Sharing resources for performance and energy optimization of concurrent streaming applications

Anne Benoit, Paul Renaud-Goud and Yves Robert

LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France
UMR 5668 - CNRS - ENS Lyon - UCB Lyon - INRIA
{Anne.Benoit|Paul.Renaud-Goud|Yves.Robert}@ens-lyon.fr

February 16, 2010

LIP Research Report RR-2010-05

Abstract

We aim at finding optimal mappings for concurrent streaming applications. Each application consists of a linear chain with several stages, and processes successive data sets in pipeline mode. The objective is to minimize the energy consumption of the whole platform, while satisfying given performance-related bounds on the period and latency of each application. The problem is to decide which processors to enroll, at which speed (or mode) to use them, and which stages they should execute. Processors can be identical (with the same modes) or heterogeneous. We also distinguish two mapping categories, interval mappings, and general mappings. For interval mappings, a processor is assigned a set of consecutive stages of the same application, so there is no resource sharing across applications. On the contrary, the assignment is fully arbitrary for general mappings, hence a processor can be reused for several applications. On the theoretical side, we establish complexity results for this tri-criteria mapping problem (energy, period, latency), classifying polynomial versus NP-complete instances. Furthermore, we derive an integer linear program that provides the optimal solution in the most general case. On the experimental side, we design polynomial-time heuristics, and assess their absolute performance thanks to the linear program. One main goal is to assess the impact of processor sharing on the quality of the solution.

Key words: mapping, concurrent streaming applications, heterogeneous platforms, resource sharing, energy, latency, period.

Contents

1 Introduction

In this paper, we aim at optimizing the parallel execution of several pipelined applications on a given platform. Such streaming applications are ubiquitous in streaming environments, as for instance video and audio encoding and decoding, DSP applications, image processing, and so on ([6, 15, 9, 16, 17]). For each application, a sequence of data sets enters the input stage and progresses from stage to stage at a fixed rate until the final result is computed. Each stage has its own communication and computation requirements: it reads an input from the previous stage, processes the data and outputs a result to the next stage. Each data set is input to the first stage, and final results are output from the last stage. A new data set enters the system each application period, and results are output at the same periodic interval.

The objective is to minimize the energy consumption of the whole platform, while satisfying given performance-related bounds on the period and latency of each application. This multi-criteria approach targets a trade-off between the users and the platform manager. The formers have specific requirements for their application, while the latter has crucial economical and environmental constraints. Indeed, the energy saving problem is becoming increasingly important, not only because of the sole cost of energy, but also because of the cost of cooling systems and related infrastructures. To help reduce energy costs, modern computing centers provide multi-modal processors: every processor has a discrete number of predefined speeds (or modes), which correspond to different voltages the processor can be subjected to. The power consumption is the sum of a static part (the cost for a processor to be turned on) and a dynamic part. This dynamic part is a strictly convex function of the processor speed, so that the execution of a given amount of work costs more energy if a processor runs in a higher mode [11]. On the one side, faster modes allow for fulfilling the performance criteria, and on the other side, they lead to a higher energy consumption, hence the above mentioned trade-off to be found.

The main performance-oriented criteria for pipelined applications are period and latency. The period of an application is the inverse of the throughput, i.e., it corresponds to the time interval between the arrival of two consecutive data sets. The period is fixed by the applicative setting, and we must ensure that data sets are processed fast enough so that there is no accumulation of data sets in the pipeline. The latency of an application is the time elapsed between the beginning and the end of the execution of a given data set, hence it measures the response time of the system to process the data set entirely. These two criteria alone already are antagonistic. The smallest latency is obtained when no communication is paid, i.e., when the same processor executes all the stages of an application. However, such a mapping may well exceed the bound on the period, since the same processor must process an entire application. Adding energy consumption as a third criterion renders everything even more complex. Obviously, energy is minimized by enrolling a single processor for all applications, namely the one with the smallest mode available among all platform resources; but again, such a mapping would most certainly exceed period and latency bounds.

This work is a follow-on of [3], where we have provided a comprehensive analysis of various instances of the previous multi-criteria optimization problem. However, the mapping rules and performance models used in this paper are different. In a nutshell, a comprehensive assessment of *one-to-one* and *interval mappings* is given in [3]. Such mappings restrict the assignment of stages to processors: each enrolled resource can execute only a single stage (one-to-one mapping) or a set of consecutive stages (interval mapping) of a given application. Therefore no inter-application reuse of resources is authorized. While prohibiting such a reuse may make good sense in some situations (e.g., for security reasons), it is also very likely to waste resources and to increase energy consumption. Indeed, without reuse, more processors are enrolled, hence the static energy gets higher, and these processors cannot benefit from a good load balancing of

computation costs across applications, hence a worse resource utilization. From the platform manager point of view, resource sharing among (non critical) applications is a key ingredient to efficiently servicing several users.

In this paper, we investigate the impact of resource sharing on the quality of the solution with respect to the three optimization criteria (energy, period and latency). We thus deal with *general* mappings where application stages can arbitrarily be assigned to processors. Unfortunately, general mappings come with a price, that of intricate scheduling problems for period and latency: even when the mapping is given, scheduling the execution is a problem of combinatorial nature [1]. With general mappings, a processor typically has several incoming and/or outgoing communications, and it is difficult to orchestrate these operations so as to minimize conflicting objectives such as period and latency. Therefore, we focus in this paper on the problem in which bounds on period and latency are fixed by the application designer, and we relax the definition of the latency using the approach of Hary and Ozguner [9]. Instead of computing the longest path, we approximate the latency L as $L = (2m-1)P$, where P is the period, i.e., the rate at which data sets enter the system, and m is the number of processor changes in the mapping. A processor change occurs each time when a stage and its successor are not mapped onto the same processor. The intuition is that the whole application is executed synchronously, and each data set progresses concurrently within a period. With m processors and $m-1$ processor changes, hence $m-1$ communications to orchestrate, each data set traverses the platform within $2m-1$ periods. We adopt the model of [9] throughout the paper, and refer to Section 2 for further details on mapping rules and objectives. The problem can then be defined as follows: given a period P_a and a bound on the latency L_a for each application a, find a mapping which consumes the minimum amount of energy, while satisfying the performance constraints: application a is processed at a period P_a, and its latency is not greater than L_a; in other words, the number m_a of processor changes in the mapping of application a does not exceed $\lfloor \frac{L_a/P_a+1}{2} \rfloor$.

A first contribution of this paper is to provide complexity results for the tri-criteria optimization problem under the new model. We restrict to homogeneous platforms whose processors have identical modes and static energy; otherwise the problem with a single application mapped onto homogeneous and uni-modal processors, paying no communication cost, is NP-complete (straightforward reduction from 2-PARTITION [8], with no bound on the latency and a tight bound on the period such that there is a solution if and only if the period can be respected: it is equivalent to a period minimization problem). We show that the problem is polynomial for interval mappings on homogeneous platforms with Hary and Ozguner's model, while it was NP-complete with the longest path model [3], thereby demonstrating the impact of the model for the latency. We also show that the tri-criteria problem becomes NP-complete for general mappings on homogeneous platforms.

