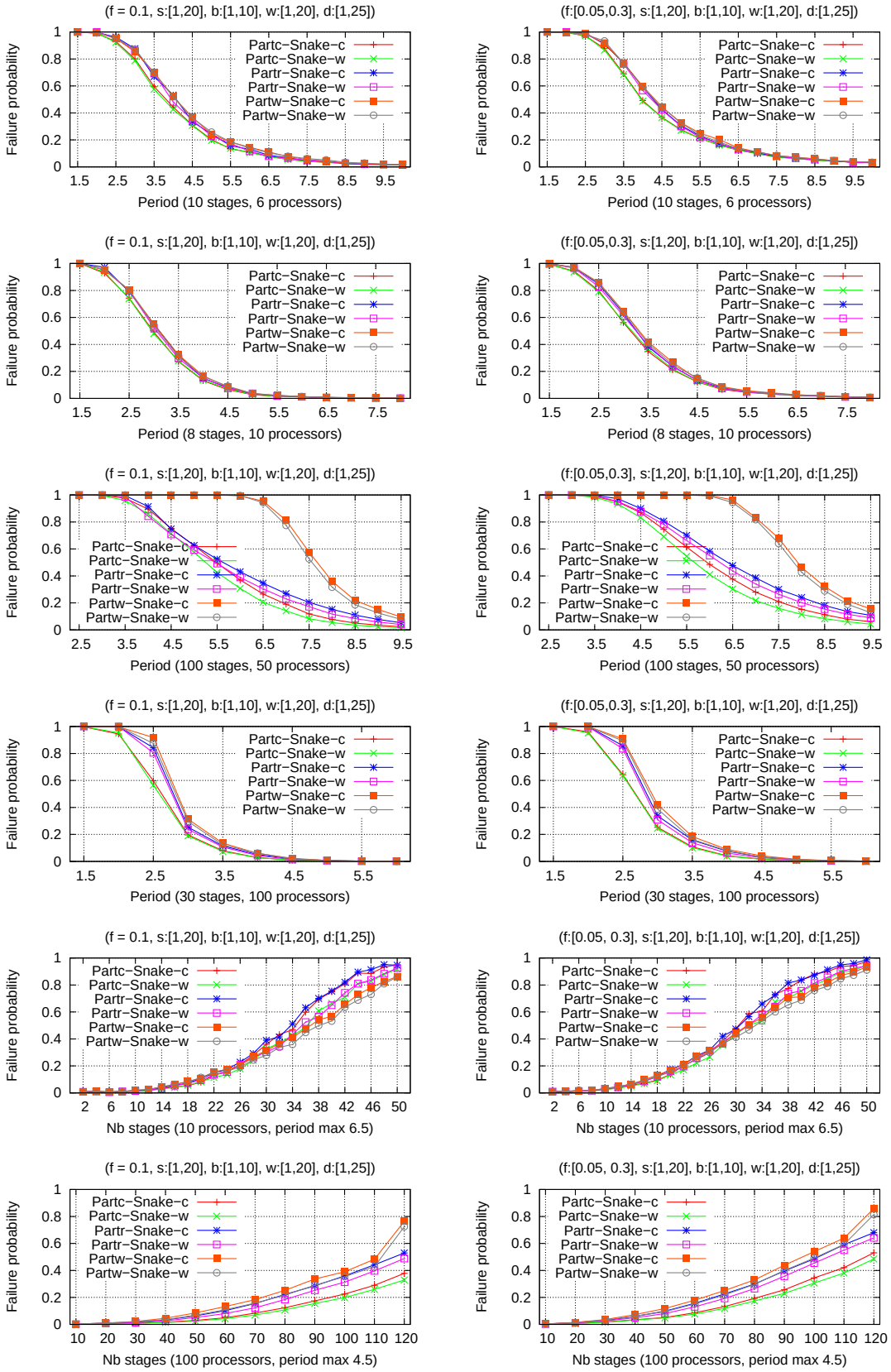

Figure 12: Comparison of {**Partc**|**Partr**|**Partw**}-**Snake**-{c|w} heuristic variants on *Failure Homogeneous* (right column) and *FullHet* (left column) platforms.

