Mapping pipelined applications with replication to increase throughput and reliability

Anne Benoit, Loris Marchal, Yves Robert, Oliver Sinnen

Mapping pipelined applications with replication to increase throughput and reliability

Anne Benoit[1], Loris Marchal[1], Yves Robert[1] and Oliver Sinnen[2]

1. LIP (jointly operated by CNRS, ENS Lyon, INRIA and UCB Lyon), ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France.
{Anne.Benoit|Loris.Marchal|Yves.Robert}@ens-lyon.fr
2. Department of Electrical and Computer Engineering,University of Auckland 1142, Private Bag 92019, Auckland, New Zealand. o.sinnen@auckland.ac.nz

LIP Research Report RR-LIP 2009-28

Abstract

Mapping and scheduling an application onto the processors of a parallel system is a difficult problem. This is true when performance is the only objective, but becomes worse when a second optimization criterion like reliability is involved. In this paper we investigate the problem of mapping an application consisting of several consecutive stages, i.e., a pipeline, onto heterogeneous processors, while considering both the performance, measured as throughput, and the reliability. The mechanism of replication, which refers to the mapping of an application stage onto more than one processor, can be used to increase throughput but also to increase reliability. Finding the right replication trade-off plays a pivotal role for this bi-criteria optimization problem. Our formal model includes heterogeneous processors, both in terms of execution speed as well as in terms of reliability. We study the complexity of the various subproblems and show how a solution can be obtained for the polynomial cases. For the general NP-hard problem, heuristics are presented and experimentally evaluated. We further propose the design of an exact algorithm based on A* state space search which allows us to evaluate the performance of our heuristics for small problem instances.

1 Introduction

Mapping applications onto parallel platforms is a difficult challenge. Several scheduling and load-balancing techniques have been developed for homogeneous architectures (see [15] for a survey) but the advent of heterogeneous clusters has rendered the mapping problem even more difficult. Typically, such clusters are composed of different-speed processors interconnected by some communication network. They constitute a common

platform found in industry and academia, well suited for applications with negligible or low inter-task communication volume.

We focus in this paper on the pipeline skeleton, i.e., an application with linear dependencies between stages, to be executed on such clusters. The pipeline skeleton is typical of streaming applications like video and audio encoding and decoding applications, etc. [6, 18, 10, 19, 20]. Each stage has its own communication and computation requirements: it reads an input file from the previous stage, processes the data and outputs a result to the next stage. For each data set, initial data is input to the first stage, and final results are output from the last stage. The pipeline workflow operates in synchronous mode: after some latency due to the initialization delay, a new task is completed every period.

Key metrics for a given workflow are the throughput and the reliability. The throughput measures the aggregate rate of processing data, and it is the rate at which data sets can enter the system. Equivalently, the inverse of the throughput, defined as the period, is the time interval required between the beginning of the execution of two consecutive data sets. The reliability of the application is the probability that all computations will be successful. The period can be minimized by dealing different data sets to different processors in a round-robin fashion, while reliability is increased by replicating computations on a set of processors (i.e., computing each data set several times). The application fails to be executed only if all processors involved in a same redundant computation fail during execution. In this paper, we focus on bi-criteria approaches, i.e., minimizing the failure probability under period constraints, or the converse. Thus we must investigate the trade-off of using processors to deal data sets and reduce the period, or to replicate computations and increase the reliability. The optimization problem can be stated informally as follows: which stage to assign to which processor?

We target applications and platforms on which the cost of communications is negligible in comparison to the computation cost. *SpeedHom* platforms are made of identical speed processors, while *SpeedHet* platforms consist of different speed processors. In terms of reliability (defined as the inverse of the overall failure probability), we consider either processors with identical failure probabilities, denoted *FailureHom*, or with different failure probabilities *FailureHet*. All four combinations of processor speed and reliability are addressed in this paper.

We require the mapping of the pipeline graph to be interval-based, i.e., a processor is assigned an interval of consecutive stages. A lot of work has been done to investigate the complexity of finding the best interval mapping of such linear graphs, according to diverse optimization criteria, onto homogeneous or heterogeneous platforms. Thus, Subhlok and Vondran prove in [16] that a mapping which minimizes the period on a homogeneous platform can be found in polynomial time. In [17], they extended this approach to bi-criteria problems accounting for the latency of a mapping, i.e., the maximum time required to process one single data set. This work was extended to heterogeneous platforms [4].

The use of stage replication techniques has also been widely studied. On the one hand, some *deal* replication is used to reduce the period of a mapping, see for instance the Data Cutter project [6], or a theoretical study in [3]. On the other hand, active replication to increase reliability is also a classical technique used to overcome processor failures: in [11, 9], the replication is such than a given number of processor failures is supported. In [2], we defined the reliability of a pipelined application similarly than in the present

paper, and used replication to decrease the failure probability of an application.

To the best of our knowledge, this paper presents the first study of problems in which both replication for performance and replication for reliability are used simultaneously, and thus considering the two antagonist criteria, throughput and reliability. First we formally detail the model and the optimization problems in Section 2. Then in Section 3 we establish the complexity of all problem instances, and derive some new approximation results. Bi-criteria problems are NP-hard in most cases because of the antagonistic criteria. In particular, it is NP-hard to minimize the failure probability, given a bound on the period of the application, when targeting heterogeneous platforms. Thus, we introduce polynomial time heuristics in Section 4 to propose practical solutions to this problem. One contribution is the design of an exact algorithm based on A* (see Section 5), which allows us to evaluate the absolute performance of our heuristics on small problem instances in Section 6. Finally, we state some concluding remarks in Section 7.

2 Framework

We outline in this section the characteristics of the applicative framework, as well as the model for the target platform. Next we detail the bi-criteria optimization problem.

2.1 Applicative framework

We consider a pipeline of n stages $\mathcal{S}_i$, $1 \leq i \leq n$, as illustrated on Fig. 1. Input data is fed into the pipeline and processed from stage to stage, until output data exits the pipeline after the last stage.

The i-th stage $\mathcal{S}_i$ receives an input from the previous stage, performs a number of w_i computations, and outputs data to the next stage. Communication costs are assumed to be negligible in comparison with the computation costs, thus the time required to compute one data set for stage $\mathcal{S}_i$ is proportional to w_i.

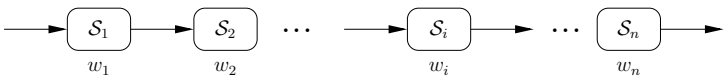


Figure 1: The application pipeline.

2.2 Target platform

We target a platform with p processors P_u, $1 \leq u \leq p$, fully interconnected as a (virtual) clique. The speed of processor P_u is denoted as s_u, and it takes X/s_u time-units for P_u to execute X floating point operations. We consider either *SpeedHom* platforms made of identical processors: $s_u = s$ for $1 \leq u \leq p$, or general *SpeedHet* platforms. Moreover, we assume that communications are negligible compared to computation costs, thus we do not characterize here the network bandwidths and communication times.