Another contribution of the paper is to evaluate the impact of resource sharing, by comparing the quality of interval mappings and of general mappings. To this end, we design a set of polynomial-time heuristics, with and without reuse, and we experimentally compare their performance on a large set of experiments. We also evaluate the absolute performance of the heuristics on small problem instances, through the solution of an integer linear program.

The paper is organized as follows. We start by describing the framework in Section 2, with the description of the applications, the platform, the different mapping strategies and the energy model. Then we provide complexity results for different mapping strategies on homogeneous platforms in Section 3. In Section 4, we first describe an integer linear program, which allows us to solve the NP-complete problem in the general case; then we design several heuristics to provide polynomial-time solutions to the tri-criteria problem; and finally we study their relative performance, and their absolute performance with respect to the integer linear program. We conclude in Section 5.

2 Framework

2.1 Applicative framework

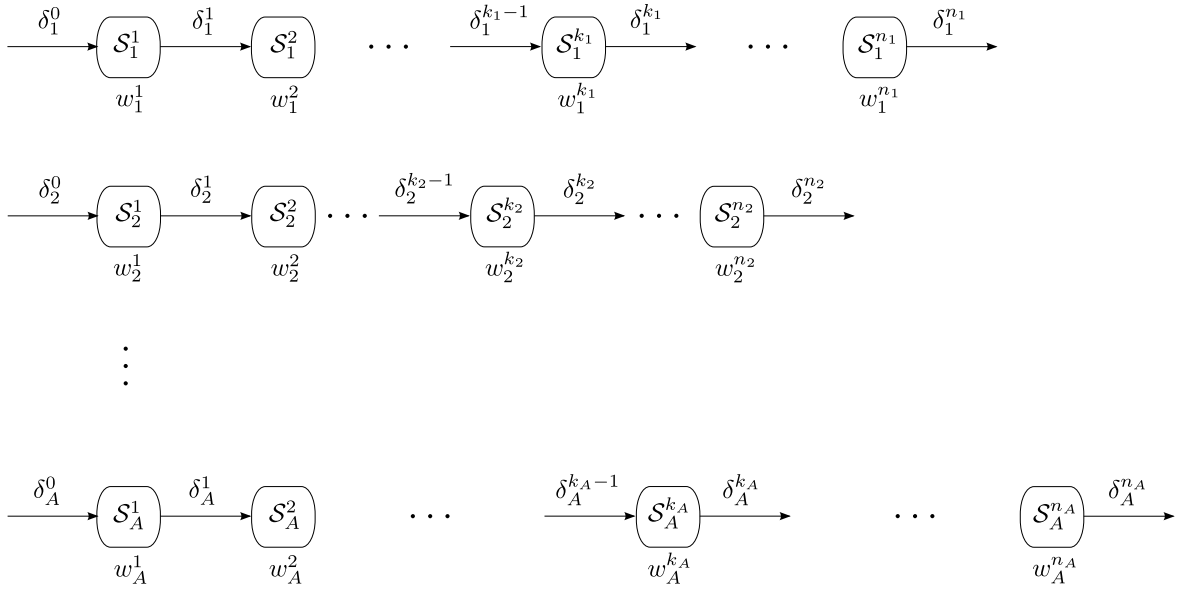

Figure 1: Notations

We consider A application workflows ($A \geq 1$) to be executed concurrently; each application operates on a collection of data sets that are executed in a pipeline fashion. For $1 \leq a \leq A$, application a consists in n_a stages, and for $1 \leq k \leq n_a$, we denote by $\mathcal{S}_a^k$ the k-th stage of application a. Stage $\mathcal{S}_a^k$ receives an input data of size δ_a^{k-1}, performs w_a^k computations, and finally outputs a data of size δ_a^k. A new data set enters the system every P_a time-units; P_a is the period of application a. The total number of stages is $N = \sum_{a=1}^{A} n_a$.

The first stage of each application $\mathcal{S}_a^1$, $a \in \{1, \ldots, A\}$, receives an input of size δ_a^0 from the outside world, while the last stage of each application $\mathcal{S}_a^{n_a}$ returns the result, of size $\delta_a^{n_a}$, to the outside world. This application model is illustrated on Figure 1.

2.2 Target platform

The platform is composed of p processors, which are fully interconnected; there is a bidirectional link $\mathsf{link}_{u,v} : \mathcal{P}_u \leftrightarrow \mathcal{P}_v$ between any processor pair $\mathcal{P}_u$ and $\mathcal{P}_v$, of bandwidth $b_{u,v}$. For simplification, we assume that $2A$ additional processors $\mathcal{P}_{\mathsf{in}_1}, \ldots, \mathcal{P}_{\mathsf{in}_A}$ and $\mathcal{P}_{\mathsf{out}_1}, \ldots, \mathcal{P}_{\mathsf{out}_A}$ are devoted to input/output operations of the applications (in fact these additional processors are virtual processes that may well be shared by the same physical resource). Initially, for each $a \in \{1, \ldots, A\}$, the input data for each task of the application a resides on $\mathcal{P}_{\mathsf{in}_a}$, while all results must be returned to and stored on $\mathcal{P}_{\mathsf{out}_a}$. These special processors are all connected to the p processors of the target platform.

We use a linear cost model for communications; hence it takes $X/b_{u,v}$ time-units for $\mathcal{P}_u$ to send (resp. receive) a message of size X to (resp. from) $\mathcal{P}_v$. Note that there is no need to have a physical link between all processor pairs. We may have a switch, or a path composed of several physical links, instead, to interconnect $\mathcal{P}_u$ and $\mathcal{P}_v$; in the latter case, $b_{u,v}$ is the bandwidth of the slowest link in the path. In addition to link bandwidths, we have processor network cards

that bound the total communication capacity of each computing resource. We denote by B_u^{in} (resp. B_u^{out}) the capacity of the input (resp. output) network card of processor $\mathcal{P}_u$. In other words, $\mathcal{P}_u$ cannot receive more than B_u^{in} data items per time-unit, and it cannot send more than B_u^{out} data items per time-unit. In this paper, we mainly target *communication-homogeneous* platforms, with identical communication devices for each processor: all link bandwidths are identical ($b_{u,v} = b$ for $1 \leq u, v \leq p$), and all network cards are identical ($B_u^{in} = B^{in}$, $B_u^{out} = B^{out}$ for all $1 \leq u \leq p$). However, the linear program of Section 4.1 applies to heterogeneous platforms as well.

As stated above, processors are multi-modal. Each processor $\mathcal{P}_u$ is associated with a set $\mathsf{S_u}$ of speeds, or modes: $\mathsf{S_u} = \{s_{u,1}, \ldots, s_{u,m_u}\}$. To ease notations, we add a special mode 0 in which the processor is inactive, and thus $s_{u,0} = 0$. *Processor-homogeneous* platforms are platforms whose processors have identical static energy and speeds, i.e., share a common speed set ($\mathsf{S_u} = \mathsf{S}$ for $1 \leq u \leq p$). We assume that *processor-homogeneous* platforms are also communication-homogeneous, so that they represent typical parallel machines. *processor-heterogeneous* platforms also are communication-homogeneous, but they have different-speed processors ($\mathsf{S_u} \neq \mathsf{S_v}$). They correspond to networks of workstations with plain TCP/IP interconnects or other LANs.

