Assessing general mappings for period/reliability optimization of streaming applications

Anne Benoit,
Hinde Lilia Bouziane,
Yves Robert

Research Report N° 2010-20

Assessing general mappings for period/reliability optimization of streaming applications

Anne Benoit, Hinde Lilia Bouziane, Yves Robert

Abstract

This report deals with the problem of mapping pipelined applications on heterogeneous platforms whose processors are subject to failures. We address a difficult bi-criteria problem, namely deciding which stages to replicate, and on which resources, in order to optimize the reliability of the schedule, while guaranteeing a minimal throughput. Previous work had addressed the complexity of *interval* mappings, where the application is partitioned into intervals of consecutive stages (which are then replicated and assigned to processors). In this report we investigate *general* mappings, where stages may be partitioned without any constraint, thereby allowing a better usage of processors and communication network capabilities. The price to pay for general mappings is a dramatic increase in the problem complexity. We show that computing the period of a *given* general mapping is an NP-complete problem, and we provide polynomial bounds to determine a (conservative) approximated value. The bi-criteria mapping problem itself becomes NP-complete on homogeneous platforms, while it is polynomial with interval mappings. We design a set of efficient heuristics, which we compare with interval mapping strategies through extensive simulations.

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

Contents

1 Introduction

This report deals with the problem of mapping pipelined applications on heterogeneous platforms whose processors are subject to failures. We address a difficult bi-criteria problem, namely deciding which tasks to replicate, and on which resources, in order to optimize the reliability of the schedule, while guaranteeing a maximal period. On the one hand, the period of a mapping is defined as the longest cycle time of a processor. Under a bounded multi-port platform model with overlap [9], 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. Hence the period is the inverse of the throughput, and measures the aggregate performance of the mapping. On the other hand, 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: 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 builds upon 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 [4] onto heterogeneous platforms. A first attempt to solve the period/reliability problem can be found in our previous work [3], with the restriction that mapping rules impose stages to be partitioned into intervals. Here we tackle the most difficult framework, that of arbitrary general mappings. General mappings allow for the best usage of processors and communication network capabilities, at the price of a dramatic increase in the problem complexity.

We point out that the makespan/reliability problem has also been addressed in [6, 1, 8] for general DAGs of operations. The paper [6] does not replicate the operations and has thus a very limited impact on the reliability. Moreover, minimizing the makespan for a DAG corresponds to minimizing the end-to-end delay, or latency, in a pipelined application, and is therefore not directly related to period (i.e., throughput) optimization.

The rest of the report is organized as follows: the additional difficulties induced by general mappings are illustrated by working out an example in Section 2; then we formally detail the problem in Section 3, and establish complexity results in Section 4. Next in Section 5, we introduce a linear program to compute the period of a given general mapping with replication on fully heterogeneous platforms. Section 6 prensents several polynomial-time heuristics to compute a mapping period as well as to solve the mapping problem. Section 7 is devoted to experimental results. Finally, Section 8 provides some concluding remarks and directions for future work.

2 Motivating example

Consider the example shown in Figure 1. Each stage of the pipeline is specified by the size of its input/output data and a number of computations to be executed on each data set. According to the pipelined execution model, all stages operate on an (possibly infinite) input stream of data. As soon as the first output data of stage S_3 is produced, the pipeline reaches a steady state and periodically produces a result. The period depends on the mapping. For instance, if we assume that S_1 and S_2 are both assigned to the same set of processors, while S_3 is assigned to other processors, we obtain the scenario shown in Figure 1. The notation

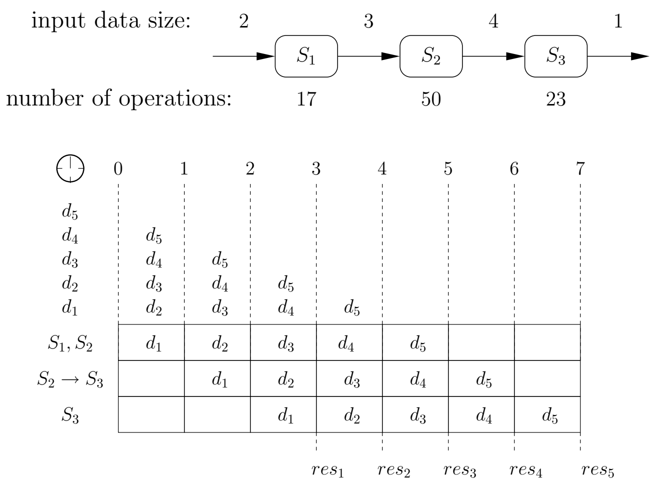

Figure 1: A pipelined application with three stages S_1, S_2, S_3, and its periodic execution; $d_1, \ldots, d_5$ are the first five data sets entering the pipeline, and $res_1, \ldots, res_5$ are the corresponding results.

$S_i \rightarrow S_{i+1}$ denotes a remote communication from the processor assigned to S_i to the one assigned to S_{i+1}. For this example, the period of the mapping is 1, which means that a new data set enters the pipeline every time unit. To execute the application, we target a platform composed of 12 heterogeneous processors $P_1, ..., P_{12}$ interconnected as a virtual clique. Processors have different speeds ($s_1 = 45$ operations per time unit for processors P_1 to P_6, and $s_2 = 55$ for P_7 to P_{12}) and different network card capacities ($B_1 = 20$ data emitted/received per time unit for P_1 to P_6, and $B_2 = 28$ for P_7 to P_{12}). Network links are homogeneous (identical bandwidth $b = 10$ data per time unit). In addition, we assume that processors are subject to unrecoverable failures: each processor has a probability $f = 0.15$ to fail during the whole application execution.

As stated above, to map the application onto the target platform, we can use either interval mappings, or general mappings. These two mapping rules are combined with replication strategies to increase the reliability. But to avoid producing redundant results if two or more processors assigned to the same stage(s) do not fail, a consensus protocol is enforced. The protocol elects one surviving processor as the sender of the output result to all processors executing the next stage. Intuitively, this election aims at choosing the surviving processor that allows for the fastest output communications. The goal is to partition the stages into subsets (intervals or arbitrary sets) and to assign processors to these subsets, so as to maximizing the reliability (or minimizing the failure probability), given a threshold on the period that should not be exceeded.

Figure 2 and Figure 3 show, in order, an optimal interval and general mapping, when the threshold period is set to $\mathcal{P}_{\max} = 1$. We now provide a comparison of these mappings. For the optimal *interval* mapping, each stage S_1 to S_3 is an interval. The period is the maximum cycle time over all processors $P_1, .., P_{12}$. Recall that we assume an overlap of communications and computations (see Figure 1). The cycle time of processor P_1 is $CT_1 = \max\left(\frac{2}{10}, \frac{2}{20}, \frac{17}{45}, \frac{3}{10}, \frac{3*4}{20}\right) = 0.6$. The first and second terms correspond to input data (size 2, link bandwidth 10, input network card 20), the third to computations (stage weight 17 divided by speed 45), the fourth and the last to output data (size 3, sent 4 times, link bandwidth

10, output network card 20). Similarly, $CT_2 = CT_3 = CT_4 = 0.6$, $CT_5 = CT_6 = 0.57$, $CT_7 = CT_8 = CT_9 = CT_{10} = 0.91$ and $CT_{11} = CT_{12} = 0.42$. Note that $P_7, \dots, P_{10}$ are critical resources, as they reach the largest cycle time. This time determines the period reachable in the case where for instance processors P_4 and P_5 fail (see Figure 2).

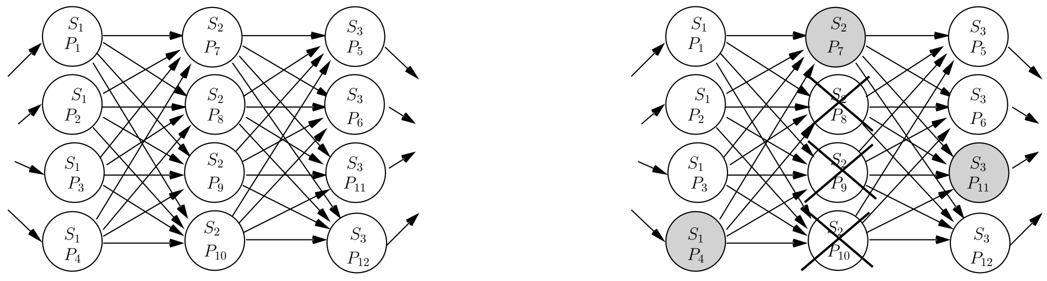

Figure 2: An *interval* mapping with a failure configuration reaching the worst-case period: P_8, P_9 and P_{10} have failed. Colored processors are elected.

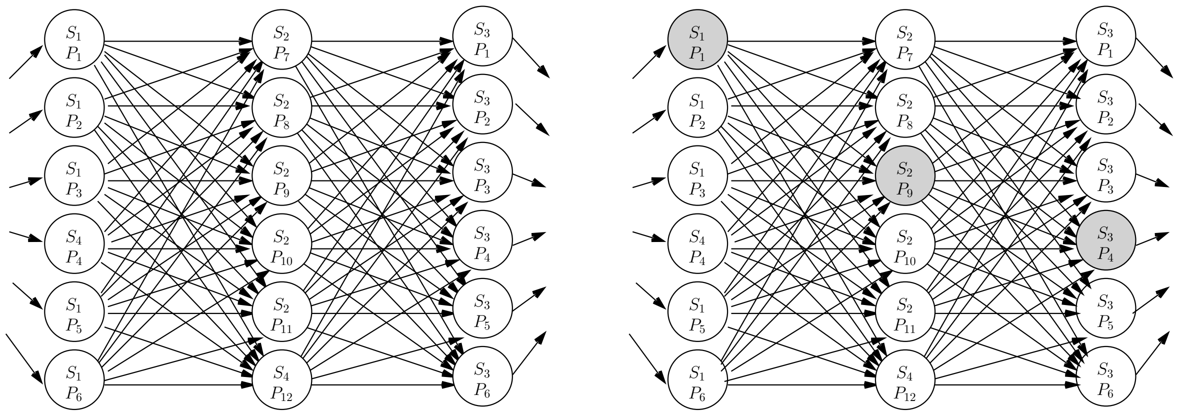

Figure 3: A *general* mapping, where the worst-case period is reduced by electing different processors for S_1 and S_3. Colored processors are elected.

For the optimal *general* mapping shown in Figure 3, stages are partitioned into 2 subsets $\{S_1, S_3\}$ and $\{S_2\}$. At first sight, the period of the mapping could be thought to be the maximum cycle time of all processors, as for interval mappings. For instance, we compute $CT_1 = \max\left(\frac{2}{10}, \frac{4}{10}, \frac{2+4}{20}, \frac{17+23}{45}, \frac{3}{10}, \frac{1}{10}, \frac{3*6+1}{20}\right) = 0.95$. The first three terms correspond to input data (size 2 for S_1 and 4 for S_3, link bandwidth 10, input network card 20). The fourth term corresponds to computations (sum of stage weights 17+23 divided by speed 45), while the last terms correspond to output data (size 3 for S_1 and 1 for S_3, sent 6 times, link bandwidth 10, output network card 20). Similarly, we obtain $CT_i = 0.95$ if $i \leq 6$ and and $CT_i = 0.91$ if $i \geq 7$. But these times assume that the same processor is elected to send the results of both S_1 and S_3. If different processors are elected for both stages, outgoing communication times may be reduced. Indeed, see Figure 3, where the maximum cycle time becomes $CT_1 = 0.9$ rather than 0.95. In fact, computing the period of a given general mapping will be shown NP-complete in Section 4 (as opposed to *interval* mappings, where identifying the critical resource is easy).