In order to evaluate and optimize the reliability of the mappings produced, we associate a failure probability $0 < f_u < 1$ with each processor P_u, $1 \leq u \leq p$. It is the

probability that P_u fails during the execution of the application. We consider constant failure probabilities as we are dealing with pipelined applications. These applications are meant to run during a very long time, and therefore we address the question of whether the processor will fail or not at any time during execution. It might seem odd that this probability is completely independent of the execution time, since one may believe that the longer a processor executes, the larger the chance that it fails. However, we target a steady-state execution of the application, for instance in a scenario in which we would loan/rent resources. Computers could be suddenly reclaimed by their owners, as during an episode of *cycle-stealing* [1, 5, 12]. The failure probability should thus be seen as a global indicator of the reliability of a processor. Also note that we consider in this work only fail-silent (a faulty processor does not produce any output) and fail-stop (no processor recovery) processor failures. We do not consider link failures since in a grid framework, a different path can be found to overcome such failures (and we consider communication costs as negligible).

A platform composed of processors with identical failure probabilities is denoted *FailureHom* and otherwise *FailureHet*. Note that it seems natural to consider *FailureHet* platforms when processors have different speeds, thus for *SpeedHet* platforms, while *SpeedHom* platforms are more likely to be *FailureHom*. However, we consider all combinations in the complexity study in Section 3.

2.3 Mapping problem

The general mapping problem consists of assigning application stages to platform processors. Instead of only considering the simple, but restrictive mapping of one stage per processor, we consider so called interval mapping, where several consecutive stages $\mathcal{S}_i, \mathcal{S}_i+1, \ldots, \mathcal{S}_j-1, \mathcal{S}_j$, i.e., an interval, are mapped together. Such mappings have been extensively studied, see [16, 17, 4, 3].

The cost model associated with interval mappings is the following. We search for a partition of $[1..n]$ into $m \leq p$ intervals $I_j = [d_j, e_j]$ such that $d_j \leq e_j$ for $1 \leq j \leq m$, $d_1 = 1$, $d_{j+1} = e_j + 1$ for $1 \leq j \leq m-1$ and $e_m = n$. Interval I_j is mapped onto a set of processors, A_j. These processors are organized into l_j teams ($l_j \leq |A_j|$), and each team is in charge of one round of the deal. We stress that processors within a team perform redundant computations, while different teams assigned to the same interval execute distinct data sets in a round-robin fashion. Note that we could envision that faster teams would perform more computations, i.e., operate on more data sets, than slower ones. However, enforcing a round-robin execution is the key to preventing out-of-order execution, thereby allowing for a reasonably simple orchestration of all operations. We assume that a processor cannot participate in two different teams, thus for all $1 \leq j, j' \leq m$, $A_j \cap A_{j'} = \emptyset$. The period is then expressed as:

$$\mathcal{P} = \max_{1 \leq j \leq m} \left\{ \frac{\sum_{i=d_j}^{e_j} w_i}{l_j \times \min_{P_u \in A_j} s_u} \right\} \quad (1)$$

Indeed, for each interval, the computation is slowed down by the slowest processor enrolled for this interval (because data sets are distributed in round robin to each team), but the period of the slowest processor is divided by l_j, since this processor gets only one data set every over l_j ones.

Given a partition of processors into l teams T_k, where $l = \sum_{j=1}^{m} l_j$ and $1 \leq k \leq l$, the failure probability of the application is computed as follows, since the computation is successful if at least one processor per team does not break down during execution:

$$\mathcal{F} = 1 - \prod_{1 \leq k \leq l} (1 - \prod_{u \in T_k} f_u) \tag{2}$$

The optimization problem is to determine the best interval mapping, over all possible partitions into intervals, and over all processor assignments. The objective can be to minimize either the period, or the failure probability, or a combination: given a threshold period, what is the minimum failure probability that can be achieved? and the counterpart: given a threshold failure probability, what is the minimum period that can be achieved?

3 Complexity Results

3.1 Mono-criterion problems

All mono-criterion problems have been studied previously, but we recall the results in this section since they introduce the bi-criteria complexity results.

Failure probability [2]. Finding the interval mapping which minimizes the failure probability $\mathcal{F}$ is easy on any kind of platforms: the value of Equation (2) is minimized with $l = 1$ and $T_1 = \{P_1, \ldots, P_p\}$. Thus, all stages are grouped as a single interval, and one single team consisting of all p processors is processing this interval.

Period [3]. In order to minimize the period, we first note that there is no need to add several processors in a team since these extra processors only aim at increasing the reliability.

On a *SpeedHom* platform, $\mathcal{P}$ is minimized by grouping all stages as a single interval and having p teams working on this interval, with one single processor per team. Since all speeds are identical and equal to s, in Equation (1), we have $\min s_u = s$ thus no computational power is lost due to differences in processor speeds.

However, on *SpeedHet* platforms, if two processors of different speeds process the same interval, then the slowest one becomes the bottleneck and the other one is not fully used. In this case, we derived a sophisticated polynomial time algorithm for homogeneous applications with $w_i = w$ for $1 \leq i \leq n$. The idea consists in first proving that there exists an optimal solution which replicates intervals of stages onto intervals of consecutive speed processors. Then, a binary search and a dynamic programming approach are combined to compute which intervals of processors should be used onto which stages (see [3]). The problem turns out NP-hard in the general case of heterogeneous pipelines, since it is no longer possible to consider only intervals of processors. To establish the completeness, we use a reduction from NUMERICAL 3D MATCHING [8], which is NP-complete in the strong sense (see [3]).

3.2 Bi-criteria problems

To the best of our knowledge, it is the first time that the bi-criteria problem considering period and failure probability minimization is studied. Since period minimization is NP-

hard on a *SpeedHet* platform, all bi-criteria problems are NP-hard on such platforms. First we derive complexity results for *SpeedHom* platforms, and we provide a polynomial time algorithm for *FailureHom* platforms. Then we prove that the problem becomes NP-hard when considering *FailureHet* platforms. Approximation results are provided in Section 3.3.

Preliminary result for *SpeedHom* platforms. We prove that there is an optimal solution which consists of a single interval of stages if we restrict to *SpeedHom* platforms with identical speed processors.

Lemma 1. *For SpeedHom platforms, there exists a mapping of the pipeline as a single interval of stages which minimizes the failure probability (resp. period) under a fixed period (resp. failure probability) threshold.*

Proof. The proof works by contradiction. Suppose that there is an optimal mapping which consists of several interval of stages $I_1, \ldots, I_m$. Processors are partitioned into l teams $T_1, \ldots, T_l$, and we use the notations introduced in Section 2.3. The period obtained by this solution is denoted $\mathcal{P}_{opt}$, and for $1 \leq j \leq m$, we have $\sum_{i=d_j}^{e_j} w_i \leq \mathcal{P}_{opt} \times l_j \times s$.