Finally, the communication model is the bounded multi-port model with overlap [10]. In the bounded multi-port model, the total communication volume outgoing from a given node is bounded (by the capacity of its network card), but several communications along different links can take place simultaneously (provided that the link bandwidths are not exceeded either). In addition, independent communications and computations can overlap. It has been pointed out that recent multi-threaded communication libraries such as MPICH2 [13] now allow for initiating multiple concurrent send and receive operations, thereby providing practical realizations of the multi-port model [2].

2.3 Mapping strategies

The mapping is an allocation function, which associates a processor number to each stage number, as well as a speed at which each processor is running.

For *general mappings with processor reuse*, there are no constraints on the allocation function. We must carefully decide how the speed of each processor is shared among all stages it is assigned to. Similarly, a communication link or processor network card may be involved in several communications, which implies to sharing bandwidths and card capacities too. Hence the question is the following: given the mapping, and given a period P_a and threshold latency L_a for each application $a \in \{1, \ldots, A\}$, is it possible to determine which fraction of computing and communicating resources to assign to each operation so that all application periods are realized and all latency thresholds are met?

Recall that we consider the latency model described in [9], in which one period is accounted for each computation of an interval of stages and for each inter-processor communication. We observe that given the mapping, we know m_a, the number of intervals, or processor changes, for each application a. We can thus check immediately whether the bounds on the latency are respected, i.e., $(2m_a - 1)P_a \leq L_a$ for $a \in \{1, \ldots, A\}$.

Now for the periods, the key idea is to distribute platform resources parsimoniously, and allocate only the needed CPU fraction to each computation, and the needed bandwidth fraction to each communication, so that the period constraint is fulfilled. The mapping is valid if neither processor speeds, nor link bandwidths, nor network card capacities are exceeded. First we merge consecutive stages $[\mathcal{S}_a^i, \ldots, \mathcal{S}_a^j]$ of application a mapped onto a same processor as one single coalesced stage $\hat{\mathcal{S}}_a^k$, with computing cost $\hat{w_a^k} = \sum_{k'=i}^{j} w_a^{k'}$, and output communication $\hat{\delta_a^k} = \delta_a^j$. The transformed application now has exactly m_a stages. In the following, stage $\hat{\mathcal{S}}_a^k$ corresponds

to the k-th stage of the transformed application a, for $1 \leq k \leq m_a$.

As for computations, consider a processor $\mathcal{P}_u$ and an application a. We define $\mathcal{K}_a^u$ such that $k \in \mathcal{K}_a^u$ if and only if $\hat{\mathcal{S}}_a^k$ is processed by processor $\mathcal{P}_u$; $\mathcal{K}_a^u$ is the set of stages of (transformed) application a processed by $\mathcal{P}_u$. Then, for all a and u, and for each $k \in \mathcal{K}_a^u$, we allocate the speed fraction $s_{a,u}^k = \hat{w_a^k}/P_a$ for $\mathcal{P}_u$ to execute $\hat{\mathcal{S}}_a^k$.

Similarly for communications, we define $\mathcal{K}_a^{u,v}$ such that $k \in \mathcal{K}_a^{u,v}$ if and only if $\hat{\mathcal{S}}_a^k$ is processed by $\mathcal{P}_u$ and $\hat{\mathcal{S}_a^{k+1}}$ is processed by $\mathcal{P}_v$, i.e., there is a communication to pay between $\mathcal{P}_u$ and $\mathcal{P}_v$. Note that $u \neq v$, otherwise stages $\hat{\mathcal{S}}_a^k$ and $\hat{\mathcal{S}_a^{k+1}}$ would have been merged as a single stage. Formally, $k \in \mathcal{K}_a^{u,v} \Leftrightarrow k \in \mathcal{K}_a^u$ and $k+1 \in \mathcal{K}_a^v$. Then we allocate the bandwidth fraction $b_{a,u,v}^k = \hat{\delta_a^k}/P_a$ to the communication.

The period of each application can be respected if and only if all the following inequalities are satisfied. There might be some spare speed and bandwidth if these are strict inequalities, and resources are fully utilized in case of equalities.

- $\forall 1 \leq u \leq p,\ \sum_{a=1}^{A} \sum_{k \in \mathcal{K}_a^u} s_{a,u}^k \leq s_u$,
- $\forall 1 \leq u, v \leq p,\ u \neq v,\ \sum_{a=1}^{A} \left(\sum_{k \in \mathcal{K}_a^{u,v}} b_{a,u,v}^k + \sum_{k \in \mathcal{K}_a^{v,u}} b_{a,v,u}^k \right) \leq b_{u,v}$,
- $\forall 1 \leq u \leq p,\ \sum_{v=1}^{p} \sum_{a=1}^{A} \sum_{k \in \mathcal{K}_a^{u,v}} b_{a,u,v}^k \leq B_u^{out}$,
- $\forall 1 \leq u \leq p,\ \sum_{v=1}^{p} \sum_{a=1}^{A} \sum_{k \in \mathcal{K}_a^{v,u}} b_{a,v,u}^k \leq B_u^{in}$.

We also consider *interval mappings without reuse*, which partition the stages of each (original) application into intervals, and map each interval onto a different processor. More precisely, if we transform each application a as explained above, the allocation function of stages $\hat{\mathcal{S}}_a^k$ (for $1 \leq a \leq A$ and $1 \leq k \leq m_a$) is a one-to-one function: each coalesced stage is allocated onto a distinct processor. It becomes then much easier to check the validity of the mapping, since each processor is only handling one single stage, receiving input data from one single other processor, and sending output data to one single other processor. In other words, previous inequalities become much simpler.

2.4 Energy model

The energy consumption of the platform is defined as the sum of the energy $E(u, \ell)$ consumed by each processor $\mathcal{P}_u$ enrolled in the mapping in mode ℓ. We assume that $E(u, \ell)$ consists of a static part and of a dynamic part. The static part $E_{stat}(u)$ is the static cost for a processor to be in service, and does not depend on the speed $s_{u,\ell}$ at which the processor is running. However, the static energy is consumed only in mode $\ell \neq 0$ (otherwise, the processor is inactive, and not enrolled in the mapping). On the contrary, the dynamic part $E_{dyn}(u, \ell)$ is of the form $E_{dyn}(u, \ell) = s_{u,\ell}^{\alpha}$, where $\alpha > 1$ is an arbitrary rational number. It is sometimes assumed that $\alpha = 2$ [12], but all our results hold for any value of α. Finally, for $\ell \neq 0$, we have $E(u, \ell) = E_{stat}(u) + E_{dyn}(u, \ell)$, while $E(u, 0) = 0$.

The energy $E(u, \ell)$ is an energy consumed per time unit, so we could also speak of dissipated power. Note that it is mandatory to minimize energy consumption per time unit, because the execution of streaming applications with arbitrarily many data sets may last for an unbounded amount of time.