The failure probability $\mathcal{F}$ for both *interval* and *general* mappings is computed as 1 minus the probability that the execution is successful. This happens if and only if all subsets are successful. In turn, a given subset fails if and only if all its assigned processors fail. For the *interval* mapping of Figure 2, $\mathcal{F}_{int} = 1 - (1 - 0.15^4) * (1 - 0.15^4) * (1 - 0.15^4) = 0.0015$, and for the *general* mapping of Figure 3, $\mathcal{F}_{gen} = 1 - (1 - 0.15^6) * (1 - 0.15^6) = 0,000023$. These values are the smallest that can be obtained among the exponential number of all possible *interval* and *general* mapping solutions. The *general* mapping reaches a better result, as it enables the assignment of both S_1 and S_3 on the same processors, thereby allowing to explore a larger solution space. At the same time, it better exploits the target platform capabilities (processors and links), while satisfying the period constraint. Altogether, *general* mappings turn out much more complicated than *interval* mappings, but may considerably improve the performance results for the bi-criteria period/reliability optimization problem.

3 Framework

3.1 Applicative framework

A pipelined application is composed of n ordered stages S_i, $1 \leq i \leq n$, which continuously operate on a stream of data. When input data are fed 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 4.

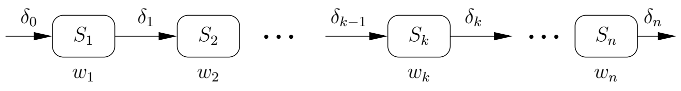

Figure 4: Overview of a pipelined application.

3.2 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. Processors may have identical speeds ($s_u = s$ for $1 \leq u \leq p$). In this case a platform is said to be *SpeedHom* (homogeneous in speed). On the opposite, the platform is *SpeedHet*.

As shown in Figure 5, 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 during the whole execution time 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 [2, 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 be seen as a global indicator of the reliability of a processor. When the failure probabilities of all processors are identical ($f_u = f$ for all $1 \leq u \leq p$), the platform is said *FailHom*. Otherwise, it is said *FailHet*.

Finally, a target platform can be specified according to different combinations of processors and links properties. In this work we consider the most general case of ***Fully Heterogeneous*** platforms (*FullHet*). These platforms are both *SpeedHet*, *CardHet*, *LinkHet* and *FailHet*.

3.3 Communication model

In the present work, we assume that communications between processors follow the *bounded multi-port model* [9]. In other words, 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 like MPICH2 [10]. The *bounded* characteristic

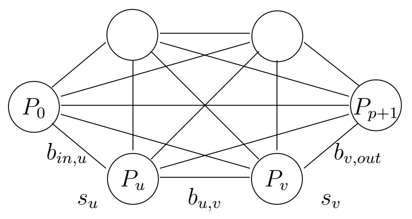

Figure 5: The target platform.

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 the stages are assigned to processors, different rules may be adopted. For instance, in a *one-to-one* mapping, each stage is assigned to a distinct processor, and each processor processes only one single stage. A less restrictive rule, *interval* mapping, is such that a processor may process a consecutive subset of stages. Last, the *general* mapping allows a processor to be assigned any subset of stages. In this section we introduce both *interval* and *general* mappings. Whe then focuses on *general* mappings to extend our previous study of *interval* mappings presented in [3].

In the following, a formal definition is given for *interval* and *general* mappings as well as for the adopted replication model to deal with processors failures. The period and failure probability of a pipelined application are then expressed for a given mapping.

Interval and general mappings: in an *interval* or a *general* 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 $d_j \leq i \leq e_j$. The difference between *interval* mapping and *general* mapping comes from the fact that in *interval* mapping, a processor can not be assigned more than one interval, i.e., $alloc(d_j) \cap alloc(d_{j'}) = \emptyset$ for $1 \leq j, j' \leq m$, $j \neq j'$. While in a *general* mapping, a processor can process multiple intervals. The set of intervals $\{[d_j, e_j] | 1 \leq j \leq m\}$ is then partinionned into l subsets $(l \leq m)$. We denote by $subSet_k$ $(k \leq l)$ a set of intervals, and by $allocS(subSet_k) = alloc(d_j)$ $(I_j \in subSet_k)$ the set of processors assigned to $subSet_k$. Finally, a processor can not be assigned more than one subset, i.e., $allocS(subSet_k) \cap allocS(subSet_{k'}) = \emptyset$ for $1 \leq k, k' \leq l$, $k \neq k'$.

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 $(u \in allocS(subSet_k), 1 \leq k \leq l)$ process the same assigned subset of intervals on the same input data. Therefore, the output data of an interval $I_j \in subSet_k$ 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 $(u \in allocS(subSet_k))$ after the execution of each interval $I_j \in subSet_k$ on an input data. The consensus aims at electing a processor per interval $I_j \in subSet_k$ as the sole one that will send the output data of I_j to all surviving processors P_v. Note that if $|subSet_k| \geq 2$ and $|allocS(subSet_k) \geq 2$, different processors may be elected for different intervals in $subSet_k$. The replication protocol is illustrated in Figure 6, where all processors are surviving. This protocol allows to pay at most $\sum_{I_j \in subSet_k} |alloc(d_{j+1})|$

outgoing communications by an elected processor (according to the *bounded multi-port* communication model) and $|subSet_{k'}|$ incoming communications by P_v, where $I_{j+1} \in subSet_{k'}$. 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 elected processors are accounted for in the performance model.

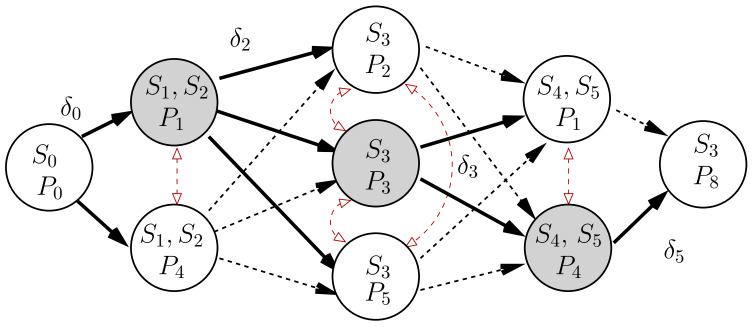

Figure 6: Replication model. Each processor periodically receives input data from elected predecessors (on the plain incoming arrow), executes all assigned intervals, exchanges extra messages (on dashed vertical arrows) with processors allocated to the same intervals, agrees upon which processor (filled circle) for each assigned interval has to send the result (on plain outgoing arrows) to all the successors allocated to the next interval.

Period: as stated and proved after in Section 4, computing the period $\mathcal{P}$ of a given mapping in the most general case of *general* mapping with replication on *Fully Heterogeneous* platforms is NP-complete. Therefore, computing the period will rely on a linear programming and heuristics based solutions (Section 5 and Section 6.1). However, we note that in the restrictive case of *interval* mapping, $\mathcal{P}$ can be computed by the following formula:

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

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 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). Last, by considering the maximum of these cycle times for $u \in alloc(d_j)$, the period is the global maximum over all intervals.

For *general* mapping, a similar principle for computing the cycle time of a processor is reused later, but with considering multiple intervals assignment to a same processor as well as possibly crached processors.

Failure probability: the failure probability $\mathcal{F}$ of a pipelined application in the most general situation of *general* mapping on *Fully Heterogeneous* platforms is computed by the following formula:

$$\mathcal{F} = 1 - \prod_{1 \leq k \leq l} \left(1 - \prod_{u \in allocS(subSet_k)} f_u \right). \tag{2}$$

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 among processors allocated to each set $subSet_k$ of intervals, i.e., for $1 \leq k \leq l$.

In the rest of this report, our aim is to determine a *general* mapping minimizing the failure probability $\mathcal{F}$ given a threshold period $\mathcal{P}_{\max}$. The mapping must be such that $\mathcal{P} \leq P_{\max}$. Note that such a constraint may reduce the solution space of this optimization problem to a subset of available processors. In fact, a processor may not satisfy this constraint, in which case, we assume that this processor is not assigned any interval.

4 Complexity results

Consider a given general mapping, and assume (for the sake of the analysis) that we know which processors have failed. The following result shows that computing the resulting period is NP-complete.

Definition 1. CONSENSUS *problem. Given a mapping, a set of failing processors, and a bound on the period $\mathcal{P}$, select one sending processor per interval so that the achieved period is less than or equal to $\mathcal{P}$.*

Theorem 1. CONSENSUS *is NP-complete.*

Considering a general mapping again, assume now that the set of failing processors is not known in advance (which is the realistic hypothesis in practice). For each failure configuration, there is a period that can be achieved. It is natural to ask what is the *worst-case period*, defined as the largest of these periods, over all possible failure configurations. However, depending upon the mapping, we may have an exponential number of such configurations, so we don't even know whether the worst-case period problem belongs to the class NP. However, consider a *no-failure* mapping, defined as a mapping where no processor can fail for the execution to be successful. It turns out that even for such (expectedly simpler) mappings, computing the period (which is then also the worst-case period, as the empty set is the only possible failure configuration), remains an NP-complete problem:

Theorem 2. *Computing the (worst-case) period of a no-failure mapping is NP-complete.*

Note that Theorem 1 is a consequence of Theorem 2, hence we only provide the proof of the latter.

Proof. The decision problem is obviously in NP. For the completeness, we use a reduction from an instance $\mathcal{I}_1$ of 3-Partition [7]: given $3n$ positive integers $a_1, a_2, \ldots, a_{3n}$ whose sum is nB, such that $\frac{B}{4} < a_i < \frac{B}{2}$ for all i, can we partition them into n triples T_j of sum B? The instance $\mathcal{I}_2$ of our problem is given by the following mapping of $7n+1$ stages (of identical weight $1/4$) onto $4n+1$ unit-speed processors labeled K_1 to K_n and U_0 to U_{3n}. All communication cards have capacity $B+1$, and all links have bandwidth B. For $i \leq n$, stage

S_i is assigned to processor K_i. Then for $0 \le i \le 3n$, stage S_{n+2i+1} is assigned to processor U_i, while in alternation for $1 \le i \le 3n$, stage S_{n+2i} is assigned to the set $\mathcal{K} = \{K_1, K_2, \dots, K_n\}$ (replication on all processors K_i). Finally, $\delta_i = 1$ for $0 \le i \le n$, $\delta_{n+2i+1} = \frac{B}{n}$ for $0 \le i \le 3n$ (outgoing communication from U_i), and $\delta_{n+2i} = a_i$ for $1 \le i \le 3n$ (outgoing communication from the i-th instance of $\mathcal{K}$). Here is an example with $n = 2$:

$$\xrightarrow{1} K_1 \xrightarrow{1} K_2 \xrightarrow{1} U_0 \xrightarrow{\frac{B}{n}} \mathcal{K} \xrightarrow{a_1} U_1 \xrightarrow{\frac{B}{n}} \mathcal{K} \xrightarrow{a_2} U_2 \xrightarrow{\frac{B}{n}} \mathcal{K}$$
$$\xrightarrow{a_3} U_3 \xrightarrow{\frac{B}{n}} \mathcal{K} \xrightarrow{a_4} U_4 \xrightarrow{\frac{B}{n}} \mathcal{K} \xrightarrow{a_5} U_5 \xrightarrow{\frac{B}{n}} \mathcal{K} \xrightarrow{a_6} U_6 \xrightarrow{\frac{B}{n}}$$

Because of the first n stages, no K_i can fail. Since each U_i is the unique processor assigned to its stage, no processor can fail during a successful execution, and we do have a no-failure mapping. We ask whether we can achieve a period $\mathcal{P} = 1$.

Suppose first that $\mathcal{I}_1$ has a solution with n triples T_j. For $a_i \in T_j$, we elect K_j for the outgoing communication of size a_i (that is also the communication incoming to U_i). Each K_j has a total outgoing volume of $B + 1$: the first communication of size 1, and the three of size a_i, whose sum is B. Each K_j is assigned 4 stages of weight 1/4. We easily check that the resulting period is $\mathcal{P} = 1$, hence a solution to I_2.