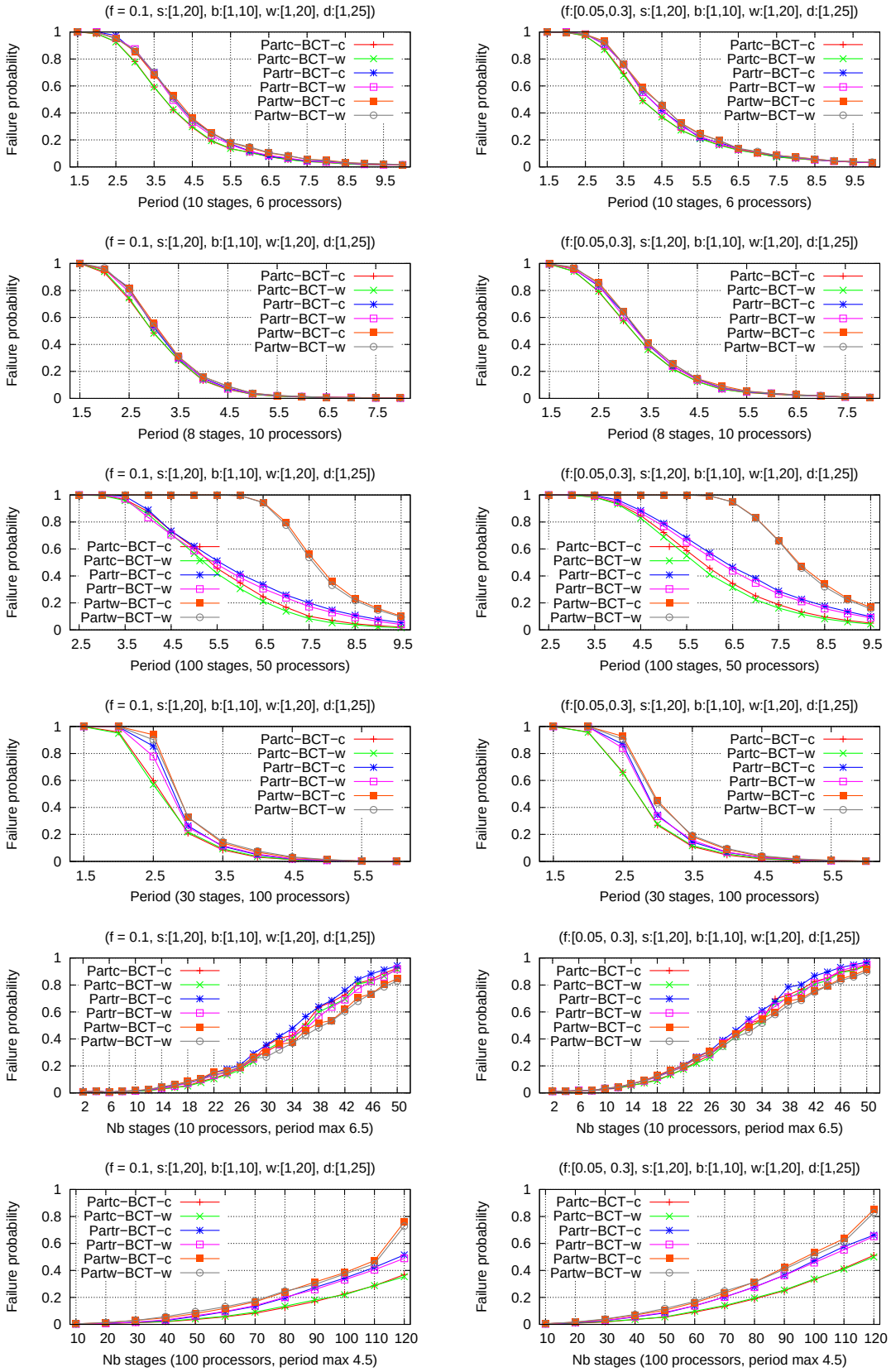

Figure 13: Comparison of {**Partc**|**Partr**|**Partw**}-**BCT**-{c|w} heuristic variants on *Failure Homogeneous* (right column) and *FullHet* (left column) platforms.