2.5 Problem definition

We consider the problem in which the applications and their characteristics (stage weights, communication costs, periods) are provided, as well as a target execution platform and its characteristics (multi-modal processor speeds, network card capacities and link bandwidths). Then, given a bound on the latency for each application, we aim at minimizing the power consumption while matching the period and latency constraints. Therefore, we formally define the problem as follows:

Definition (TriCriteria($E[P_a, L_a]$)). *Given A applications, p multi-modal processors, one array of periods $[P_a]$ and one array of latencies $[L_a]$, both of length A, what is the minimum power consumption of the platform, so that for each $a \in \{1, \ldots, A\}$, application a is processed at a period P_a, and its latency does not exceed L_a?*

3 Complexity study

We first provide results for interval mappings without reuse, exhibiting dynamic programming algorithms for fully homogeneous platforms, even with several concurrent applications (and the problem was known to be NP-complete on processor-heterogeneous platforms). Then, we establish the NP-completeness of the tri-criteria problem for general mappings with reuse, even on homogeneous platforms.

3.1 Interval mappings without reuse

3.1.1 With one application

Theorem 1. TriCriteria*($E[P_a, L_a]$) is polynomial for interval mappings on processor-homogeneous platforms with one single application.*

Proof. Let n be the number of stages of the single application, P_{giv} be the given period, and L_{giv} be the given latency. First of all, note that the latency is given by $L = (2m-1) \times P_{\text{giv}}$, where m is the number of intervals. Therefore, we can compute a priori the maximum possible number of intervals in the mapping. Let $m^{\max}$ be this number; note that it cannot exceed n, the total number of stages, nor p, the number of processors: $m^{\max} = \min(n, p, \lfloor (\frac{L_{\text{giv}}}{P_{\text{giv}}} + 1)/2 \rfloor)$. If we use more intervals, the bound on the latency will be exceeded. Otherwise, we just have to check if the period constraint is fulfilled.

We exhibit a dynamic programming algorithm that returns the optimal energy consumption. We compute recursively the value $E(i, j, q)$, which is the optimal energy consumption that can be achieved by any interval-based mapping of stages $\mathcal{S}^i$ to $\mathcal{S}^j$ using exactly q processors. The goal is to determine $\min_{m \in \{1,\ldots,m^{\max}\}} E(1, n, m)$. The recurrence relation can be expressed as:

$$E(i, j, q) = \min_{i \leq \ell \leq j-1} \left(E(i, \ell, q-1) + E(\ell+1, j, 1) \right),$$

with the initialization:

- $E(i, i, q) = +\infty$ if $q > 1$ (we cannot run one stage with many processors);
- $E(i, j, 1) = \begin{cases} \min \mathcal{F}^{i,j} & \text{if } \mathcal{F}^{i,j} \neq \varnothing \\ +\infty & \text{otherwise,} \end{cases}$
  where $\mathcal{F}^{i,j} = \left\{ E_{dyn}(s) + E_{stat}, \max\left(\frac{\delta^{i-1}}{b}, \frac{\sum_{k=i}^{j} w^k}{s}, \frac{\delta^j}{b} \right) \leq P_{\text{giv}}, s \in \mathsf{S} \right\}$.

Since the platform is homogeneous, we denote by E_{stat} the static energy of all processors, and by $E_{dyn}(s)$ the dynamic energy consumed at speed s ($s \in \mathsf{S}$). Then, the recurrence is easy to justify: to compute $E(i, j, q)$, we create an interval from stages $\mathcal{S}^{\ell+1}$ to $\mathcal{S}^j$ that is assigned to one single processor, and we use the $q-1$ remaining processors to process stages $\mathcal{S}^i$ to $\mathcal{S}^\ell$. The initialization states that one single stage cannot be run on exactly more than one processor, and it returns the energy consumed by the processor in charge of interval $[i, j]$ so that the bound on the period is satisfied.

□

3.1.2 With many applications

Theorem 2. TriCriteria*($E[P_a, L_a]$) is polynomial for interval mappings on processor-homogeneous platforms.*

Proof. For $a \in \{1, \ldots, A\}$ and $q \in \{0, \ldots, p\}$, let E_a^q the minimum energy consumed by q processors on the application a, computed by one the previous dynamic programming algorithms. If the period constraint cannot be fulfilled, or if the latency constraint cannot be fulfilled ($q > k_a^{\max}$), we set $E_a^q = +\infty$.

We recursively compute the value $E(a, q)$, which is the minimum energy consumed by exactly q processors on applications $1, \ldots, a$. The goal is thus to compute $\min_{1 \leq q \leq p} E(A, q)$. The recurrence relation can be expressed as:

$$E(a, q) = \min_{1 \leq r \leq q-1} \left(E(a-1, q-r) + E_a^r \right),$$

with the initialization:

$$E(1, q) = E_1^q, \forall 1 \leq q \leq p.$$

Indeed, when there is only one application left, the result is known from the previous dynamic programming algorithm. For several applications, we try to assign r processors to application a, and find the value of r which returns the lowest energy consumption. □

Remark. On processor-heterogeneous platforms, the problem of finding an interval mapping which minimizes the power consumption for a given period and a given latency by application is NP-complete. Indeed, the problem of finding an interval mapping which minimizes the period of one single application for processor-heterogeneous platforms without communication cost already is NP-complete [4].

3.2 General mappings with reuse

Theorem 3. TriCriteria*($E[P_a, L_a]$) is NP-complete on processor-homogeneous platforms for general mappings with reuse.*

Proof. We consider the associated decision problem: given periods P_a, latencies L_a ($1 \leq a \leq A$) and an energy E, does there exist a general mapping such that, for all $a \in \{1, \ldots, A\}$, application a is processed at period P_a, its latency is not larger than L_a, and the total energy does not exceed E?

- The problem is obviously in NP: given periods, latencies, an energy and a mapping, it is easy to check in polynomial time that the mapping is valid.
- To establish the completeness, we use a reduction from 2-PARTITION [8]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given n strictly positive integers $x_1, x_2, \ldots, x_n$, does there exists a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} x_i = \sum_{i \notin I} x_i$? Let $S = \sum_{i=1}^n x_i$.

We build an instance $\mathcal{I}_2$ of our problem with 2 identical processors, each with a single possible speed $s = S/2$, and we consider that the cost of static energy is null. We have then n single-stage applications, whose stage weights are x_a, $1 \leq a \leq n$. We ask whether it is possible to achieve an energy $E^o = 2 \times (S/2)^\alpha$, with periods of 1, and latencies not exceeding 1. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$.

We now prove that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does. Assume first that $\mathcal{I}_1$ has a solution. For each $a \in I$, the stage of application a is executed by the first processor. Other stages are executed by the second processor. The mapping consumes an energy $2 \times (S/2)^\alpha$, and all applications have a period and latency equal to 1. Now assume that $\mathcal{I}_2$ has a solution. Since $\sum_{i=1}^{n} x_i = S = S/2 + S/2$, all the periods must be 1, and each processor must run an amount of work of size exactly $S/2$; in other words, $\mathcal{I}_1$ has a solution.

□

4 Experiments

In this section, we first propose an integer linear program which allows us to solve the tri-criteria problem with or without processor reuse, even on fully heterogeneous platforms. However, this program has a prohibitive execution time for large platforms (it may run in exponential time). Therefore, we propose some polynomial-time heuristics in Section 4.2. For small problem instances, we evaluate the absolute performance of the heuristics with respect to the optimal solution returned by the integer linear program, while for large problem instances we have to restrict to a relative comparison of their performances (see Section 4.3).