Suppose now that $\mathcal{I}_2$ has a solution with $\mathcal{P} = 1$. The total volume outgoing of the set $\mathcal{K}$ is $n(B + 1)$. No K_i can send more than $B + 1$, so they all send exactly $B + 1$. This leads to the desired partition for the solution of $\mathcal{I}_2$. $\square$

5 Mixed integer linear program formulation to compute the period

This section studies an optimal solution to the CONSENSUS problem. As stated in Section 4, this problem is NP-complete for *general* mappings on *FullHet* platforms. Thus, we introduce a mixed integer linear program that computes a consensus reaching the minimum period for a given *general* mapping and for a configuration of processors failures.

We recall that a pipelined application is composed of n stages to be mapped onto a 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 introducing the parameters of the program and its 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^i_u ($u \in [1..p]$): the input network card capacity of processor P_u.
- B^o_u ($u \in [1..p]$): 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 $P_u \leftrightarrow P_v$.
- $\mathit{surviving} = \{u | P_u$ is surviving $(0 \leq u \leq p+1)\}$: the set determining the surviving processors in the input mapping. All processors P_v $(1 \leq v \leq p)$ with $v \notin \mathit{surviving}$ are assumed to be crashed. Hypothesis: $0 \in \mathit{surviving}$ and $p+1 \in \mathit{surviving}$.
- $\Delta_{i,u}$ ($i \in [0..n+1]$ and $u \in \mathit{surviving}$): a boolean variable equal to 1 if stage S_i is assigned to surviving processor P_u. Hypothesis: $\Delta_{0,0} = \Delta_{n+1,p+1} = 1$, $\Delta_{0,u} = 0$ for $(1 \leq u \leq p+1)$, $\Delta_{n+1,u} = 0$ for $(0 \leq u \leq p)$, $\Delta_{i,0} = 0$ for $(1 \leq i \leq n+1)$ and $\Delta_{i,p+1} = 0$ for $(0 \leq i \leq n)$.

Decision variables:

- P: the period to minimize.
- $x_{i,u}$ $(i \in [0..n], u \in \mathit{surviving})$: a boolean variable equal to 1 if P_u is elected for stage S_i $(i \in [0..n])$. Hypothesis: $x_{0,0} = 1$, $x_{0,u} = 0$ for $(1 \leq u \leq p+1)$, $x_{i,0} = 0$ for $(1 \leq i \leq n)$ and $x_{i,p+1} = 0$ for $(0 \leq i \leq n)$.

Constraints:

- Conditions to elect a processor:
  - A processor P_u may be elected for stage S_i if it is allocated to this stage:

$$\forall i \in [1..n], \forall u \in \mathit{surviving}, x_{i,u} \quad \leq \quad \Delta_{i,u}$$

  - Only one processor is elected for each stage of the pipeline:

$$\forall i \in [1..n], \sum_u \Delta_{i,u} x_{i,u} \quad = \quad 1$$

- Cycle time of processors:
  - The computation load of each surviving processor is expressed as:

$$\forall u \in \mathit{surviving}, \sum_i \frac{w_i}{s_u} \Delta_{i,u} \quad \leq \quad P$$

  - Outgoing communications[1] are done by an elected processor and are expressed as:

$$\forall u,v \in \mathit{surviving}, \sum_i \frac{\delta_i \Delta_{i+1,v}(1-\Delta_{i,v})}{b_{u,v}} x_{i,u} \quad \leq \quad P$$
$$\forall u \in \mathit{surviving}, \sum_i \frac{\delta_i \sum_v \Delta_{i+1,v}(1-\Delta_{i,v})}{B^o_u} x_{i,u} \quad \leq \quad P$$

  Note that these constraints assume an elected processor to be aware about crashing processors. Thus output data are sent to surviving processors only.

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

– Incoming communications[1] done by any surviving processor are expressed as:

$$\begin{aligned} \forall u, v \in surviving, \sum_i \frac{\delta_{i-1}\Delta_{i,u}(1-\Delta_{i-1,u})}{b_{v,u}} x_{i-1,v} &\leq P \\ \forall u \in surviving, \sum_i \frac{\delta_{i-1}\Delta_{i,u}(1-\Delta_{i-1,u})}{B_u^i} &\leq P \end{aligned}$$

Objective function: we aim at finding values for each variable $x_{i,u}$ in order to minimize P, given that all constraints are satisfied.

6 Heuristics

In this section, we first propose heuristics for computing (in fact, approximating) the period of a given *general* mapping with replication. Then, we propose heuristics for determining a mapping which optimizes the failure probability $\mathcal{F}$ under a fixed period bound $P_{\max}$. We study several strategies to partition application stages into subsets of intervals which will be assigned to processor sets. We propose two classes of mapping heuristics. In the first class (Section 6.2), interval subsets are computed before addressing their mapping. In the second class (Section 6.3), processors are grouped into sets before addressing their assignment, and interval subsets are computed on the fly during the mapping process. We aim at exploring quite a large set of mapping solutions, in order to produce a final mapping with small failure probability.

6.1 Heuristics for period computation

As stated in Section 4, computing the worst-case period for *general* mapping on *FullHet* platforms with replication is NP-complete. For a given mapping and a failure configuration, this period depends on elected processors performing remote outgoing communications over all interval subsets. We thus propose four heuristics[2] that explore different election strategies. Once a processor is elected for each interval in each interval subsets, the cycle-time of surviving processors can be computed, thus a corresponding period. The heuristics aim at computing a small reachable worst-case period.

Before presenting heuristics, we introduce how the cycle-time of a processor is computed. According to the communication and replication models introduced in Section 3.3, the cycle-time of a surviving processor P_u allocated to an interval subset $subSet_k$ is computed as the maximum between the time needed to **1)** compute intervals in $subSet_k$, **2)** to receive input data from each remote elected processor assigned to an interval I_{j-1} with I_j assigned to P_u, and **3)** to send output data to remote surviving successors of P_u allocated to each interval I_{j+1} such that P_u is elected for I_j. Algorithm 1 details how the cycle-time of a processor is computed. This algorithm is applied by the following heuristics when computing the period for a given mapping. We recall that the period is defined as the biggest cycle-time over all surviving processors (denoted by $surviving$).

[2]Heuristics computing the period are designed for *general* mappings with replication and without any processor allocation constraints (a same processor may be allocated to different interval subsets).

Algorithm 1: Compute the cycle-time of a surviving processor P_u given a mapping, a failure configuration and an elected processor by each stage of the pipeline.

```
begin
    // Workload
    Initialize a set $Stages_u$ with all stages $S_i$ $(1 \leq i \leq n)$ assigned to $P_u$
    $load_u = \frac{\sum_{S_i \in Stages_u} w_i}{s_u}$
    // Outgoing communications
    Initialize a set $Electedfor_u$ with all stages $S_i$ $(1 \leq i \leq n)$ for which $P_u$ is elected
    and such that $u \notin alloc(i+1)$. For each stage in $Electedfor_u$, $P_u$ performs a
    remote outgoing communication
    $sent_u = \frac{\sum_{i \in Electedfor_u} \delta_i |alloc(i+1) \cap surviving|}{B_u^o}$
    foreach surviving processor $P_v$ successor of $P_u$ do
        // a successor $P_v$ is such that $v \in alloc(i+1)$, $i \in Electedfor_u$
        $comm_{u,v}^o = \sum_{i \in Electedfor_u} \frac{\delta_i mapping[i+1][v]}{b_{u,v}}$
        // $mapping[j][r] = 1$ if stage $S_j$ is assigned to processor $P_r$
    end
    Set $comm_u^o$ to the maximum obtained value $comm_{u,v}^o$
    // Incoming communications
    $recv_u = \sum_{i \in Stages_u} \frac{\delta_{i-1}(1 - mapping[i-1][u])}{B_u^i}$
    foreach elected processor $P_v$ predecessor of $P_u$ do
        // a predecessor $P_v$ is such that $v \in alloc(i-1)$ and $i-1 \notin Stages_u$
        Initialize a set $Electedfor_v$ with all stages $S_i$ $(0 \leq i \leq n)$ for which $P_v$ is elected
        and such that $v \notin alloc(i+1)$.
        $comm_{v,u}^i = \sum_{i \in Electedfor_v} \frac{\delta_i (mapping[i+1][u])}{b_{v,u}}$
    end
    Set $comm_u^i$ to the maximum obtained value $comm_{v,u}^i$
    return $\max(load_u, comm_u^o, comm_u^i)$
end
```

Random: random election – For each stage of a pipeline, this heuristic randomly elects a surviving assigned processor. Once all stages are treated, the heuristic returns the biggest cycle time of surviving processors as the period of the mapping. This heuristic is detailed in Algorithm 2. Note that only the election for the last stage of an interval assigned to a processor accounts for a remote communication. All local communications are negligible.

Algorithm 2: **Random** heuristic: greedy election of processors for a given mapping and a given failure configuration (crashed processors). The heuristic returns the resulted mapping period.

```
begin
    for $i = 1$ to $n$ do
        Perform a random election of a surviving processor $P_u$ ($1 \leq u \leq p$) allocated to
        stage $S_i$
        Mark this processor as elected for $S_i$
    end
    foreach $u \in surviving$ do
        Compute the cycle-time of $P_u$ by applying Algorithm 1
    end
    return the biggest resulted cycle-time (equal to the period)
end
```

MaxBout: biggest *Bout* – For each stage of a pipeline, this heuristic elects the surviving assigned processor with the biggest *Bout* (for fast data emission). Once all stages are treated, the heuristic returns the biggest cycle time of surviving processors as the period of the mapping. This heuristic is detailed in Algorithm 3. Only the election for the last stage of an interval assigned to a processor accounts for a remote communication.

Algorithm 3: **MaxBout** heuristic: greedy election of processors with the biggest B^o. The heuristic returns the resulted mapping period.

```
begin
    for $i = 1$ to $n$ do
        Elect a surviving processor $P_u$ ($1 \leq u \leq p$) allocated to stage $S_i$, having the
        biggest $B^o_u$
        Mark this processor as elected for $S_i$
    end
    foreach $u \in surviving$ do
        Compute the cycle-time of $P_u$ by applying Algorithm 1
    end
    return the biggest resulted cycle-time (equal to the period)
end
```

MinComm: minimum communication time – This heuristic attempts to minimize output communications of elected processors in a set $allocS(subSet_k)$. It starts by considering each stage which is allocated to only one processor, and this processor is elected for this stage (no other alternative). In a second step, the heuristic repeatedly elects a processor for the stage S_i with the biggest time needed to send its output data on a communication link to another surviving processor. The elected processor is one of those achieving the smallest outgoing communication time. This time considers all other stages for which this processor is already elected. If a stage S_i and its successor S_{i+1} are assigned to exactly the same surviving processors, a processor is randomly elected for S_i and results to negligible communications. Once all stages are treated, the heuristic computes and returns the period of the mapping. This heuristic is detailed in Algorithm 4.

MinNbProcs: smallest number of processors – This heuristic gives a priority to stages with less alternatives to design an elected processor. It also attempts to minimize output communications of elected processors. The heuristic repeatedly elects a processor for the stage S_i with the smallest number of assigned processors (surviving ones). If equality, the stage with largest output data messages ($\delta_i(|alloc(i+1) \cap surviving) \setminus alloc(i)|)$ has the priority. The elected processor is one of those achieving the smallest outgoing communication time. This time considers all other stages for which this processor is already elected. In addition, if a stage S_i and its successor S_{i+1} are assigned to exactly the same surviving processors, a processor is randomly elected for S_i and results to negligible communications. Once all stages are treated, the heuristic computes and returns the period of the mapping. This heuristic is detailed in Algorithm 5.

In the following, both **Random**, **MaxBout**, **MinComm** and **MinNbProcs** heuristics are applied when computing a mapping. The smallest result provided by these heuristics is defined as the period of the mapping.