We build a new solution consisting of a single interval of stages which are processed by the same l teams as in the optimal solution. According to Equation (2), the value of the failure probability remains the same, since the teams are identical. The period of the new solution is $\mathcal{P} = \frac{\sum_{i=1}^{n} w_i}{l \times s}$. If $\mathcal{P} > \mathcal{P}_{opt}$, then for $1 \leq j \leq m$, $\sum_{i=d_j}^{e_j} w_i < \mathcal{P} \times l_j \times s$, and if we sum this equation for all j, we obtain $\sum_{i=1}^{n} w_i < \mathcal{P} \times l \times s = \sum_{i=1}^{n} w_i$. Thus, $\mathcal{P} \leq \mathcal{P}_{opt}$, and the new solution consisting of a single interval of stages is optimal. □

Note that this result does not hold on *SpeedHet* platforms. Consider for instance the following problem with two stages and two processors such that $w_1 = s_1 = 1$ and $w_2 = s_2 = 10$. The threshold on the failure probability is fixed to 1, thus it cannot be violated. In this case, we can obtain a period 1 with two intervals while we cannot do better than 11/10 with a single interval.

With identical failure probabilities. Building upon Lemma 1, we are able to derive a polynomial algorithm to solve the bi-criteria problem when further restricting to *FailureHom* platforms.

Theorem 1. *For SpeedHom-FailureHom platforms, the optimal mapping which minimizes the failure probability under a fixed period threshold can be determined in polynomial time $O(p)$; the optimal mapping which minimizes the period under a fixed failure probability threshold can be determined in polynomial time $O(p \log(p))$.*

Proof. Consider first the case with a fixed value of the period, $\mathcal{P}^*$. According to Lemma 1, we restrict the search to solutions consisting in a single interval of stages. We can thus compute the minimum number of teams required to achieve the threshold period: $l_{min} = \left\lceil \frac{\sum_{i=1}^{n} w_i}{\mathcal{P}^* \times s} \right\rceil$.

According to Equation (2), a higher value of l can only increase the failure probability: if $l > l_{min}$ teams, we can obtain a lower failure probability be removing $l - l_{min}$ teams. Moreover, the minimum of the function is reached by greedily assigning all processors to teams in order to have balanced teams: $\forall 1 \leq k, k' \leq l$, $-1 \leq |T_k| - |T_{k'}| \leq 1$. To see this, assume that the optimal solution involves two teams with x and $y \geq x + 2$ processors

respectively. Using two teams of $x+1$ and $y-1$ processors leads to a smaller failure probability. Indeed,
$(1-f^{x+1})(1-f^{y-1}) > (1-f^x)(1-f^y) \Leftrightarrow f^{x+1}+f^{y-1} < f^x+f^y \Leftrightarrow f^{y-x-1}(1-f) < 1-f$.
The processor assignment can thus be done in $O(p)$, where p is the number of processors.

For the converse problem, we cannot easily derive the optimal number of teams from the failure probability threshold $\mathcal{F}^*$. Rather, we try all p possible number of teams. With $1 \leq l \leq p$ teams, the period is $\mathcal{P} = \frac{\sum_{i=1}^{n} w_i}{l \times s}$, which is increasing when l decreases. Starting with $l = p$, we use the previous algorithm to compute the corresponding failure probability, and check if this failure probability is lower than $\mathcal{F}^*$. We stop as soon as we get a failure probability under the threshold. The overall complexity is thus $O(p^2)$ since we may run p times the previous algorithm which works itself in $O(p)$. We could also perform a binary search on the number of teams, thus reducing this complexity to $O(p \log(p))$. □

With different failure probabilities. In the previous algorithm for *FailureHom* platforms, we exploit the fact that balancing the number of processors into teams minimizes the failure probability. This property does not hold with different failure probabilities. Indeed, we prove that the problem becomes NP-had in this case, even though Lemma 1 still holds.

Theorem 2. *For SpeedHom-FailureHet platforms, the problem of finding a mapping which respects both a fixed threshold period and a fixed threshold failure probability is NP-complete.*

Proof. The problem clearly belongs to the class NP: given a solution, it is easy to verify that it is an interval mapping, that the teams are partitioning the set of processors, and to compute its period and reliability with Equations (1) and (2) in polynomial time.

To establish the completeness, we use a reduction from 3-PARTITION, which is NP-complete in the strong sense [8]. We consider an arbitrary instance $\mathcal{I}_1$ of SP: given $3n$ positive integer numbers $\{a_1, \ldots, a_{3n}\}$ and a bound B, assuming that $\frac{B}{4} < a_i < \frac{B}{2}$ for all i and that $\sum_{i=1}^{3n} a_i = nB$, are there n subsets $I_1, I_2, \ldots, I_n$ such that $I_1 \cup I_2 \ldots \cup I_n = \{1, 2, \ldots, 3n\}$, $I_j \cap I_{j'} = \emptyset$ if $j \neq j'$, and $\sum_{i \in I_j} a_i = B$ for all j (and $|I_j| = 3$ for all j). Because 3-PARTITION is NP-complete in the strong sense, we can encode the $3n$ numbers a_i in unary and assume that the size of $\mathcal{I}_1$ is $O(n+M)$, where $M = \max_i\{a_i\}$ (or equivalently $O(n+B)$).

We build the following instance $\mathcal{I}_2$ of our problem: the application has only one pipeline stage with $w = n$, and the *SpeedHom-FailureHet* platform consists of $3n$ processors with speeds $s_u = 1$ and failure probabilities $f_u = 2^{-a_u}$, for $1 \leq u \leq 3n$. The threshold period is fixed to $\mathcal{P} = 1$, and the threshold reliability is fixed to $R = (1-2^{-B})^n$. We ask whether there exists a mapping which respects both thresholds. The size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$: the f_u can be encoded in binary with $O(M)$ bits, and the bound R can be encoded in binary with $O(nB)$ bits.

We now 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, $I_1, \ldots, I_n$. We build the following solution for $\mathcal{I}_2$: n teams are working on the unique stage, with processors in team T_j being the three processors P_u such that $u \in I_j$, for $1 \leq j \leq n$. Since there are n teams, the period of this mapping is $n/n = 1 = \mathcal{P}$, and the reliability is:

$$\prod_{j=1}^{n}(1-\prod_{u\in T_j} f_u) = \prod_{j=1}^{n}(1-2^{-\sum_{u\in T_j} a_u}) = \prod_{j=1}^{n}(1-2^{-B}) = R.$$

Thus we have a solution to $\mathcal{I}_2$.