4.1 Integer linear program

This section provides an integer linear program which gives the exact solution to TriCriteria($E[P_a, L_a]$). Although we expect its cost to restrict its use to small problem instances, this program allows us to assess the absolute performance of the heuristics introduced in Section 4.2 on these instances. The optimization problem includes parameters to describe the applications and the platform, and constraints, as for instance those on the periods. The linear program assigns variables so that they fulfill all constraints, and so that the objective function (the energy) is minimized. We observe that for a given application we can compute the maximum possible number of intervals, given the latency threshold of this application, before calling the linear program.

4.1.1 Parameters

Applications: For all $a \in \{1, \ldots, A\}$, we note $n(a)$ the number of stages in the application a, $P(a)$ its period and $m(a)$ its maximum number of intervals. We add $2A$ fictitious stages $\mathcal{S}_1^0, \ldots, \mathcal{S}_A^0$, $\mathcal{S}_1^{n(1)+1}, \ldots, \mathcal{S}_A^{n(A)+1}$, respectively assigned to processors $\mathcal{P}_{\mathsf{in}_1}, \ldots, \mathcal{P}_{\mathsf{in}_A}$, and $\mathcal{P}_{\mathsf{out}_1}, \ldots, \mathcal{P}_{\mathsf{out}_A}$.

Stages: For all $a \in \{1, \ldots, A\}$ and $k \in \{0, \ldots, n(a)+1\}$, let $w(a,k)$ be the weight of stage $\mathcal{S}_a^k$, and, if $k \neq n(a)+1$, let $\delta(a,k)$ be the output data of stage $\mathcal{S}_a^k$.

Processors: We denote by $\mathcal{IO}$ the index set of input and output processors (hence $\mathcal{IO} = \{\mathsf{in}_1, \ldots, \mathsf{in}_A\} \cup \{\mathsf{out}_1, \ldots, \mathsf{out}_A\}$), and by $\mathcal{NIO}$ the index set of the other processors (with $|\mathcal{NIO}| = p$). We also assume that there is an order on $\mathcal{NIO} \cup \mathcal{IO}$. Each processor $\mathcal{P}_u$, for $u \in \mathcal{NIO}$, has an input (resp. output) network card capacity of $B^{in}(u)$ (resp. $B^{out}(u)$),

and a static energy $E_{stat}(u)$. It can be in $m(u)+1$ different modes. Its speed in mode ℓ, where $\ell \in \{0,\dots,m(u)\}$, is $s(u,\ell)$; mode 0 corresponds to the inactivity of the processor (and thus $s(u,0)=0$); therefore, for $\ell \in \{1,\dots,m(u)\}$, the power consumption of $\mathcal{P}_u$ in this mode is $E(u,\ell) = E_{stat}(u) + s(u,\ell)^\alpha$, while $E(u,0)=0$ (no energy consumption when inactive). The link bandwidth between processors $\mathcal{P}_u$ and $\mathcal{P}_v$, with $(u,v) \in \mathcal{NIO}^2$ and $u \neq v$, is denoted by $b(\min(u,v),\max(u,v))$.

4.1.2 Variables

- For $a \in \{1,\dots,A\}$, $k \in \{0,\dots,n(a)+1\}$ and $u \in \mathcal{NIO} \cup \mathcal{IO}$, $x_{a,k,u}$ is a boolean variable equal to 1 if stage $\mathcal{S}_a^k$ is assigned to processor $\mathcal{P}_u$; we have $x_{a,0,\mathsf{in}_a} = x_{a,n(a)+1,\mathsf{out}_a} = 1$, and $x_{a,k,\mathsf{in}_a} = x_{a,k,\mathsf{out}_a} = 0$ for $a \in \{1,\dots,A\}$ and $1 \leq k \leq n(a)$. We also have $x_{a,k,\mathsf{in}_{a'}} = x_{a,k,\mathsf{out}_{a'}} = 0$ for $a \in \{1,\dots,A\}$, $0 \leq k \leq n(a)+1$ and $a' \neq a$.
- For $a \in \{1,\dots,A\}$, $k \in \{0,\dots,n(a)\}$, $(u,v) \in (\mathcal{NIO} \cup \mathcal{IO})^2$, $y_{a,k,u,v}$ is a boolean variable equal to 1 if stage $\mathcal{S}_a^k$ is assigned to $\mathcal{P}_u$ and stage $\mathcal{S}_a^{k+1}$ is assigned to $\mathcal{P}_v$. For all $u \in \mathcal{NIO} \cup \mathcal{IO}$ and $a \in \{1,\dots,A\}$, if $k \neq 0$ then $y_{a,k,\mathsf{in}_a,u} = 0$ and if $k \neq n(a)$ then $y_{k,u,\mathsf{out}_a} = 0$.
- For $u \in \mathcal{NIO}$ and $\ell \in \{0,\dots,m(u)\}$, $z_{u,\ell}$ is a boolean variable equal to 1 if processor $\mathcal{P}_u$ is in the mode ℓ and 0 otherwise.
- For $u \in \mathcal{NIO}$, $a \in \{1,\dots,A\}$ and $k \in \{1,\dots,n(a)\}$, $s_{a,k,u}$ is the computing power given by processor $\mathcal{P}_u$ to compute stage $\mathcal{S}_a^k$.
- For $(u,v) \in (\mathcal{NIO} \cup \mathcal{IO})^2$, $a \in \{1,\dots,A\}$ and $k \in \{0,\dots,n(a)\}$, $b_{a,k,u,v}$ is the allocated part of the link bandwidth between $\mathcal{P}_u$ and $\mathcal{P}_v$ so that $\mathcal{P}_u$ will send the output data of the stage $\mathcal{S}_a^k$ to $\mathcal{P}_v$.

4.1.3 Objective function

We aim at minimizing $E = \sum_{u=1}^{p} \sum_{\ell=0}^{m(u)} z_{u,\ell} \times E(u,\ell)$.

4.1.4 Constraints

- Each processor runs in one and only one mode: $\forall u \in \mathcal{NIO}, \sum_{\ell=0}^{m(u)} z_{u,\ell} = 1$.
- Each stage is assigned to a processor:
  $\forall a \in \{1,\dots,A\}, \forall k \in \{0,\dots,n(a)+1\}, \sum_{u \in \mathcal{NIO} \cup \mathcal{IO}} x_{a,k,u} = 1,$
  $\forall a \in \{1,\dots,A\}, \forall k \in \{0,\dots,n(a)\}, \sum_{(u,v) \in (\mathcal{NIO} \cup \mathcal{IO})^2} y_{a,k,u,v} = 1.$
- By construction:
  $\forall a \in \{1,\dots,A\}, \forall k \in \{0,\dots,n(a)\}, \forall (u,v) \in (\mathcal{NIO} \cup \mathcal{IO})^2, x_{a,k,u} + x_{a,k+1,v} \leq 1 + y_{a,k,u,v}.$
- Each processor does not exceed its computing speed:
  $\forall u \in \mathcal{NIO}, \sum_{a=1}^{A} \sum_{k=1}^{n(a)} s_{a,k,u} \leq \sum_{\ell=0}^{m(u)} z_{u,\ell} \times s(u,\ell).$
- Each processor does not exceed its maximum outgoing and ingoing total communication

volume:

$$\forall u \in \mathcal{NIO}, \sum_{a=1}^{A} \sum_{k=1}^{n(a)} \sum_{\substack{v \in \mathcal{NIO} \cup \mathcal{IO} \\ v \neq u}} b_{a,k,u,v} \leq B^{out}(u),$$

$$\forall u \in \mathcal{NIO}, \sum_{a=1}^{A} \sum_{k=0}^{n(a)-1} \sum_{\substack{v \in \mathcal{NIO} \cup \mathcal{IO} \\ v \neq u}} b_{a,k,v,u} \leq B^{in}(u).$$