6.2 Mapping heuristics, class 1: partitioning stages then mapping

This class of mapping heuristics extends heuristics defined for interval mappings [3] to the case of general mappings. As in [3] there are two steps: first, a pipeline is partitioned into subsets of intervals. Then, these subsets are mapped onto the platform in such a way that the period bound is satisfied, and the reliability of the mapping is computed. We try several partitioning techniques, and keep the solution which returns the most reliable mapping.

6.2.1 Partitioning stages

The partitioning phase returns different partitions created by varying the number of target interval subsets and some partitioning criteria. In more details, stages are distributed over k subsets of intervals, with $1 \leq k \leq \min(n, p)$, according to one of the following criteria:

- Communication cost: the stage with smallest output data size (δ_i) is affected to the interval subset minimizing the maximum between the workload ($\sum_{I_j=[d_j,e_j] \in subSet_k} \sum_{i \in I_j} w_i$) of the subset and its output data size ($\sum_{I_j=[d_j,e_j] \in subSet_k} \delta_{e_j}$). Thus, costly computations can be avoided and longest remote communications can be replaced by local memory accesses on a processor. Heuristics using this partitioning criteria are identified by a prefix **PartStc**.

- Computation cost: stages are distributed over l interval subsets such that the computation load of each subset approximates the average $\frac{\sum_{i=1}^{n} w_i}{l}$. Then, costly interval subsets in terms of computation may be reduced. Heuristics using this partitioning criteria are identified by a prefix **PartStw**.

Algorithm 4: **MinComm** heuristic: election of processors resulting to fastest outgoing communications. The heuristic returns the resulted mapping period.

```
begin
    foreach surviving processor $P_u$ ($1 \leq u \leq p$) do
        Initialize a set $Set_u$ of stages to empty. This set will contain stages for which $P_u$
        is elected
        foreach $S_i$ assigned to $P_u$ do
            if $alloc(i) = \{u\}$ then
                Add $S_i$ to $Set_u$
            end
        end
    end
    foreach $S_i \notin Set_u$ ($1 \leq u \leq p$) do
        if $alloc(i) \cap surviving = alloc(i+1) \cap surviving$ then
            // no remote outgoing communications are needed
            Add $S_i$ to $Set_v$, where $v \in alloc(i) \cap surviving$ is randomly chosen to be
            elected
        end
    end
    Order all stages not belonging to a set $Set_u$ ($1 \leq u \leq p$) by decreasing order of
    output data transfer times in list $L_i$. This time is expressed as:

    $$\frac{\delta_i}{min_{(u \in alloc(i) \cap surviving,\ v \in (alloc(i+1) \cap surviving) \setminus alloc(i))}(b_{u,v})}$$

    foreach $S_i \in L_i$ in order do
        Add $S_i$ to $Set_u$ such that processor $P_u$ ($1 \leq u \leq p$) results to the smallest value:

        $$max\left(\frac{|(alloc(i+1) \cap surviving) \setminus alloc(i)| * \delta_i + \sum_{S_k \in Set_u} |(alloc(k+1) \cap surviving) \setminus alloc(k)| * \delta_k}{B_u^o},\right.$$
        $$\left. max_{v \in (alloc(i+1) \cap surviving) \setminus alloc(i)} \frac{\delta_i + \sum_{S_k \in Set_u} \delta_k mapping[k+1][v](1 - mapping[k][v])}{b_{u,v}}\right)$$

        $P_u$ becomes elected for $S_i$
    end
    foreach $u \in surviving$ do
        Compute the cycle-time of $P_u$ by applying Algorithm 1
    end
    return the biggest resulted cycle-time (equal to the period)
end
```

Algorithm 5: **MinNbProcs** heuristic: election of processors with fastest outgoing communications and with election priority for stages assigned to smallest number of surviving processors. The heuristic returns the resulted mapping period.

```
begin
    foreach surviving processor $P_u$ ($1 \leq u \leq p$) do
        Initialize a set $Set_u$ of stages to empty. This set will contain stages for which $P_u$ is elected
    end
    foreach $S_i \notin Set_u$ ($1 \leq u \leq p$) do
        if $alloc(i) \cap surviving = alloc(i+1) \cap surviving$ then
            // no remote outgoing communications are needed
            Add $S_i$ to $Set_v$, where $v \in alloc(i) \cap surviving$ is randomly chosen to be elected
        end
    end
    Order all stages not belonging to a set $Set_u$ ($1 \leq u \leq p$) by increasing order of $|alloc(i) \cap surviving|$ in list $L_i$. In the case of equivalent values, order concerned stages by decreasing order of $(\delta_i(|alloc(i+1) \cap surviving) \setminus alloc(i)|)$
    foreach $S_i \in L_i$ in order do
        Add $S_i$ to $Set_u$ such that processor $P_u$ ($1 \leq u \leq p$) results to the smallest value:
        $max(\frac{|(alloc(i+1) \cap surviving) \setminus alloc(i)| * \delta_i + \sum_{S_k \in Set_u} |(alloc(k+1) \cap surviving) \setminus alloc(k)| * \delta_k}{B_u^o}, max_{v \in (alloc(i+1) \cap surviving) \setminus alloc(i)} \frac{\delta_i + \sum_{S_k \in Set_u} \delta_k mapping[k+1][v](1 - mapping[k][v])}{b_{u,v}})$
        $P_u$ becomes elected for $S_i$
    end
    foreach $u \in surviving$ do
        Compute the cycle-time of $P_u$ by applying Algorithm 1
    end
    return the biggest resulted cycle-time (equal to the period)
end
```

- Random partitioning: each stage is randomly affected to one of the k interval subsets. Heuristics using this criteria are identified by a prefix **PartStr**.

These different ways adopted for creating interval subsets aim at providing a good trade-off when mapping costly stages, in terms of computations and/or communication, on *FullHet* platforms.

6.2.2 Heuristics for mapping pre-defined subsets of intervals

In this section, we propose four heuristics and derive some variants. These heuristics differ in the way processors are allocated over input subsets of intervals and in the priority to order the assignment of these subsets. Input subsets are formed according to one of the previous partitioning criteria.

Small: smallest f_u – This greedy heuristic starts by randomly assigning each subset of intervals to one processor satisfying the period constraint. Then, it repeatedly assigns the subset with the highest failure probability to the more reliable processor. After all processors are considered, the heuristic attempts to improve the global failure probability. For that, it repeatedly performs a fusion of the subset having the highest failure probability with another subset such that the resulting failure probability is smaller than the original one (Algorithm 10). The fusion process is done as long as the failure probability of the whole mapping can be decreased and the period bound is still satisfied. The **Small** heuristic is further detailed in Algorithm 6.

Algorithm 6: **Small** heuristic: greedy mapping of l given subsets of intervals to most reliable processors, under a fixed period $\mathcal{P}_{\max}$.

```
begin
    for $k = 1$ to $l$ do
        Assign $subSet_k$ to a non-used processor $P_u$ randomly selected and satisfying the
        period $\mathcal{P}_{\max}$. The period of the current mapping is computed by applying
        Random, MaxBout, MinComm, MinNbProcs heuristics and by retaining
        the smallest result
        If success, mark $P_u$ 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 set $subSet_k$ with the highest failure probability and for
        which $P_u$ satisfies the period $\mathcal{P}_{\max}$ (application of Random, MaxBout,
        MinComm, MinNbProcs heuristics)
        If success, mark $P_u$ as used
    end
    Apply Algorithm 10 (fusion) to improve the failure probability of the current
    mapping
end
```

Snake: snake allocation – This heuristic starts by assigning each subset of intervals to the most reliable processor satisfying the period constraint. In a second step, each subset is assigned to the least reliable processor, and steps are repeatedly alternated. After all subsets and processors are treated, the heuristic attempts to improve the failure probability of the resulting mapping. For that, it performs the same fusion step as done by the **Small** heuristic (application of Algorithm 10). The **Snake** heuristic is further detailed in Algorithm 7. From this heuristic, we can derive some variants, depending upon the order in which subsets of intervals are considered for assignment. We define two variants **Snake-c** and **Snake-w**. **Snake-c** considers a decreasing order of output data size ($\sum_{I_j=[d_j,e_j]\in subSet_k} \delta_{e_j}$), while **Snake-w** considers subsets in a decreasing order of their workload ($\sum_{I_j=[d_j,e_j]\in subSet_k} \sum_{i=d_j}^{e_j} w_i$). Therefore, the mapping priority is given to costly subsets in terms of either output communications or workload.

Algorithm 7: **Snake** heuristic: snake allocation of p processors to l given subsets of intervals, under a fixed period $\mathcal{P}_{\max}$.

```
begin
    Order processors $P_u, 1 \leq u \leq p$ by increasing failure probability $f_u$ in list $Lp$
    Order input sets $subSet_k$ $(1 \leq k \leq l)$ by decreasing workload
    $(\sum_{I_j=[d_j,e_j]\in subSet_k} \sum_{i=d_j}^{e_j} w_i)$ in list $Ls$ (or decreasing output data size, i.e.,
    $(\sum_{I_j=[d_j,e_j]\in subSet_k} \delta_{e_j}))$
    for $i = 1$ to $roundUpInt(\frac{p}{l})$ do
        foreach $subSet_k \in Ls$ in order do
            Assign $subSet_k$ to the first processor $P_u$ found in $Lp$ that satisfies the period
            $\mathcal{P}_{\max}$. The mapping period is computed by applying Random, MaxBout,
            MinComm, MinNbProcs heuristics and by retaining the smallest result
            If success, remove $P_u$ from $Lp$
        end
        Inverse the order of processors in $Lp$
    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 set $subSet_k$ with the biggest failure probability and for which
        $P_u$ satisfies the period $\mathcal{P}_{\max}$ (application of Random, MaxBout, MinComm,
        MinNbProcs heuristics)
    end
    Apply Algorithm 10 (fusion) to improve failure probability of the resulted mapping
end
```

BCT: biggest cycle-time – this heuristic repeatedly considers each subset of intervals and searches the most critical processor, i.e., the processor with the longest cycle-time satisfying the period constraint, and allocates it to this subset. Once all subsets and processors have been treated, the heuristic attempts to improve the failure probability of the resulting mapping: it applies Algorithm 10, as done by previous heuristics. The **BCT** heuristic is further detailed in Algorithm 8. We can also derive some variants, depending upon the order in which subsets of intervals are treated. As for the **Snake** heuristic, we define two variants **BCT-c** and **BCT-**

w. **BCT-c** (respectively **BCT-w**) considers a decreasing order of output data size (resp. workload). We recall that the objective of such variants is to give a priority for mapping costly subsets of intervals.

Algorithm 8: **BCT** heuristic: mapping l given subsets of intervals on critical processors, under a fixed period $\mathcal{P}_{\max}$.

```
begin
    Order input sets $subSet_k$ $(1 \leq k \leq l)$ by decreasing computation load
    $(\sum_{I_j=[d_j,e_j] \in subSet_k} \sum_{i=d_j}^{e_j} w_i)$ in list $Ls$ (or decreasing output data size, i.e.
    $(\sum_{I_j=[d_j,e_j] \in subSet_k} \delta_{e_j}))$
    for $i = 1$ to $roundedToUpperInt(\frac{p}{l})$ do
        // p is the number of processors.
        foreach $subSet_k \in Ls$ in order do
            Assign $subSet_k$ to the non-used processor $P_u$ resulting to biggest cycle-time
            and satisfying the period $\mathcal{P}_{\max}$. The mapping period is computed by
            applying Random, MaxBout, MinComm, MinNbProcs heuristics and
            by retaining the smallest result
            If success, mark $P_u$ as used
        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 set $subSet_k$ with the biggest failure probability and for which
        $P_u$ satisfies the period $\mathcal{P}_{\max}$ (application of Random, MaxBout, MinComm,
        MinNbProcs heuristics)
    end
    Apply Algorithm 10 (fusion) to improve failure probability of the resulted mapping
end
```

Bal: balancing failure probabilities – This heuristic assigns each subset of intervals to a set of most critical processors, i.e., with the longest cycle-time satisfying the period constraint. A set $allocS(subSet_k)$ of processors allocated to a subset $subSet_k$ of intervals is such that the product of processors failure probabilities $\prod_{u \in allocS(subSet_k)} f_u$ approximates the average value $(\sqrt[l]{\prod_{u \in [1..p]} f_u})$ (p is the number of all processors and l the number of all subsets $subSet_k$). When all subsets of intervals are assigned, the heuristic attempts to improve the failure probability of the computed mapping by applying the fusion algorithm (Algorithm 10). The **Bal** heuristic is further detailed in Algorithm 9. 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 subsets of intervals in a decreasing order of their output data size (resp. their workload). The objective is still to provide a mapping priority for costly subsets of intervals.