Conversely, if $\mathcal{I}_2$ has a solution with l teams $T_1, \ldots, T_l$, then the period is $n/l \leq \mathcal{P} = 1$, and so $l \geq n$. If $l > n$, we can build a solution with a period 1 and a higher reliability by removing $l - n$ teams, so we consider in the following that $l = n$. The reliability is $\prod_{j=1}^{n}(1-2^{-\sum_{u\in T_j} a_u})$. This quantity is always strictly larger than $\prod_{j=1}^{n}(1-2^{-B})$ unless when $\sum_{u\in T_j} a_u = B$ for $1 \leq j \leq n$ (see Lemma 2 below, where $U_j = \sum_{u\in T_j} a_u$ and $C = 2^{-nB}$, hence $(1-C^{1/n})^n = R$). Thus, the processor indices of the mapping correspond to a solution of $\mathcal{I}_1$, which concludes the proof. □

Lemma 2. *Given n real numbers U_j, $0 < U_j < 1$, whose product $C = \prod_{j=1}^{n} U_j$ is given, the quantity $R = \prod_{j=1}^{n}(1-U_j)$ has a unique maximum which is obtained when all the U_j have the same value $C^{1/n}$.*

Proof. The proof is by induction. For $n = 2$, $R = (1-U_1)(1-\frac{C}{U_1}) = 1 - U_1 - \frac{C}{U_1} + C$, which has a unique maximum when $U_1 = \frac{C}{U_1}$, hence the result. Assume the result is true for $n-1$ variables and consider the case with n variables. We have $R = \left(\prod_{j=1}^{n-1}(1-U_j)\right)(1-U_n)$. By induction the quantity $\prod_{j=1}^{n-1}(1-U_j)$ is maximum when the first $n-1$ variables U_j, whose product is $\frac{C}{U_n}$, have the same value $\left(\frac{C}{U_n}\right)^{\frac{1}{n-1}}$. We derive that $R = \left(1-\left(\frac{C}{U_n}\right)^{\frac{1}{n-1}}\right)^{n-1}(1-U_n)$.

Considering R as a function $R(U_n)$ and differentiating, we obtain

$$R'(U_n) = \left[(n-1)\left(1-\left(\frac{C}{U_n}\right)^{\frac{1}{n-1}}\right)^{n-2}\left(\frac{1}{n-1}C^{\frac{1}{n-1}}U_n^{-(\frac{1}{n-1})-1}\right)(1-U_n)\right] - \left(1-\left(\frac{C}{U_n}\right)^{\frac{1}{n-1}}\right)^{n-1}.$$

We simplify

$$R'(U_n) = \left(1-\left(\frac{C}{U_n}\right)^{\frac{1}{n-1}}\right)^{n-2}\left(C^{\frac{1}{n-1}}U_n^{-(\frac{1}{n-1})-1}(1-U_n) - 1 + C^{\frac{1}{n-1}}U_n^{-(\frac{1}{n-1})}\right),$$

which has a unique zero when $C^{\frac{1}{n-1}}U_n^{-(\frac{1}{n-1})-1} = 1$, hence when $U_n = C^{\frac{1}{n}}$. The value of U_j for $j < n$ is thus $\left(\frac{C}{U_n}\right)^{\frac{1}{n-1}} = C^{\frac{1}{n}}$, which concludes the proof. □

3.3 Approximation results

According to Lemma 1, it is always efficient to use only one single interval for *SpeedHom* platforms. In this section, we revisit the (mono-criterion) period minimization problem for *SpeedHet* platforms. This problem is NP-hard but the following theorem allows us to compare the single interval solution with the optimal one. Unfortunately, there are cases for which the period of the optimal solution with one single interval is m times greater than the period of the optimal solution with m intervals.

Lemma 3. *The optimal single-interval mapping for period minimization on a SpeedHet platform with p processors can be found in time $O(p \log p)$.*

Proof. Simply sort the processors by non-increasing speed: $s_1 \geq s_2 \geq \ldots s_p$. According to Equation (1), if we use the fastest i processors, we obtain a period $\mathcal{P} = \frac{W}{i \times s_i}$, where $W = \sum_{i=1}^{n}$ is the total work. Therefore the optimal solution is to select the value of i, $1 \leq i \leq p$, such that $i \times s_i$ is maximal and to use the fastest i processors. □

Theorem 3. *For a workflow with n stages, single interval mapping is a n-approximation algorithm for the period minimization problem on SpeedHet platforms. Moreover, this approximation factor cannot be improved.*

Proof. Consider that the optimal solution consists in $m \leq n$ intervals, with l_j processors in charge of interval I_j and s_j being the speed of the slowest processor for this interval. The period $\mathcal{P}_m$ of this mapping is such that $\sum_{i=d_j}^{e_j} w_i \leq l_j s_j \mathcal{P}_m$.

We can build a solution with one single interval which is an m-approximation of the solution: let k be one of the interval for which the product $l_k s_k$ is maximal, i.e., $l_j s_j \leq l_k s_k$ for $1 \leq j \leq m$. Then we use only processors in charge of this interval to compute the whole pipeline. The achieved period is:

$$\mathcal{P}_1 = \frac{\sum_{i=1}^{n} w_i}{l_k s_k} = \sum_{j=1}^{m} \frac{\sum_{i=d_j}^{e_j} w_i}{l_k s_k} \leq \sum_{j=1}^{m} \frac{\sum_{i=d_j}^{e_j} w_i}{l_j s_j} \leq m \mathcal{P}_m.$$

Thus, $\mathcal{P}_1 \leq \mathcal{P}_m$, which proves the approximation result.

To prove that we cannot obtain a better approximation result, we consider the following problem instance with n pipeline stages. Let K be a constant, which can be arbitrarily large For $1 \leq i \leq n$, we have $w_i = 1$, and $l_i = K^{i-1}$ processors of speed $s_i = 1/K^{i-1}$. The optimal solution has n intervals of size 1, and allocates stage $\mathcal{S}_i$ to the l_i processors of speed s_i. The optimal period is $\mathcal{P} = 1$. There is no idle time in the previous mapping, which shows its optimality. Furthermore, there is no other mapping without idle time, which shows the uniqueness of the optimal solution.

With a single interval mapping, we need to decide which processors to use. If we use the first j sets of processors, which are the fastest ones, we obtain a period

$$\mathcal{P}_j = \frac{\sum_{i=1}^{n} w_i}{(\sum_{k=1}^{j} l_k) \min_{k=1}^{j} s_k} = \frac{n}{(\sum_{k=1}^{j} l_k) s_j} = \frac{n K^{j-1}}{1 + K + \cdots + K^{j-1}} = \frac{n K^{j-1}(K-1)}{K^j - 1} = n(1 + o(K)).$$

Therefore, the optimal single-interval mapping, whose period is the minimum of the $\mathcal{P}_j$, has a period which is arbitrarily close to n for K large enough. This concludes the proof. □

The proof of Theorem 3 actually shows that single-interval mapping is a m-approximation for the period, where m is the number of intervals in the optimal solution (but of course m is unknown a priori, and we can only bound it by n, the number of stages). However, for bi-criteria problems, we stress that is always good to reduce the number of intervals in order to reduce the failure probability. Hence it looks very challenging to derive approximation algorithms for such problems on *SpeedHet-FailureHet* platforms.