- The link capacity is not exceeded between two processors:
  $\forall u \in \mathcal{NIO} \cup \mathcal{IO}, \forall v > u, \sum_{a=1}^{A} \sum_{k=0}^{n(a)} (b_{a,k,u,v} + b_{a,k,v,u}) \leq b(u,v)$.
- Computation time fits in the period (no constraint if stage $\mathcal{S}_a^k$ is not assigned to processor $\mathcal{P}_u$): $\forall a \in \{1,\ldots,A\}, \forall k \in \{1,\ldots,n(a)\}, \forall u \in \mathcal{NIO}, x_{a,k,u} \times w(a,k) \leq P(a) \times s_{a,k,u}$.
- Communication time fits in the period:
  $\forall a \in \{1,\ldots,A\}, \forall k \in \{0,\ldots,n(a)\}, \forall u \in \mathcal{NIO} \cup \mathcal{IO}, \forall v \neq u,$
  $y_{a,k,u,v} \times \delta(a,k) \leq P(a) \times b_{a,k,u,v}$.
- The maximum number of intervals is not exceeded:
  $\forall a \in \{1,\ldots,A\}, \sum_{(u,v) \in \mathcal{NIO}^2,\ u \neq v} \sum_{k=1}^{n(a)} y_{a,k,u,v} \leq m(a) - 1.$

4.1.5 Additional constraints for interval mappings with no reuse

The previous constraints correspond to the problem of general mappings with processor reuse. We can obtain the optimal solution for interval mappings with no reuse, adding two more constraints:

- A processor cannot process two stages of two different applications:

$$\forall a \in \{1,\ldots,A\}, \forall a' \in \{1,\ldots,A\} \setminus \{a\}, \forall k \in \{0,\ldots,n(a)\}, \forall k' \in \{0,\ldots,n(a')\},$$
$$\forall u \in \mathcal{NIO}, x_{a,k,u} + x_{a',k',u} \leq 1.$$

- A processor cannot process two different intervals of the same application:

$$\forall a \in \{1,\ldots,A\}, \forall k \in \{0,\ldots,n(a)\}, \forall k' \in \{k+1,\ldots,n(a)\}, \forall u \in \mathcal{NIO},$$
$$\forall v \in \mathcal{NIO} \setminus \{u\}, \forall v' \in \mathcal{NIO} \setminus \{u\}, y_{a,k,u,v} + y_{a,k',v',u} \leq 1.$$

4.2 Heuristics

In this section, we present several heuristics for mapping streaming applications onto communication homogeneous platforms. The code of these heuristics is available at `http://graal.ens-lyon.fr/~prenaud/Codes/tri-crit.tar`.

We design three main heuristics, each of them including some variants. The first heuristic H1 is a greedy random heuristic, which will serve as a basis for comparison. The second one, H2, tries to assign each application entirely to a processor, and its variant H2-split starts either with the solution of H2 (if H2 has a solution) or assigns all applications to one processor, and then iteratively improves the current solution by splitting applications into several intervals. The last

heuristic H3 changes iteratively the mode distribution until it can find a feasible mapping; the way to change the speeds comes in three variants and the way to choose the mapping comes in two variants: H3 is thus available in six variants.

Except for H2, which does not use the possibility of sharing the processors (one application onto one processor), each of the heuristic variants has two versions, either with or without processor reuse, which will allow us to observe the impact of resource sharing.

In most of the heuristics, for each processor $\mathcal{P}_u$, we keep its possible modes $(s_{u,\ell})$, for $\ell \in \{0, \ldots, m_u\}$, the index ℓ_u of its current mode, and the minimum speed at which the processor must run in order to be able to process all stages that it is currently assigned to without exceeding the bound on the period, s_u^{needed}. When a stage of weight w of application a is assigned to processor $\mathcal{P}_u$, we add w/P_a to s_u^{needed}. When a stage is de-assigned, we perform a subtraction instead of the addition. Finally, the power consumption of the platform is computed from the speeds s_{u,ℓ_u}, where $s_{u,\ell_u} \geq s^{\text{needed}}(u)$ for all processors.

H1: random. At the start, each application a consists of a single interval composed of all its stages. Then we randomly draw $m_a^{\max} - 1$ stages of application a, where $m_a^{\max}$ is the maximum possible number of intervals of application a such that the latency constraint is respected. Each time we draw a new stage, say $\mathcal{S}_a^k$, we create a new interval by splitting the interval containing $\mathcal{S}_a^k$ just after $\mathcal{S}_a^k$, thus generating one new interval. If a stage is drawn more than once, no new interval is created, so that the final number of intervals will lie between 1 and $m_a^{\max}$, and the latency will never be exceeded. Then we assign each interval to a random processor, without any consideration on the modes of the processors. In the "no-reuse" version of H1, a processor is assigned at most one interval, whereas in the "reuse" version, a processor can be assigned several intervals.

Finally, we decide which modes are used: for each processor we choose its first mode large enough to handle with the needed speed, if such a mode exists; if it does not, the heuristic fails. More formally, for each processor $\mathcal{P}_u$, ℓ_u is the lowest index such that $s_{u,\ell_u} \geq s_u^{\text{needed}}$ if it exists; otherwise, the heuristic fails.

H2: one-to-one. This heuristic assigns each application to one single processor. This problem corresponds to the well-known assignment problem, and we implement the Hungarian algorithm (see [14, 7]) to solve it. The rows represent the processors, and the columns the applications. We do not have to take care of the latency constraint, because one interval by application is the best assignment from the latency perspective. For the processor $\mathcal{P}_u$ and the application a, the corresponding element of the matrix (row numbered u, column numbered a) is the smallest energy which allows the processor to run the application, if possible, and $+\infty$ otherwise.

H2-split: one-to-one with split. We first try to assign each application to one processor by calling H2. If H2 is successful, each application is assigned to one processor, and H2 finds which application to assign to which processor. We perform this assignment. The processors that are not assigned to any application are set in their mode 0. If H2 fails, we assign all application stages to the first processor of the list. If it has enough speed to execute all applications within the period bound, then ℓ_u is the smallest mode such that $s_{u,\ell_u} \geq s_u^{\text{needed}}$. Otherwise, we set $\ell_u = m_u$, but the period is not satisfied in this case.

Therefore, at this point, all the stages are assigned (and we can consider, if the applications are concatenated, that each processor is assigned an "interval"), but this mapping may not be valid: there might be a processor $\mathcal{P}_u$ such that $s_{u,\ell_u} < s_u^{\text{needed}}$.

The main idea of this heuristic is then to try to split each "interval" at any place, and to keep the best split. More precisely, a split consists in:

1. de-assigning one part of the concerned "interval";
2. assigning it to another processor $P_{u'}$;
3. updating the two concerned modes ℓ_u and $\ell_{u'}$ as mentioned previously, thanks to the new values s_u^{needed} and $s_{u'}^{\text{needed}}$.