Algorithm 9: **Bal** heuristic: mapping l given subsets of intervals with balancing their failure probabilities, under a fixed period $\mathcal{P}_{\max}$.

```
begin
    Order input sets $subSet_k$ $(1 \leq k \leq l)$ by decreasing computation load
    $(\sum_{I_j=[d_j,e_j] \in subSet_k} \sum_{i=d_j}^{e_j} w_i)$ in list $Ls$ (or decreasing output data size, i.e.,
    $(\sum_{I_j=[d_j,e_j] \in subSet_k} \delta_{e_j}))$
    foreach $subSet_k \in Ls$ in order do
        Assign $subSet_k$ to a set of non-used processors $procs$ of $P_u (1 \leq u \leq p)$ with
        $\prod_{u \in allocS(subSet_k)} f_u \approx (\sqrt[l]{\prod_{u \in [1..p]} f_u})$ and which result to the largest
        cycle-times satisfying the period $\mathcal{P}_{\max}$. The mapping period is computed by
        applying Random, MaxBout, MinComm, MinNbProcs heuristics and by
        retaining the smallest result
        If success, 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 set $subSet_k$ with the highest failure probability and for
        which the period $\mathcal{P}_{\max}$ is satisfied (application of Random, MaxBout,
        MinComm, MinNbProcs heuristics)
    end
    Apply Algorithm 10 (fusion) to improve failure probability of the resulted mapping
end
```

Algorithm 10: Fusion of subsets of intervals for a given mapping with initially l subsets to decrease failure probability $\mathcal{F}$, under a fixed period $\mathcal{P}_{\max}$.

```
begin
    while it is possible to decrease $\mathcal{F}$ and there are at least 2 subsets of intervals do
        // Step 1
        Find $subSet_k$ $(1 \leq k \leq l)$ in the current mapping with the highest failure
        probability
        // Step 2
        repeat
            Fusion $subSet_k$ with any non tested $subSet_{k'}$ $(subSet_k \neq subSet_{k'})$
            Discard processors among those initially assigned to $subSet_k$ and $subSet_{k'}$
            and non-used ones that do not satisfy the period $\mathcal{P}_{\max}$ (application of
            Random, MaxBout, MinComm, MinNbProcs heuristics) after fusion
            Discarded processors become non-used
            Mark $subSet_{k'}$ as tested for fusion with $subSet_k$
            if the fusion decreases the failure probability of the initial mapping then
                Retain the new resulting mapping
            end
            else
                Ignore the fusion
                Mark $subSet_{k'}$ as tested
            end
        until the fusion decreases the failure probability OR there is another non tested
        subset
    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 set $subSet_k$ with the biggest failure probability and for which
        $P_u$ satisfies the period $\mathcal{P}_{\max}$ (application of Random, MaxBout, MinComm,
        MinNbProcs heuristics)
    end
end
```

6.2.3 Partitioning stages then mapping heuristics

Once interval subsets for a pipelined application have been created according to a partitioning criterion, they are mapped by using one of the mapping heuristics. With three partitioning criteria and four proposed mapping strategies, each one with eventually three variants, we obtain a total of 21 heuristics. Algorithm 11 details the **PartStr-Snake-c** heuristic. The other heuristics work in a similar way, with different variants.

Algorithm 11: **PartStr-Snake-c** heuristic: computing a general mapping optimizing $\mathcal{F}$, under a fixed period $\mathcal{P}_{\max}$.

```
begin
    Initialize the failure probability of the application $\mathcal{F}$ to 1
    for $l = 1$ to $\min(n, p)$ do
        // Step 1: create subsets of intervals according to a random affectation.
        Initialize $l$ subsets of intervals to empty
        repeat
            foreach $subSet_k$ $(1 \leq k \leq l)$ do
                Add a stage randomly chosen among those not yet affected to a subset
            end
        until all stages are affected to a subset
        // Step 2: compute a mapping for the created subsets of intervals.
        Apply Algorithm 7 on the $l$ created subsets ordered according to their output
        data size
        Compute the failure probability $\mathcal{F}_t$ of the resulted mapping (if a subset is not
        assigned to any processor, 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"
    end
    Return a mapping solution among the $\min(n, p)$ computed ones with the final
    failure probability $\mathcal{F}$
    return "success"
end
```

6.3 Mapping heuristics, class 2: partitioning processors then mapping

Differently from the previous class, heuristics in class 2 start by partitioning the available processors into disjoint sets. Then, stages are assigned to these sets in such a way the period bound is satisfied. The objective of partitioning processors before mapping is to group processors according to close speeds that may respond to performance requirement for executing a same interval subset. Different partitions of processors are explored and the mapping resulting to the smallest failure probability is retained.

6.3.1 Partitioning processors

This partitioning phase returns different partitions of the platform processors $P_1, ..., P_p$. Each partition groups a set of processors with close speeds and such that the failure probability of each set approximates the average failure probability of the p processors. In more details, processors are handled in decreasing (or increasing) order of their speeds and then, are split into q sets, with $1 \leq q \leq \min(n, p)$. Each set has a failure probability close to $(\sqrt[q]{\prod_{u \in [1..p]} f_u})$.

This partitioning solution aims at enabling maximum processor speed exploitation when sets of stages will be assigned, while promoting the balancing of failure probabilities for final interval subsets.

6.3.2 Partitioning processors then mapping

Mapping heuristics follow a same principle. Each heuristic repeatedly attempts to assign a stage S_i $(1 \leq i \leq n)$ to a set of processors created by the partition phase. The chosen processor set is the one achieving the smallest period and such that the bound $P_{\max}$ is satisfied. Once all stages are treated, mapped interval subsets are resulted. Heuristics attempt to improve the failure probability of derived mapping. For that, they repeatedly try to allocate the most reliable processor among possibly not assigned ones to the interval subsets with the biggest failure probability. They also apply the fusion process as done by heuristics in class 1 (Algorithm 10). Finally, different partitions of processors are explored and the mapping reaching to the smallest failure probability is returned.

The difference between heuristics in this class resides in the mapping order of stages. In the present work, we define two heuristics according to two orders. The first order gives a mapping priority to stages with biggest output data size. Associated heuristic is named **PartPrc**. The second order gives the priority to stages with biggest workload (w_i). Associated heuristic is named **PartPrw**. Without lost of generality, Algorithm 12 details the principle of **PartPrw** heuristic.

Algorithm 12: **PartPrw** heuristic: computing a general mapping optimizing $\mathcal{F}$, under a fixed period $\mathcal{P}_{\max}$.

```
begin
    Sort processors $P_1, ..., P_p$ by decreasing order of their speed in list $Lp$
    Sort stages $S_1, ..., S_n$ by decreasing order of their workload ($w_i$) in list $Ls$
    Initialize the failure probability of the application $\mathcal{F}$ to 1
    for $q = 1$ to $\min(n, p)$ do
        Initialize sets $SetP_r$ ($1 \leq r \leq q$) of processors to empty
        // Distribute processors over $SetP_r$ sets.
        foreach $SetP_r$ do
            repeat
                Add the first processor in $Lp$ that is not yet affected to a set
            until $SetP_r$ verifies $\prod_{u \in SetP_r} f_u \approx (\sqrt[q]{\prod_{u \in [1..p]} f_u})$
        end
        // Assign stages to $SetP_r$ sets.
        foreach $S_i \in Ls$ in order do
            Assign $S_i$ to the set $SetP_r$ resulting to the smallest period satisfying $\mathcal{P}_{\max}$.
            The period is computed by applying Random, MaxBout, MinComm,
            MinNbProcs heuristics and by retaining the smallest result
        end
        // At this point, some processor sets may be allocated no stage.
        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 set $subSet_k$ with the biggest failure probability and for
            which $P_u$ satisfies the period $\mathcal{P}_{\max}$ (application of Random, MaxBout,
            MinComm, MinNbProcs heuristics)
        end
        // Try to improve the current mapping.
        Apply Algorithm 10 (fusion) to improve failure probability of the resulted
        mapping
        // Compute current $\mathcal{F}$.
        Compute the failure probability $\mathcal{F}_t$ of the resulted mapping (if some stages are
        not assigned to any processor, set $\mathcal{F}_t$ to 1)
        Accept the mapping with $\min(\mathcal{F}_t, \mathcal{F})$ and set $\mathcal{F}$ to this value
    end
    // Return a mapping solution.
    if $\mathcal{F}_t = 1$ then
        return "failure"
    end
    Return a mapping solution among the $\min(n, p)$ computed ones with the final
    failure probability $\mathcal{F}$
    return "success"
end
```

7 Experiments

This section reports experimental results assessing the performance of the heuristics. We first focus on heuristics solving the CONSENSUS problem. Then, we deal with mapping heuristics. All the heuristics have been developed using C/C++ and the gcc compiler version 4.3.2. The reader can find the corresponding source code at:
http://graal.ens-lyon.fr/~hbouzian/code/gen-FT-FullHet.tgz.

7.1 Heuristics computing the period vs linear program

To measure the performance of the **Random**, **MaxBout**, **MinComm** and **MinNbProcs** heuristics, we have simulated consensus scenarios on randomly generated mappings (with randomly generated applications and heterogeneous platforms). We compare the results with the optimal period obtained by the mixed integer linear program presented in Section 5. For implementing and executing this program, we used the CPLEX Interactive Optimizer version 11.2.0.

In more details, we have simulated scenarios for applications with n stages, where n is varying from 2 to 120, and platforms with $10, 30$ and 100 processors. The workload w_i of stages have been set to double values chosen in interval $[1, 20]$ and input/output data sizes δ_i to integer values in $[1, 25]$. For processors, the speed (s_u) has been set to double values belonging to interval $[1, 20]$, and the input/output network card capacity (B_u^i, B_u^o) to double values chosen in $[1, 10]$, like the links bandwidths $(b_{u,v})$. Last, to generate a mapping, we have randomly chosen a placement for each stage on a randomly chosen set of processors. Experiments have been executed on a 32-bit Pentium 1.6 GHz processor with $1GB$ of RAM.