4 Heuristics

In this section we propose heuristics for the general mapping problem on processors that are heterogeneous both in terms of computation speed, *SpeedHet*, and in terms of reliability, *FailureHet*. We concentrate on the optimization problem of minimizing the failure probability under a fixed period, because this is the more frequently encountered scenario. The opposite problem of minimizing the period under a fixed failure probability can be tackled by doing a simple binary search over the prescribed period using the proposed algorithms. We will see this in the evaluation of Section 6.

From Lemma 1 we know that for *SpeedHom* platforms, an optimal mapping can be found using a single-interval mapping. For this reason, we start with the design of a heuristic that employs only one interval. We expect this approach to be reasonable even for slightly speed-heterogeneous processors. Further, it will serve as a building block for our multiple-interval heuristic which we discuss subsequently.

4.1 Single-interval mapping

The heuristic proposed here, called OneInterval, minimizes the failure probability $\mathcal{F}$ under a fixed upper period bound $\mathcal{P}$ on a *SpeedHet-FailureHet* platform. As our approach is to group all stages into a single interval, the heuristic has to determine the number of teams and how the processors are assigned to these teams. The idea is to test all possible numbers of teams, i.e., from 1 to p. For each fixed number of teams l, we discard all processors that would not allow to achieve the given period $\mathcal{P}$. This procedure is motivated by Lemma 3 for finding the optimal mapping for period minimization. The remaining processors are assigned to the l teams, so that the total failure probability is minimized.

From Lemma 2 we know that to minimize the total failure probability, we need to minimize the maximum team failure probability among all teams. Note that this problem, similar to a partition problem, remains NP-hard. However, we use a simple greedy heuristic which is known to be efficient: we consider processors by non-decreasing failure probability and assign the next processor to the team with the highest failure probability. This heuristic, detailed in Algorithm 1, has a time complexity of $O(p^2 \log p)$.

4.2 Multiple-interval mapping

Theorem 3 demonstrates that even an optimal single-interval solution can be far from the absolute optimal on a *SpeedHet* platform. Hence we propose in this section a multiple-interval mapping heuristic, called MultiInterval. Again the heuristic minimizes the failure probability $\mathcal{F}$ under a fixed upper period bound $\mathcal{P}$ on a *SpeedHet-FailureHet* platform.

The proof of Theorem 3 shows that it can be beneficial for the minimal period to have multiple intervals. Our proposed heuristic therefore starts with $\min(n, p)$ intervals. Either we create one interval per stage, or we create at most p intervals, starting from the first stage and adding stages to the current interval while its computation cost is lower than $\frac{\sum_{i=1}^{n} w_i}{p}$ (step 1). The processors are then distributed over these intervals in such a way that the maximum ratio of interval computation weight to accumulated processor speed is

Algorithm 1: Heuristic OneInterval optimizing $\mathcal{F}$ with fixed $\mathcal{P}$ using single interval

for $l = 1$ *to* p **do**
- Discard processors whose speed is less than minimum speed necessary to achieve period with l teams: $procs = \{P_u, 1 \leq u \leq p : s_u \geq \frac{\sum_{i=1}^{n} w_i}{l \times \mathcal{P}}\}$
- Order processors *procs* by non-decreasing failure probability f_u into list L
- Create l teams T_k, $1 \leq k \leq l$, set failure probability of all teams to $f_{T_k} = 1$
- **foreach** P_u *in* L **do**
  - Find team T_{max} with highest failure probability
  - Assign P_u to T_{max}; $f_{T_{max}} \leftarrow f_{T_{max}} \times f_u$
- Compute the total failure probability $\mathcal{F}_l = 1 - \prod_{k=1}^{l}(1 - T_k)$

Choose among the p previous solutions the one with the lowest total failure probability $\mathcal{F}_l$

minimized: we greedily add processors, sorted by non-increasing speed, to intervals with the highest current ratio (step 2). Teams are formed and processors are allocated in each interval using the single-interval Algorithm 1, including the utilization of processors that were unused in the allocation of previous intervals (step 3). If no solution can be found for an interval, we increase the bound on the period for this interval until Algorithm 1 returns a valid allocation. Then, if the period bound is not achieved for at least one interval, we merge the interval with the largest period, either with the previous or with the next interval, until the bound on the period is respected (step 4). The actual optimization of the failure probability happens in the last step (step 5), in which we merge intervals with highest failure probability as long as it is beneficial. Indeed, we know from Section 3.1 that the fewer intervals and teams the (potentially) better is the failure probability. In other words, Equation (2) for the failure probability $\mathcal{F}$ is minimized for $l = 1$. Note that Algorithm 1 is called each time we try to merge intervals (steps 4 and 5).

This heuristic, described in has a time complexity of $O(p^3 \log p)$.

5 Optimal algorithm using A*

In order to find optimal solutions for small instances of our problem, we employ an A* best-first state space search algorithm [7, 13]. A* has successfully been employed before for optimal scheduling, e.g. [14]. For the given problem it is especially interesting, as the non-linear nature of the failure probability rules out other exact approaches, e.g. Integer Linear Programming. A* is a generic algorithm that performs a best-first search over all possible solutions. Every node in the search space is a state s that represents a partial solution and has an underestimated cost value $c(s)$ associated with it. This value represents the minimum cost of any complete solution based on the partial solution represented by s. For our mapping problem a state s is a partial mapping and the cost value $c(s)$ is either the failure probability or the period. A* maintains an OPEN list of states, populated with the initial state in the beginning. At each step, A* expands the state with the lowest $c(s)$ value, i.e., it creates new states that are larger partial solutions. A c value is calculated for each of these new states and they are inserted into

Algorithm 2: Heuristic MULTIINTERVAL optimizing $\mathcal{F}$ with fixed $\mathcal{P}$ using multiple intervals