Then we have to define a way to sort the different resulting mappings in order to choose the best one. The first thing we expect from a mapping is that it respects the period and latency bounds; once we have valid mappings with respect to these performance criteria, the best one is the one whose power consumption is the lowest. Finally, when two mappings lead to the same power consumption, we choose the one in which we are likely to spare the most energy giving the less speed to the new processor during the next split. This is why we finally sort the mappings by:

1. increasing $\sum_{u=1}^{p} \max(s_u^{\text{needed}} - s_{u,\ell_u}, 0)$: the mapping is valid if and only if this value is equal to 0;
2. increasing energy of the platform, that is increasing $E = \sum_{u \in \{1,\dots,p\}} E(u, \ell_u)$;
3. decreasing
$$\max \left\{ \frac{E(u,\ell_u) - E(u,\ell_u - 1)}{s_u^{\text{needed}} - s_{u,\ell_u - 1}} | u \in \{1,\dots,p\}, \ell_u \neq 0 \right\}.$$

While we find a better mapping, we try another split. More formally, the heuristic is detailed in Algorithm 1. In the "no-reuse" version, the processor added in a split cannot be assigned another non-adjacent interval, whereas there is no constraint in the "reuse" version.

Algorithm 1: H2-split(PI)

```
/* PI represents a problem instance, i.e. a platform/applications pair */
Run H2 on PI
PIBest ← PI
repeat
    PI ← PIBest
    PIBU ← PIBest
    forall application in PI do
        if the latency authorizes a split then
            forall interval in the application do
                foreach processor p that is not assigned the interval do
                    foreach stage s in the interval do
                        Assign s to p
                        if PI is better than PIBest then
                            PIBest ← PI
                    PI ← PIBU
until PIBest is better than PI
return PIBest
```

H3: increasing speeds. We start with all processors in their smallest mode. Then we map applications onto the current platform (Algorithm 2), and check whether the mapping is valid or not. If the algorithm returns true, then we are done. Otherwise, we repeatedly change the distribution of the modes and call Algorithm 2 until we find a valid mapping. There are different ways to change the distribution of the modes, thus leading to different variants of the heuristic (see below for variants speed, energy and upDown).

We briefly explain Algorithm 2: the mapping procedure is quite different from that of previous heuristics. Indeed, we never assign a stage to a processor if it has not enough speed to run it while not exceeding the bound on the period. In other words, H3 never allows $s_{u,\ell_u} < s_u^{\text{needed}}$. In the previous heuristics, we first decided for the mapping, and then we chose the modes. In H3, we first choose the mode of each processor, and then we try to find an assignment which is valid with these modes, which may either success or fail.

Algorithm 2: H3-mapping

```
for a ← 1 to A do
    h_a ← Σ_{i=1}^{n_a} w_a^i / k_a^max
Sort the applications by decreasing h_a in L
forall application a in L do
    k ← k_a^max
    Sort the processors by decreasing remaining speed
    while all stages are not assigned and k > 0 and the processors list is not empty do
        Assign the longest interval from the first unassigned stage to the first processor
        Remove the first processor from the list
        k ← k − 1
    if all stages are assigned then
        De-assign the last interval and assign it to the last possible processor
    else
        return false
return true
```

H3-sort: application sorting. This heuristic proposes a modification in the H3-mapping procedure, in which we re-sort the applications after each interval assignment. In H3, we first sort all the applications, and, application by application, we choose a processor and assign it the longest possible interval. If all stages are not assigned, we choose another processor and try to assign the next stages, until the whole application is assigned. In H3-sort (see Algorithm 3), after the first interval assignment, we find the new place of the application in the sorted list, considering this application as if the assigned stages would not exist and if there would be one less possible interval in the application (for the latency constraint). Then we iterate.

This heuristic also comes with variants in the way of changing the distribution of modes.

H3-speed/energy/upDown. We detail now the three variants, which are used for both H3 and H3-sort:

- **speed:** the processors are sorted by increasing speed of the current mode (and if there is a tie by increasing speed gain between the current mode and the next higher one). We check whether function H3 finds a solution; if yes, we stop, and if not, we upgrade the first processor (taken in the previous order) and repeat.

Algorithm 3: H3-sort-mapping

```
for a ← 1 to A do
    h_a ← Σ_{i=1}^{n_a} w_a^i / k_a^max   (S_a^i is unassigned)
Sort the applications by decreasing h_a in L
while L is not empty do
    Pick and remove the first application a in L
    k_a^max ← k_a^max − 1
    Sort the processors by decreasing remaining speed
    Assign the longest interval from the first unassigned stage to the first processor
    if all stages are assigned then
        De-assign the last interval and assign it to the last possible processor
    else
        if k_a^max = 0 then
            return false
        else
            Update h_a and place the application a in L
return true
```

- **energy:** the processors are sorted by increasing energy spent (which is different from an ordering based on modes because of static energy). Again, if there is a tie, we refine the sort according to increasing speed gain between the current mode and the next higher one. We stop when, after upgrading, function H3 returns true.

- **upDown:** We use the same ordering of processors as in the "energy" variant, but we improve the upgrade. The main idea is that if processor modes are distant from each other, the total available speed increases a lot at each upgrade. In this variant we ensure that the total available speed is increasing at each step, but try to increase it slowly. To do that, before every upgrade, we downgrade the mode of the last upgraded processor, if the total available speed is still increasing.

Summary of heuristics. Each heuristic is denoted by its heuristic number, followed by variants. For instance, H3-sort-speed is the H3-sort heuristic with the speed variant. Also, we add "-n" at the end of the heuristic name for the "without reuse" version of the heuristic, and "-r" for the "with reuse" version. Thus, H2-split-n is the H2-split heuristic with no reuse.

Finally we consider another heuristic, called the "best" heuristic, which simply takes the minimum energy returned by all the heuristics. Of course this value is achieved by different heuristics over all experiments, but it helps quantify what can be achieved in polynomial time vs. the linear program.

4.3 Experimental results

We have performed a comprehensive set of experiments in order to: (i) assess the absolute performance of the heuristics, (ii) analyze the impact of reusing resources (interval vs. general mappings), and (iii) study the scalability of the heuristics. We run two experiments for each of these goals.

In the first two experiments, we compare the heuristics with the linear program that finds the optimal general mapping (denoted as cplex-r), whereas in the following two ones, we use the linear program in its "without reuse" version (cplex-n). In both cases, since the integer linear program runs in exponential time and can be very time consuming, we restrict the experiments to a small set of small platforms. On the contrary, we do not launch the linear program for the last two experiments, which allows us to deal with larger applications and platforms.

4.3.1 Experimental setup

We first present the experimental setup for the first four experiments, in which we run the linear program, and finally we describe the last two ones, in which we run only the heuristics.

With the linear program

In each experiment, we generate a set of 30 random platforms and applications. In Experiment 1, we explore the behavior of the heuristics when the number of possible intervals is increasing, while in Experiment 2, we increase the number of processors, in order to confirm that the (best) heuristics stay close to the optimal solution. In Experiments 3 and 4, we respectively vary the maximum static energy and the average gap between two consecutive modes in order to observe the impact of resource sharing.