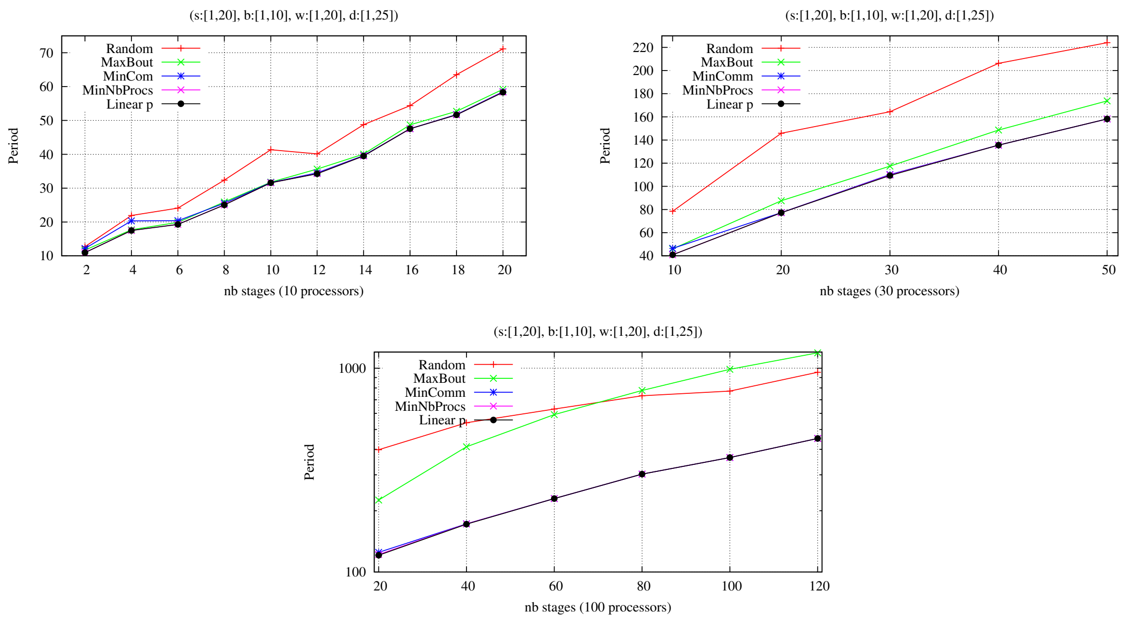

Figure 7: Heuristics vs linear program to solve the CONSENSUS problem. Comparison of the average behavior for randomly generated mappings on *FullHet* platforms.

Figure 7 shows the average behavior of the heuristics and the linear program when the size of applications and platforms varies. The relative deviation of the periods returned by

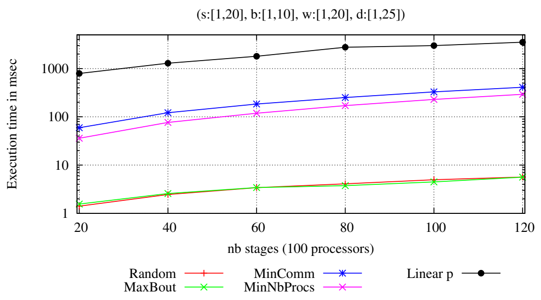

Figure 8: Average execution duration of the linear program and heuristics for the experiment done on 100 processors (Figure 7).

heuristics compared to the optimal result, for all simulated scenarios, are also reported in Table 1. We see that **Random** and **MaxBout** diverge from the optimal solution when the processor number increases. However, heuristics **MinComm** and **MinNbProcs** are still close to the optimal solution (with a small difference). The good performance of these latter two heuristics shows the relevance of considering the costs of outgoing communications (both emission and transfer times). Attempting to minimize costly communications, and electing more than one surviving processor per assigned subset of intervals, proves very effective in reducing the cycle time of a processor, thus the period.

We have also measured the execution times of heuristics and linear program, in particular, for large simulated platforms ($p = 100$) and for applications with $n = 20, 40, ...120$ (Figure 8). We have observed that the linear program is on average between 527 and 627 times slower than the fastest heuristics **MaxBout** and **Random**, while it is between 9 and 13 (resp. 12 and 22) times slower than **MinComm** (resp. **MinNbProcs**) heuristics. These latter heuristics require more complex operations (browsing multiple tables) in the implementation for processor election.

Finally, we conclude that we can reach satisfying results to the consensus problem with small execution costs, thanks to the proposed heuristics. For the rest of the experiments, we execute the previous four heuristics and retain the smallest result to estimate the period of a given mapping.

7.2 *General* vs *interval* mapping heuristics

The second set of experiments aims at evaluating the performance of general mapping heuristics proposed in Section 6.2 and Section 6.3. This evaluation is done through the simulation of several mapping scenarios for applications, and for heterogeneous platforms with different sizes. The performance of the heuristics is expressed by their ability to achieve a small failure probability, under a period bound $\mathcal{P}_{\max}$. We compare the results with those obtained by *interval* mapping heuristics presented in [3].

In more details, we have generated applications with 16 and 24 stages. The workload of these stages has been set to double values randomly chosen in interval $[1, 100]$ and input/output data sizes to integer values in $[1, 5]$. The mapping is determined on platforms with 16, 32 and 64 processors. Processor speeds have been set to double values randomly chosen in interval $[10, 20]$, the input/output network card capacity and the links bandwidths to double values chosen in $[1, 10]$. The failure probabilities have been randomly generated

		max	av.	stdv.	best rate
$p = 10$ 500 scenarios	**Random**	9.082	0.342	0.873	62.20%
	MaxBout	3.903	0.049	0.225	86.80%
	MinComm	3.196	0.049	0.279	92.80%
	MinNbProcs	0.217	0.001	0.015	99.00%
	Linear p	0.000	0.000	0.000	100.00 %
$p = 30$ 250 scenarios	**Random**	6.157	0.768	1.068	26.00%
	MaxBout	1.641	0.138	0.253	57.60%
	MinComm	3.674	0.032	0.263	95.20%
	MinNbProcs	0.209	0.001	0.013	99.60%
	Linear p	0.000	0.000	0.000	100.00 %
$p = 100$ 600 scenarios	**Random**	11.861	1.769	1.501	1.17%
	MaxBout	4.870	1.573	0.800	2.50%
	MinComm	0.631	0.007	0.047	94.83%
	MinNbProcs	0.208	0.002	0.015	95.67%
	Linear p	0.000	0.000	0.000	100.00 %

Table 1: Heuristics vs linear program to solve the CONSENSUS problem (maximum, average and standard relative deviation of the period for 1350 consensus instances).

between 0.05 and 0.3. Lastly, we have selected different period bounds varying between 1.5 and 12.5. The simulations have been executed on four machines: a quad-processor machine (64-bit AMD Opteron at 2.2GHz) with 32 GB of RAM, two quad-processor machines (64-bit AMD Opteron at 2.3GHz) with 32 GB of RAM and a quad-processor machine (64-bit AMD Opteron at 2.4GHz) with 80 GB of RAM.

Results over different configurations: Figures 9, 10, 11, 12 and 13 compare the variants of heuristics using *general* mapping and *interval* mapping models. There are six pairs of plots in each Figure. Each pair reports the average results obtained for a set of given configuration of applications and platforms and for different period bounds. It compare the average failure probabilities obtained by *interval* (on the left) and *general* mapping (on the right) heuristics using a similar mapping approach (except for the heuristics belonging to class 2). Recall that *general* mapping heuristics in class 1 extend previously designed *interval* mapping heuristics [3]. While in class 2, different mapping strategies are adopted, even if they both rely on progressive partitioning and mapping.

From each figure, we first observe that *general* mapping heuristics can behave better when the number of processors increases. In fact, in such a situation, enabling non consecutive intervals to be mapped on a larger set of processors have a more relevant impact on decreasing the final failure probability. In addition, the improvement resulted by *general* mapping heuristics appears when the period bound is not too small (failure probability close to 1) nor too large (failure probability close to 0). In fact, a small period bound can easily lead to costly intervals (in term of workloads) or subsets of intervals (in terms of workloads and communications). Therefore, both *interval* and *general* mapping models may fail to find a solution. The opposite situation appears for large period bounds, where all heuristics may reach very small failure probabilities. However, when the number of processors increases, *general* mapping heuristics are able to reach such results earlier (according to the period bound evolution) than *interval* mapping heuristics.

Summary: Table 2 sums up the behavior of *interval* and *general* mapping heuristics, over all conducted experiments (5650 scenarios). For each *interval* (respectively *general*) mapping heuristic, the table represents the relative failure probability compared to the smallest prob-

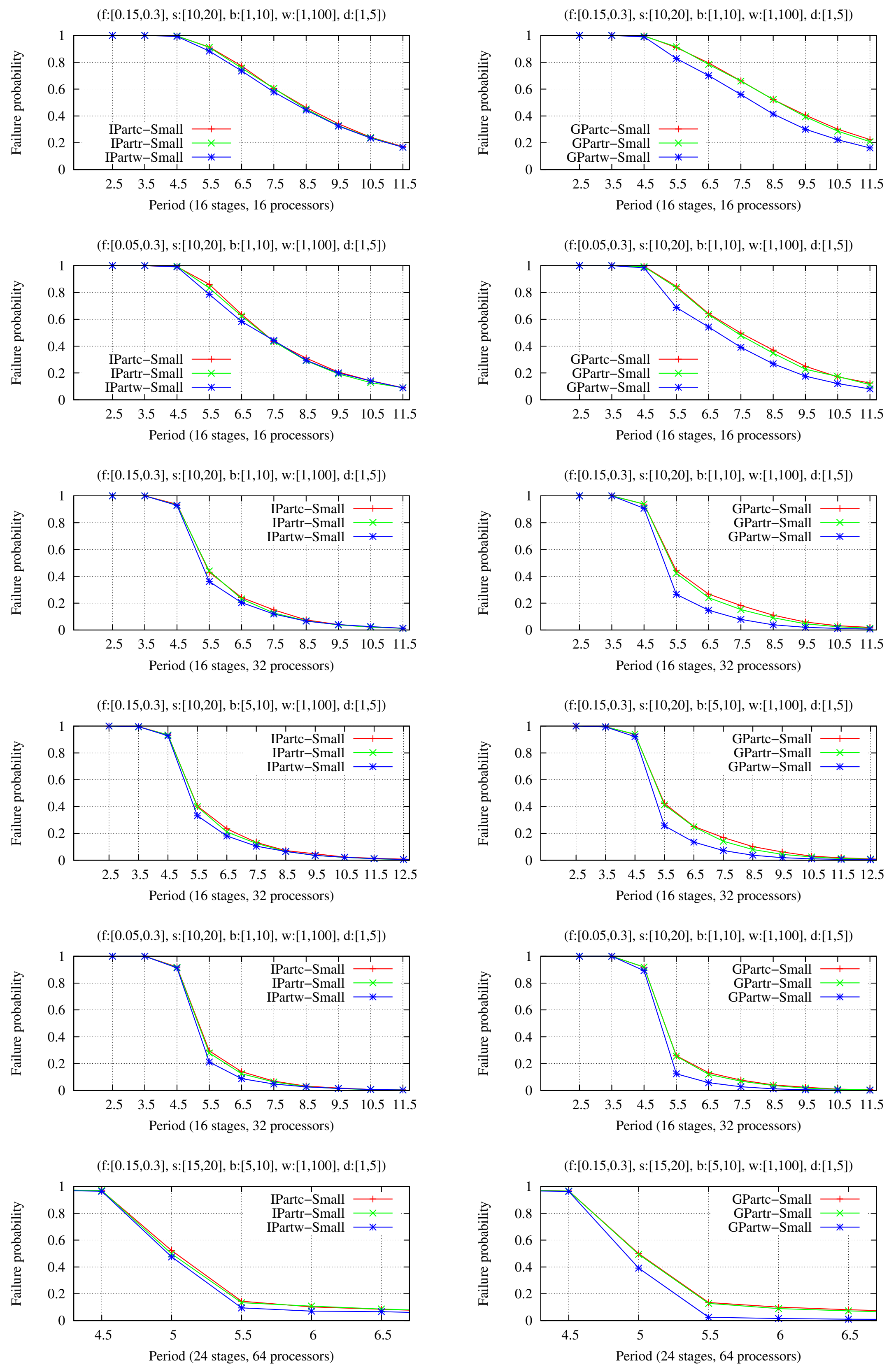

Figure 9: Comparison of {**PartStc**|**PartStr**|**PartStw**}-**Small** heuristic variants on *FullHet* platforms. On the left column (respectively the right column), the results for *interval* mapping (resp. *general* mapping)