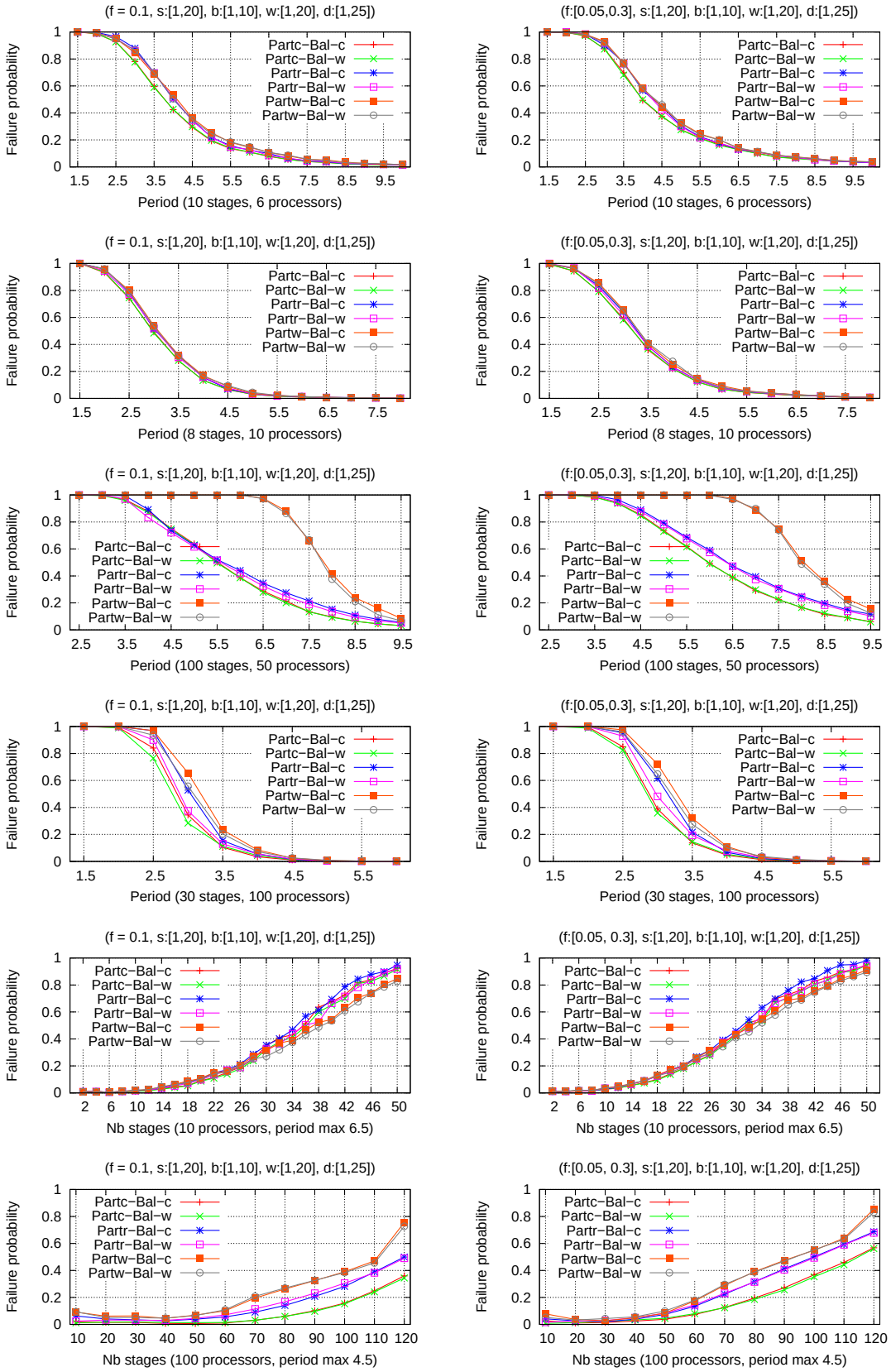

Figure 14: Comparison of {**Partc**|**Partr**|**Partw**}-**Bal**-{c|w} heuristic variants on *Failure Homogeneous* (right column) and *FullHet* (left column) platforms.