▷ *1. Make initial intervals*
if $n \leq p$ **then**
  Create n intervals, one for each stage
else
  Create p intervals, each with approximate computation weight of $\frac{\sum_{i=1}^{n} w_i}{p}$
Let $w(I)$ be the computation amount of interval I
let $speed(I)$ be the sum of all processor allocated to interval I (zero for the moment)
▷ *2. Assign processors to intervals*
foreach *each* P_u, $1 \leq u \leq p$, *in non-decreasing speed* s_u *order* **do**
  Assign P_u to interval I_{max} with highest ratio $w(I_{max})/s(max)$
▷ *3. Build Team in each interval*
for *each interval* I_i *in non-decreasing* $\frac{w(I_i)}{s(I_i)}$ *order* **do**
  Apply Algorithm 1 for I_i and its allocated processors plus unused processors of previous intervals
  If Algorithm 1 is unable to find a solution within the period bound, increase this bound and re-run Algorithm 1, until we have a valid solution
Let $\mathcal{P}(I) = w(I)/(n_I \times s_{minI})$, where n_I is the number of teams chosen for interval I and s_{minI} the minimum speed of processors allocated to interval I
▷ *4. Merge intervals to reach period bound*
while *the period bound is not reached, and there are at least two intervals* **do**
  Find interval I_i with highest period $\mathcal{P}(I_i)$
  Merge I_i with I_{i-1} if it exists; apply Algorithm 1 on this interval
  Merge I_i with I_{i+1} if it exists; apply Algorithm 1 on this interval
  Accept the merging which results in smallest period for the current interval
▷ *5. Merge intervals to decrease failure probability*
while *we can decrease the failure probability, and there are at least two intervals* **do**
  Find interval I_i with highest failure probability $\mathcal{F}_{I_i}$
  Merge I_i with I_{i-1} if it exists; apply Algorithm 1 on this interval
  Merge I_i with I_{i+1} if it exists; apply Algorithm 1 on this interval
  Accept the merging which results in better remaining highest failure probability, provided that the period bound is satisfied
Return the current solution, provided it satisfies the period bound

the OPEN list, so that the list remains sorted by non-decreasing c value. The algorithm terminates when the state s that is taken out of the OPEN list is a goal state, i.e., a complete solution. With a good cost estimation function c, A* can find a complete mapping which is guaranteed to be optimal (with certain conditions on the function $c(s)$) without exploring the entire solution space. The determining factor in the effectiveness of A* is the quality of the underestimated $c(s)$ values, i.e., how tight they are to the real costs.

A CLOSED set of the states that have already been visited, which is used in classical A*, is not necessary to maintain, because our state space construction does not create duplicates.

5.1 Solution space

The first step in the design of A* is to describe the solution space of the problem and the successive construction of larger partial solutions. In order to find the optimal solution for our general mapping problem, the grouping into intervals and the mapping of the processors is integrated into one solution space. The proposed approach consists in sequentially assigning stages to intervals, and processors to teams for these intervals. We start with an *initial state*, which is an empty state, where no stage nor processor has been assigned yet. A state can be *expanded* by considering the first stage $\mathcal{S}_j$ which is not yet allocated. We have two choices when adding this stage:
(i) Stage $\mathcal{S}_j$ is included in the previous interval, and inherits from the team structure: only one new state is added to the solution space.
(ii) Stage $\mathcal{S}_j$ starts a new interval, without any processor assigned to it at the moment, which forms the new state. Then, the state expansion process continues by assigning the not yet allocated processors, i.e., the free processors, to the current interval. For each free processor P_u, we have three choices, hence (potential) new states:

1. P_u is not allocated to the current interval;
2. P_u is allocated to an existing team for the current interval;
3. P_u is allocated to a new team for the current interval.

Some *invalid states* may be created this way, but are immediately discarded, such as a state where no processor at all has been allocated to an interval. All valid states are added in the sorted OPEN list of states. Finally, a *goal state* is a state where all stages have been included into intervals, and the process of allocating processors to the last interval is complete.

Fig. 2 shows the state tree after expanding all states, for a small example of an application with two stages, on two processors. For this example, the expansion reaches six goal states and four invalid states.

5.2 Underestimate functions

The $c(s)$ function is the central part of an A* algorithm. It is an underestimate of the exact minimum cost $c^*(s)$ of any goal state that is based on the state s. If the function $c(s)$ fulfills $c(s) \leq c^*(s)$ for any state s, it is called admissible. With an admissible $c(s)$ function, A* is guaranteed to find an optimal solution. The number of examined states

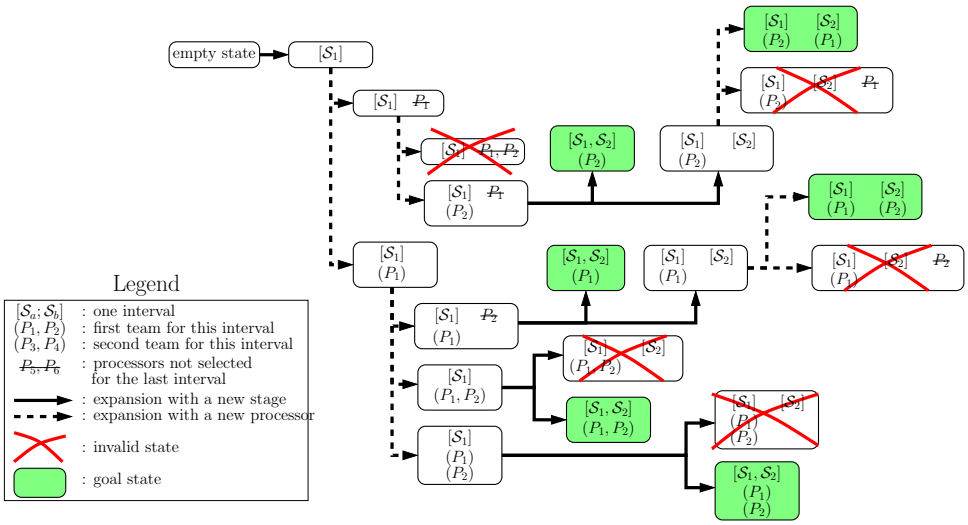


Figure 2: Complete state tree for two-stage application mapped on two processors. E.g., state at bottom right corner is made of a single interval $[\mathcal{S}_1, \mathcal{S}_2]$ and two teams with one processor each: (P_1) and (P_2).

depends on how close $c(s)$ is to $c^*(s)$. In general, the more accurate, the fewer states have to be examined.

In our problem, we have to design two such underestimate functions, both for $\mathcal{F}$ and $\mathcal{P}$. Like in the previous section, we concentrate on the problem of minimizing the failure probability $\mathcal{F}$ with a given period bound $\mathcal{P}$. Thus, we first need an underestimate function f for the failure probability: for a partial solution s, $f(s)$ is an underestimate of the minimum failure probability of any complete solution based on s. We also have to check that a solution never exceeds the period bound $\mathcal{P}$. Therefore, we also use an underestimate function p for the period: $p(s)$ is an underestimate of the minimum period of any complete solution based on the partial state s.

5.2.1 Estimate of the failure probability

To understand the underestimation of the failure probability of a state s, recall that the failure probability does not depend on the structure of intervals, but only on how processors are grouped into teams, Equation (2). Adding a new team to a partial mapping can never reduce the failure probability, only increase it.