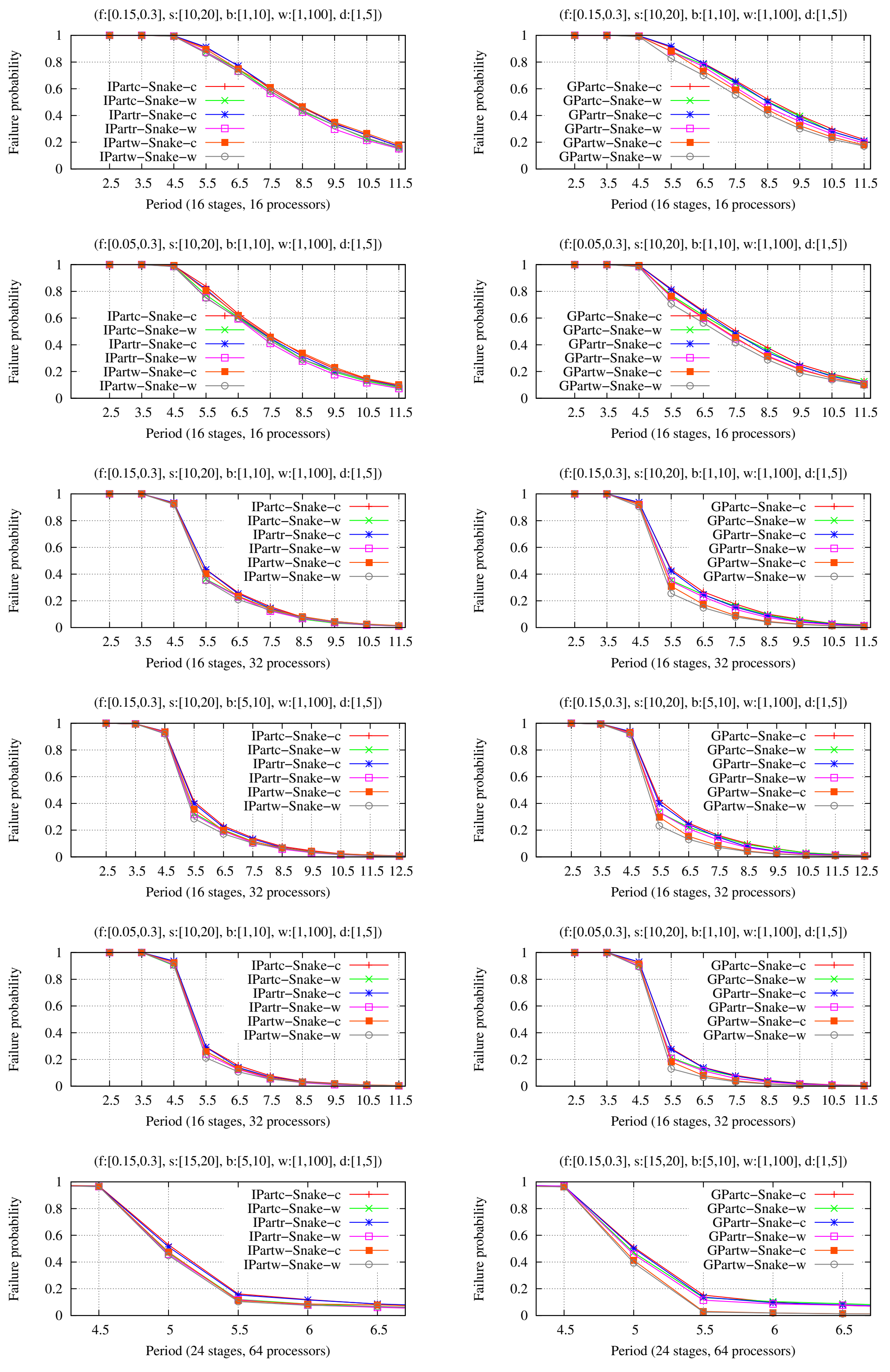

Figure 10: Comparison of {**PartStc**|**PartStr**|**PartStw**}-**Snake**-{c|w} heuristic variants on *FullHet* platforms. On the left column (respectively the right column), the results for *interval* mapping (resp. *general* mapping)

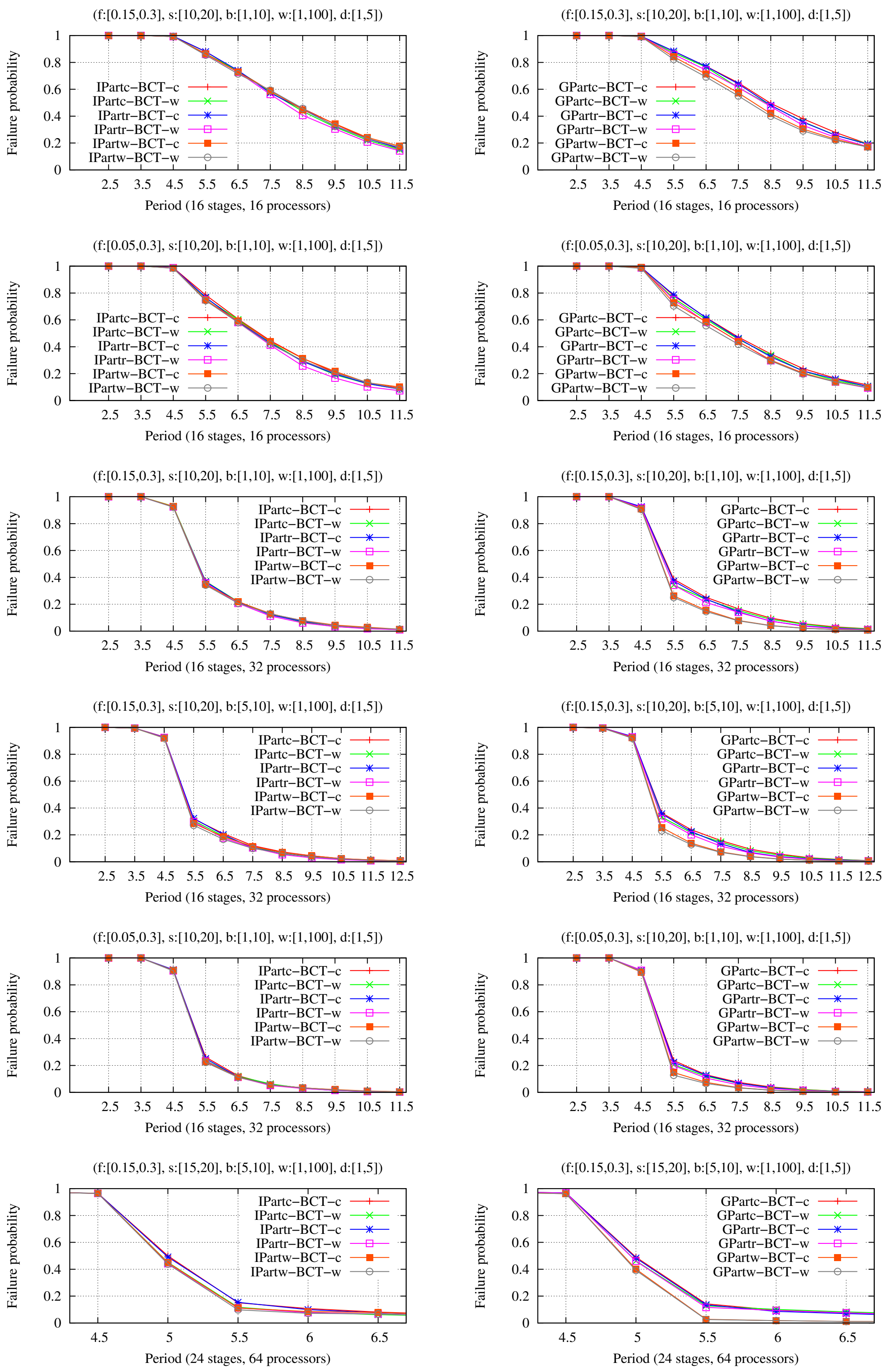

Figure 11: Comparison of {**PartStc**|**PartStr**|**PartStw**}-**BCT**-{c|w} heuristic variants on *FullHet* platforms. On the left column (respectively the right column), the results for *interval* mapping (resp. *general* mapping)

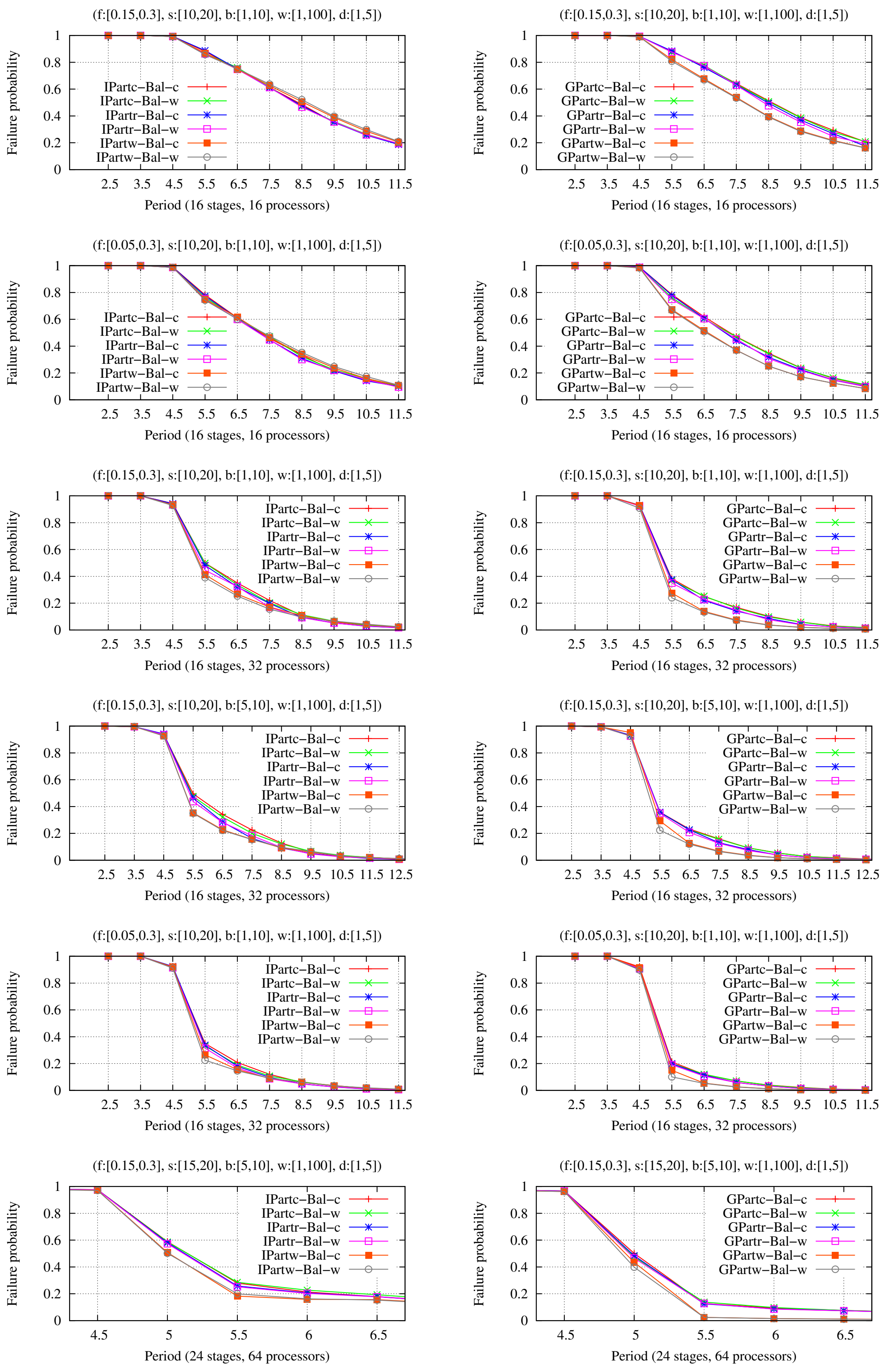

Figure 12: Comparison of {**PartStc**|**PartStr**|**PartStw**}-**Bal**-{c|w} heuristic variants on *FullHet* platforms. On the left column (respectively the right column), the results for *interval* mapping (resp. *general* mapping)