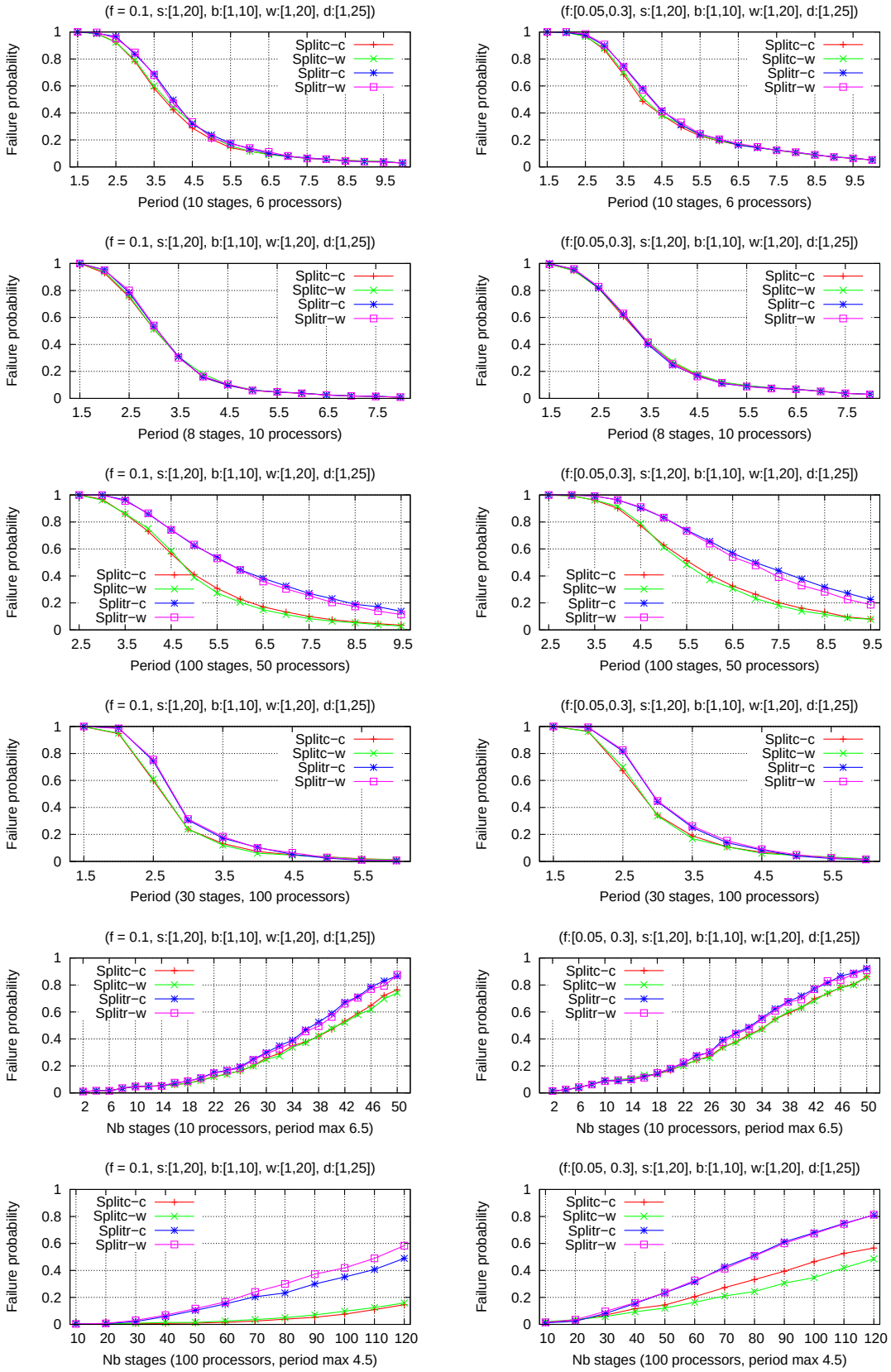

Figure 15: Comparison of {**Splitc**|**Splitr**}-{**c**|**w**} heuristic variants on *Failure Homogeneous* (right column) and *FullHet* (left column) platforms.