The underestimation must therefore assume that all remaining stages are added to the currently open interval (having more than this last interval would at least require one additional team). It is straight forward to calculate the failure probabilities of the already closed teams of the closed intervals. The difficulty arises for underestimating the failure probabilities of the teams of the last interval. It must be assumed that all free processors are assigned to the already existing teams of the last interval (if there is no team yet, all free processors are assigned to a single team). Assigning the free processors to the existing teams in an optimal way is an NP-hard problem though.

From Lemma 2 we know that the best failure probability would be achieved if we perfectly balanced the failure probability among the teams. Thus, we make the ideal-

izing assumption that processors are perfectly divisible, and that we have an "amount of reliability" (corresponding to the failure probability of a single team made of all free processors). This amount is then distributed to the existing teams, by starting with the teams having the worst failure probability, to minimize the maximum failure probability. The result of this procedure gives us an underestimate of the best achievable failure probability for the open teams. Together with the failure probabilities of the closed teams we obtain an underestimate for the considered partial state. The complete process is described in Algorithm 3.

Algorithm 3: Estimate of the failure probability for incomplete teams

Input: K teams with $T_1, \ldots, T_K$, with failure probabilities $f_{T_1}^{team}, \ldots, f_{T_K}^{team}$, and p available processors with failure probabilities $f_1, \ldots, f_p$

Compute the aggregated available failure probability $F \leftarrow \prod_{u=1}^{p} f_u$; we consider now that we have a divisible processor with failure probability F.

while $F < 1$ **do**

  Select the X teams $T_{\mathsf{max}_1}, \ldots, T_{\mathsf{max}_X}$ with maximum failure probability f_{max}^{team}

  Compute the second maximum failure probability f_{max2}^{team} (the maximum among all other teams)

  $\rho \leftarrow \max\left\{\frac{f_{\mathsf{max2}}^{team}}{f_{\mathsf{max}}^{team}}, F^{1/X}\right\}$

  foreach *team* T_i *in* $T_{\mathsf{max}_1}, \ldots, T_{\mathsf{max}_X}$ **do**

    Decrease the failure probaility of T_i by ρ: $f_i^{team} \leftarrow f_i^{team} \times \rho$

  Update the available failure probability: $F \leftarrow F/\rho^X$

5.2.2 Estimate of the period

Our estimation of the period takes into account the stages already structured into intervals, and the processors already allocated to teams for these intervals. We simply use Equation (1) on those closed intervals. In order to better estimate the period of a final solution expanded from the current state, we also compute the minimum period that can be achieved with the other stages and processors. As outlined in Section 3.1, the mono-criterion problem of period minimization is already NP-hard in the heterogeneous case. However, we can derive a simple lower bound on the period, by assuming a perfect load-balancing of processor speeds: $\mathcal{P} \leq \frac{\sum_{\mathcal{S}_i} w_i}{\sum_{P_u} s_u}$.

We use this bound, with all stages that do not belong to the closed intervals, and all processors that are not allocated to a closed interval, to derive a second underestimate of the achievable period. The final underestimate of the period is then the maximum of the previous two estimations.

5.3 Optimization

A classical optimization for A* is employed. We first compute the solution given by a heuristic (in practice, the single-interval heuristic OneInterval). During A*, all states that have an underestimated failure probability larger than the one obtained from the

heuristic can be immediately discarded. This does not create less states (as those states would not have been expanded) but can save queue memory which is crucial for A*.

6 Evaluation results

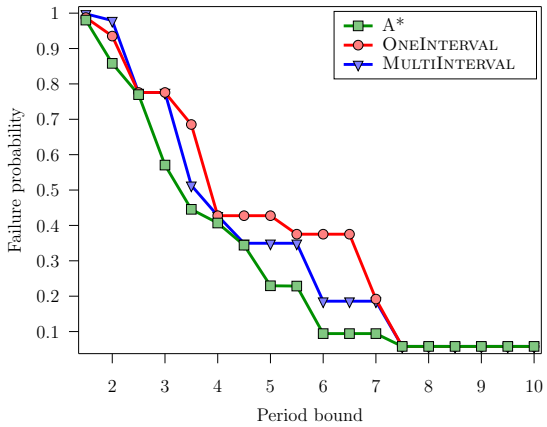
In this section, we present the simulations we have undertaken to evaluate the performance of the two proposed heuristics and the A* algorithm.

6.1 Simulation settings

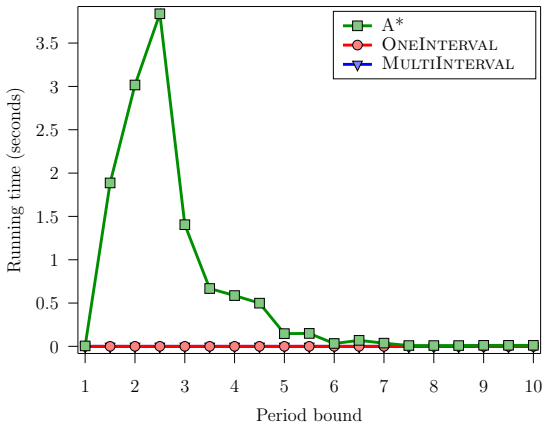
We have randomly generated several workload scenarios. Stage computational weights are uniformly distributed in the interval $[1; 10]$. Processor speeds are uniformly distributed in the interval $[1; 10]$, while processor failure probabilities are uniformly distributed in the interval $[0.1; 0.9]$. The number of stages randomly varies from 5 to 20, just as the number of processors. However, for A*, we had to limit both numbers to 10: for larger scenarios, the memory demand of A* is too high. Finally, all tests were performed on a quad-processor machine (64-bit AMD Operon at 2.2GHz) with 32Gbytes of RAM.

6.2 Results

We first give the results of both heuristics compared to A*. Figure 3(a) shows the behavior of the different algorithms on a given scenario (with 8 processors and 8 stages). We see that the heuristics approach the optimal (minimal) failure probability computed by A*. For a period bound between 3 and 7, MultiInterval achieves a better performance than OneInterval: the use of several intervals is then crucial to get the optimal performance. On the contrary, for a period bound around 2, OneInterval is closer to the optimal: because of its initial construction of the interval, and of the choice of the intervals to merge, MultiInterval sometimes fails to find the best one-interval solution.