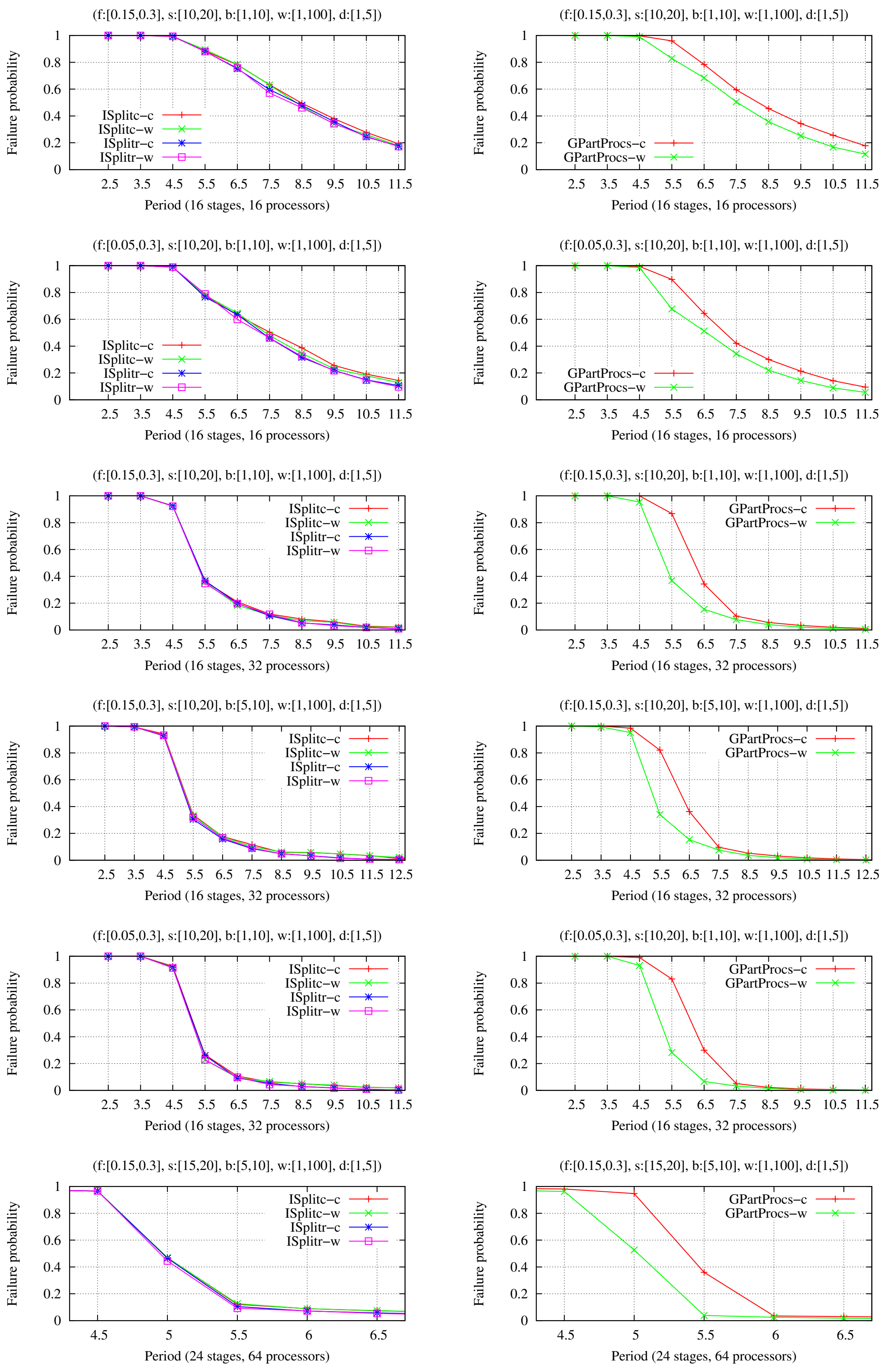

Figure 13: Comparison of {**Splitc**|**Splitr**}-{**c**|**w**} (for *interval* mapping) heuristic and **PartPrc**, **PartPrw** (for *general* mapping) on *FullHet* platforms.

	Interval mappings				*General* mappings			
	max	av.	stdv.	best rate	max	av.	stdv.	best rate
PartStc-Small	925	2.2	14.8	28.8%	959	5.9	29.0	28.3%
PartStr-Small	135	1.7	5.2	1.6%	1001	4.5	26.6	3.2%
PartStw-Small	113	1.4	3.4	1.8%	65	0.7	2.2	4.8%
PartStc-Snake-c	677	2.3	13.8	1.1%	7269	7.4	101.9	5.5%
PartStc-Snake-w	269	1.7	7.3	2.4%	2830	6.9	56.6	4.9%
PartStr-Snake-c	257	2.1	7.5	1.6%	1924	5.1	37.6	2.6%
PartStr-Snake-w	241	1.4	5.5	2.5%	2804	4.6	43.7	4.0%
PartStw-Snake-c	161	1.9	5.4	1.0%	310	1.2	5.1	2.3%
PartStw-Snake-w	175	1.4	4.8	1.9%	219	1.1	4.2	1.5%
PartStc-BCT-c	2356	2.6	33.2	1.4%	5291	6.9	80.9	3.1%
PartStc-BCT-w	968	2.1	18.4	2.5%	2952	6.9	59.1	3.4%
PartStr-BCT-c	940	1.8	13.4	2.1%	3395	4.9	52.5	5.0%
PartStr-BCT-w	216	1.4	5.9	3.6%	2718	4.7	46.1	4.8%
PartStw-BCT-c	542	1.9	9.4	1.6%	77	1.0	3.1	1.2%
PartStw-BCT-w	234	1.7	6.1	2.5%	76	1.0	3.1	2.7%
PartStc-Bal-c	3539	4.2	48.9	1.4%	3196	7.3	60.4	3.0%
PartStc-Bal-w	1297	4.3	30.6	1.7%	4496	7.3	72.9	3.5%
PartStr-Bal-c	158	2.7	7.4	2.7%	2941	4.7	46.1	4.4%
PartStr-Bal-w	623	3.0	13.6	2.3%	1090	4.3	29.6	4.7%
PartStw-Bal-c	2238	3.9	36.4	2.9%	213	0.9	4.8	7.5%
PartStw-Bal-w	620	3.6	18.1	2.3%	186	0.9	4.1	11.5%
Splitc-c	175115	54.6	2359.5	5.9%	no equivalent			
Splitc-w	65161	46.4	1241.1	4.8%	no equivalent			
Splitr-c	482	1.3	7.5	11.4%	no equivalent			
Splitr-w	289	1.2	6.0	11.8%	no equivalent			
PartPrc	no equivalent				298	2.4	8.5	2.8%
PartPrw	no equivalent				90	0.4	1.7	22.6%

Table 2: Behavior of mapping heuristics across all experiments for *interval* and *general* mappings (maximum, average and standard relative deviation for 5650 mapping instances).

ability reached by an *interval* (respectively a *general*) mapping heuristic for each simulated scenario. From the table, several conclusions may be drawn.

First, for class 1 heuristics, it can be noted that partitioning stages according to the computation cost criterion (prefix **PartStw**) reaches better results when using a *general* mapping model. In fact, when the workload of stages considerably varies, and when $\mathcal{P}_{\max}$ is well chosen, using this criterion allows for a better grouping of less costly stages, possibly not consecutive ones, in the same subset. This situation corresponds to the introductory example in Section 2.

Next we observe that the majority of the heuristics has a much better rate of reaching the smallest failure probability in the *general* mapping context. This is the case even if the rate is quite low. However, the standard deviation does not follow this improvement. This may be explained by the fact that heuristics may well considerably fail to reach a satisfying result (column “max”). For example, the maximum relative value 175115, obtained for **Splitc-c** heuristic, corresponds to a failure probability $\mathcal{F} = 0.2665271$ against a minimum $\mathcal{F} = 0.0000015$ reached by the **PartStc-Bal-w** heuristic. Several complex factors can explain these results. In particular, all the mapping heuristics are based on the knowledge of only partial properties of both applications and platforms. Another difficulty is added to the *general* mapping heuristics, for which the estimation of the minimum worst-case period satisfying the period bound may be less accurate.

Finally, it is difficult to distinguish dominant heuristics. A better approach to solve the present optimization problem seems to use several heuristics, and to retain the best returned

		max	av.	stdv.	best rate
$p = 16$ 2000 scenarios	*Interval*	1.824	0.063	0.170	67.15 %
	General	2.359	0.074	0.183	32.85 %
$p = 32$ 3200 scenarios	*Interval*	27.169	0.298	1.066	57.84 %
	General	6.188	0.152	0.447	42.16 %
$p = 64$ 450 scenarios	*Interval*	17.788	2.193	2.770	27.11 %
	General	10.690	0.030	0.508	72.89 %

Table 3: *Interval* vs *general* mapping: comparison of respective smallest failure probabilities ($\mathcal{F}$), over all experiments (maximum, average and standard relative deviation for 5650 results).

result. Following such an approach, Table 3 reports the results for *interval* and *general* mappings over all experiments. For each model, all proposed heuristics are used. From this table, it can be noted that a *general* mapping is able to achieve dramatically better mapping results on large platforms.

8 Conclusion

This report contributes to the design of efficient scheduling and mapping strategies for pipelined applications, on heterogeneous and failure-prone computational platforms. We have focused on the most relevant bi-criteria optimization problem, namely computing a reliable mapping while guaranteeing a given throughput. The main contribution is the study of *general* mapping solutions, which extends the more limited *interval* mapping class considered in [3].

We have first presented complexity results, that show a significant increase in the difficulty of the problem. In particular, computing the period of a given *general* mapping is NP-complete even in the simplest case of no-failure mappings, while it is polynomial for *interval* mappings in all cases. This led us to introduce polynomial algorithms to approximate the period, and to develop a mixed linear program to compute the optimal solution. Experimental results have shown the efficiency of our heuristics to approach the optimal period for various problems sizes. Building upon these heuristics, we have addressed the bi-criteria period/reliability optimization problem. We have designed polynomial *general* mapping algorithms, and we have assessed, through an extensive set of experiments, their ability to significantly improve the reliability of the solution returned by the best *interval* mapping.

The superiority of *general* mapping heuristics comes with a price in terms of execution time. We typically report a few minutes for medium-size application/platform instances, as opposed to a few seconds for *interval* mapping heuristics. Given the efficiency of the new heuristics, it seems worth investigating more efforts to improve their design and decrease their execution time. It would be also be very interesting to experiment our solutions for real life applications and platforms.

References

[1] I. Assayad, A. Girault, and H. Kalla. A bi-criteria scheduling heuristics for distributed embedded systems under reliability and real-time constraints. In *Int. Conf. on Dependable Systems and Networks, DSN'04*, pages 347–356. IEEE CS Press, 2004.

[2] 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.

[3] A. Benoit, H. L. Bouziane, and Y. Robert. Optimizing the reliability of pipelined applications under throughput constraints. In *The 9th International Symposium on Parallel and Distributed Computing (ISPDC 2010)*. IEEE CS Press, 2010. To appear; available at `graal.ens-lyon.fr/~abenoit`.

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

[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] A. Dogan and F. Özgüner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Trans. Parallel Dist. Systems*, 13(3):308–323, 2002.

[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] A. Girault and H. Kalla. A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate. *IEEE Trans. Dependable Secure Computing*, 6(4):241–254, 2009.

[9] 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.

[10] 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.

[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.