(a) Failure probabilities for different period bound



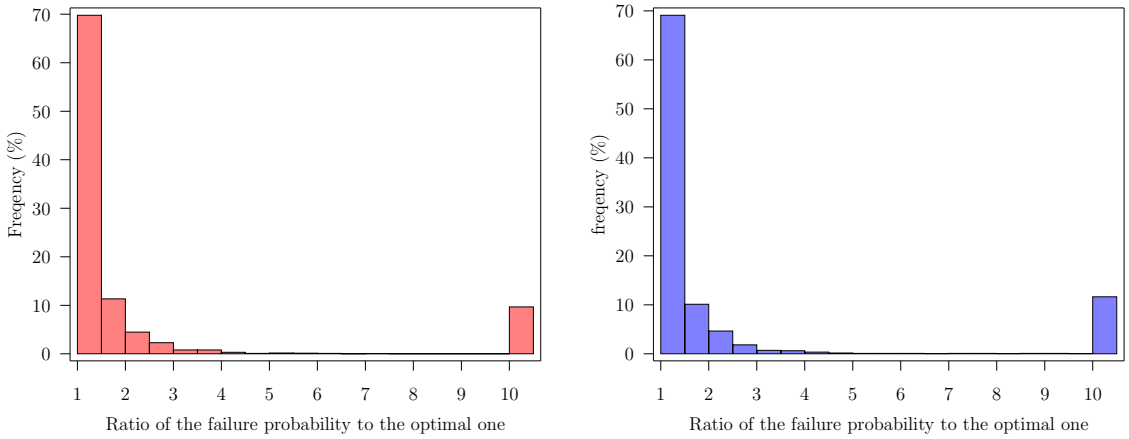
(b) Running times

Figure 3: Behavior on an example scenario.

Figure 3(b) shows the running times of the algorithms: the heuristics are extremely fast, whereas A* need a few seconds to run on this example: when the period is neither too

small nor too large (say in the interval $[1.5; 4.5]$), A* is confronted with a large solution space, hence it performs many operations. However, the main limitation of A* is the memory needed to store all partial states generated during the execution; this is why we we had to limit our search with A* to smaller scenarios.

Figures 4(a) and 4(b) show the distribution of the ratio between the failure obtained by a heuristic, and the optimal failure computed with A*: when this ratio is close to one, the performance of the heuristic is good. The box at 10 accounts for all scenarios where the heuristic does not find a solution, whereas A* finds one. We can see that both heuristics perform similarly. On average in the cases where the heuristics find a solution, their failure probability is 20% above the optimal failure probability. In very few cases (a couple of scenarios, representing less than 0.1% of all cases), the failure probability can be up to 10 times the optimal one. However, the heuristic sometimes fail to find a solution within the period bound (whereas A* exhibits a solution): this happens in 9.6% of the cases for OneInterval, and 11.5% for MultiInterval.



(a) Heuristic OneInterval (b) Heuristic MultiInterval

Figure 4: Performance of the heuristics compared to the optimal solution

In the complete set of experiments (with larger scenarios), OneInterval is better than MultiInterval in 61% of the cases, and MultiInterval is better than OneInterval in 20% of the cases (in the remaining 19% cases, either they both fail, or they find the same solution). On average, OneInterval gives a failure probability which is 2% higher than MultiInterval.

We also compared OneInterval with the optimal solution restricted to one interval: it is easy to limit the search space of A* to solution using only one interval. This shows that OneInterval is very good, with an average failure probability only 0.05% larger than the optimal one (and 5% larger in the worst case). This proves that the problem restricted to one interval is not very difficult, but the difficult part is to find how to split stages into intervals, and how to allocate processors to these intervals.

7 Conclusion

In this paper, we have investigated the problem of mapping pipelined applications onto heterogeneous platforms, while maximizing both the throughput and the reliability of

the application. The main challenge was to perform a good trade-off between reliability and performance in the use of processors. Given that the complexity of mono-criterion problems had already been established, we presented new results concerning bi-criteria optimization. In particular, we derived a polynomial-time algorithm for *Speed-Hom-FailureHom* platforms, while we proved that the problem is NP-hard as soon as one level of heterogeneity is introduced. Moreover, we provided approximation results to compare the optimal mapping consisting of a single interval with any other solution.

To address the practical solution of these difficult problems, we proposed polynomial-time heuristics for the most general problem instance *SpeedHet-FailureHet*. Strongly inspired by the theoretical study, the main heuristic is built upon a simpler heuristic which proposes a single interval mapping. Because of the strong non-linearity of the failure probability function, we were not able to formulate an integer linear program to compute the optimal solution for small problem instances. Rather, we investigated the use of A*, and proposed non-trivial underestimate functions leading to an efficient execution of this (exponential-time) algorithm. The experimental results demonstrated the very good behavior of the heuristics, which failed only in 10% of the tests, and which otherwise returned a failure probability only 20% worse than the optimal one.

Future work includes the investigation of further approximation results, enhanced multi-interval heuristics, and improved cost estimation and pruning techniques for the exact A* based algorithm.

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, V. Rehn-Sonigo, and Y. Robert. Optimizing latency and reliability of pipeline workflow applications. In *HCW'2008, the 17th Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2008.

[3] A. Benoit and Y. Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica*, Oct. 2008. http://dx.doi.org/10.1007/s00453-008-9229-4.

[4] A. Benoit and Y. Robert. Mapping pipeline skeletons onto heterogeneous platforms. *J. Parallel 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] DataCutter Project: Middleware for Filtering Large Archival Scientific Datasets in a Grid Environment. `http://www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm`.

[7] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, July 1985.

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

[9] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An Algorithm for Automatically Obtaining Distributed and Fault-Tolerant Static Schedules. In *International Conference on Dependable Systems and Networks, DSN'03*, 2003.

[10] S. L. Hary and F. Ozguner. Precedence-constrained task allocation onto point-to-point net works for pipelined execution. *IEEE Trans. Parallel and Distributed Systems*, 10(8):838–851, 1999.

[11] K. Hashimito, T. Tsuchiya, and T. Kikuno. Effective Scheduling of Duplicated Tasks for Fault-Tolerance in Multiprocessor Systems . *IEICE Transactions on Information and Systems*, E85-D(3):525–534, 2002.

[12] 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.

[13] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[14] A. Z. S. Shahul and O. Sinnen. Optimal scheduling of task graphs on parallel systems. In *Proc. of 9th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'08)*, Dunedin, New Zealand, Dec. 2008. IEEE Press.

[15] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

[16] 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, PPoPP'95*, pages 134–143. ACM Press, 1995.

[17] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *ACM Symposium on Parallel Algorithms and Architectures SPAA'96*, pages 62–71. ACM Press, 1996.

[18] K. Taura and A. A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Heterogeneous Computing Workshop*, pages 102–115. IEEE Computer Society Press, 2000.

[19] Q. Wu, J. Gao, M. Zhu, N. Rao, J. Huang, and S. Iyengar. On optimal resource utilization for distributed remote visualization. *IEEE Trans. Computers*, 57(1):55–68, 2008.

[20] Q. Wu and Y. Gu. Supporting distributed application workflows in heterogeneous computing environments. In *14th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2008.