For each platform, and each value of the parameter that we vary, we run all heuristics, and compute the solution of the linear program using the CPLEX software [5]. Then, for each value of the parameter, and for each heuristic, we sum up (over the platforms) the inverse of the consumed energy returned by the heuristic. If the heuristic fails, we add zero. We plot on a graph the latter sum as a function to the changing parameter. So the higher the curve, the better the heuristic. Finally, we normalize each plot by the optimal solution returned by the linear program. In other words, we show the gap that separates each heuristic from the optimal solution.

Platform sizes are chosen so that the optimal solution can be found in reasonable time (each graph has been obtained within a week, and the execution time of each heuristic was under 1 second per trial). Unless mentioned otherwise, we use those following settings in the experiments. We have 3 applications, each composed of 5 to 11 stages, whose weights vary from 5 to 9. The communication costs between stages are also ranging from 5 to 9. The latency threshold is such that 3 intervals are allowed within each application. The platform consists in 6 to 8 processors, and each of these processors has between 2 and 8 different modes. The distribution of the modes is a Gaussian law centered in 5, and the speeds are chosen between 0 and 50. The static energy of each processor is randomly drawn between 0 and 200.

In Experiment 2, we have only 2 applications with 9 to 15 stages each, and the speeds are drawn between 0 and 90, so that one processor can compute all stages. In Experiment 3, only 4 to 6 processors are available, because the problem becomes untractable starting from 7 processors. Processor speeds are drawn between 0 and 80. In these experiments, we do not represent the "sort" variant of H3, because it leads to negligible variations compared to H3.

Without the linear program

In each of these large-scale experiments, we generate a set of 5000 random platforms, since the running time of the heuristics is negligible. Experiment 5 illustrates the global behavior of the heuristics when the number of applications and processors increases, whereas Experiment 6 studies more precisely their characteristics for some large instances.

In Experiment 5, each application is composed of 15 stages, whose characteristics are the same as previously, 3 intervals are authorized within an application, and the processors have 8

modes distributed between 0 and 80. The applications of Experiment 6 are defined similarly, but this time, the processors have 10 modes, distributed between 0 and 100. For each trial, we draw between 8 and 13 applications, and between 30 and 40 processors.

4.3.2 Comparison with the optimal solution

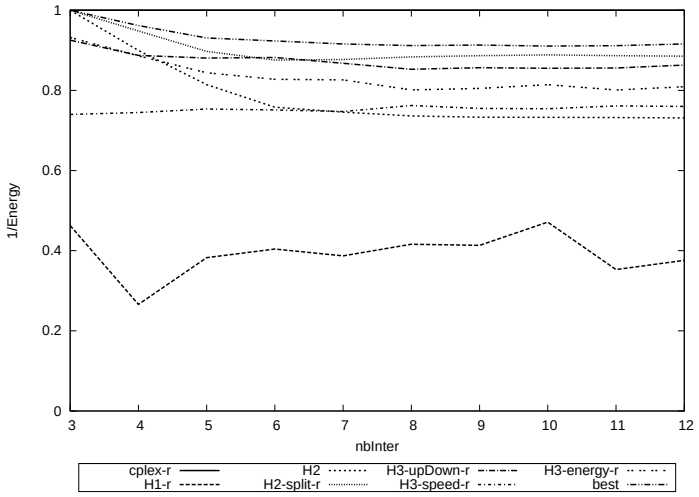

Figure 2: Experiment 1

Experiment 1: Latency

In this first experiment, we vary the latency of the applications: at the beginning, the latency constraint imposes that each application is mapped as a single interval, while it can go up to four in the end. All the heuristics are run in their "with reuse" variants. This experiment gives us a first idea of the ordering of the heuristics: the "upDown" is the best variant of the heuristic H3, before "energy" and "speed". The "speed" variant is the only one which is not better with fewer intervals by application, because it does not choose the processors whose static energy is low.

Heuristic H2-split is the best heuristic on average, but for some platforms, H3-upDown is better. The best heuristic is always at 0.9 of the optimal solution. As expected, H2 finds the optimal solution when one single interval is authorized in each application. Then its performance decreases as soon as two intervals are allowed in each application. Finally, it remains approximately constant at 0.7 of the optimum. Without much surprise, heuristic H1 is worse than the others, therefore demonstrating that a random approach does not provide satisfying results.

Experiment 2: Processor number

In this second experiment, we increase the number of processors for a given application. H2 does not reuse processors, thus it does not find the solution with one processor. Then, with more than two processors, its efficiency decreases when the number of processors increases. As in the first experiment, H3-upDown-r and H2-split-r return the best results, depending upon the platform. Moreover H2-split-r is the best in average if and only if the processor number is not greater than 6. However, the "energy" and "speed" variants of H3 are always worse than H2-split-r in average. The "speed" variant becomes very bad, because when the number of available

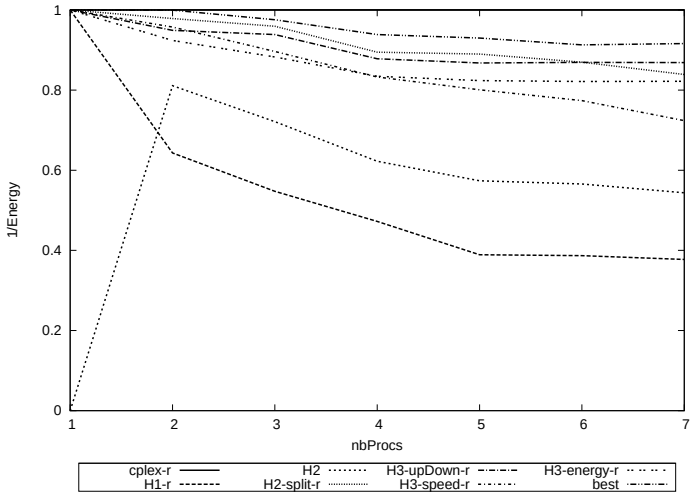

Figure 3: Experiment 2

processors is increasing, these processors are used in their lowest mode, and the static part of the energy becomes crucial. Finally, the "best" heuristic is quite good, never below 0.92 of the optimal.

4.3.3 Impact of reuse

In this second set of experiments, we compare the heuristics to the optimal solution without reuse, in order to assess the impact of reuse on the mapping.

Experiment 3: Static energy

In this third experiment, we vary the maximum static energy, that can be drawn from 0 to 2400. When the static energy is becoming high, it is advantageous to use fewer processors. For variants "without reuse", this leads to one processor per application. That is why H2 and H2-split-n seem to tend to the optimal solution without reuse, when the maximum static energy is increasing.

Processor reuse becomes interesting as soon as the maximum static energy exceeds 400, since the heuristics with reuse perform better than the optimal solution with no reuse. H2-split-r and H3-upDown-r are becoming more and more efficient when the static energy increases, and H2-split-r ultimately reaches 1.15 of the optimal solution without reuse for a static energy of 2400. Processor reuse allows the heuristics to use fewer processors than applications, and thus to spare some static energy cost.

Experiment 4: Mode distribution

In this fourth experiment, we vary the average gap between two modes from 5 to 40. When the modes are not close together, the first mode is high, and the best solution for the "without reuse" variants is reached with one processor per application. As before, H2 and H2-split-n tend to the optimal solution without reuse. This time, as the processors are not very different, H3-upDown-n also gets very close to the optimal solution without reuse.

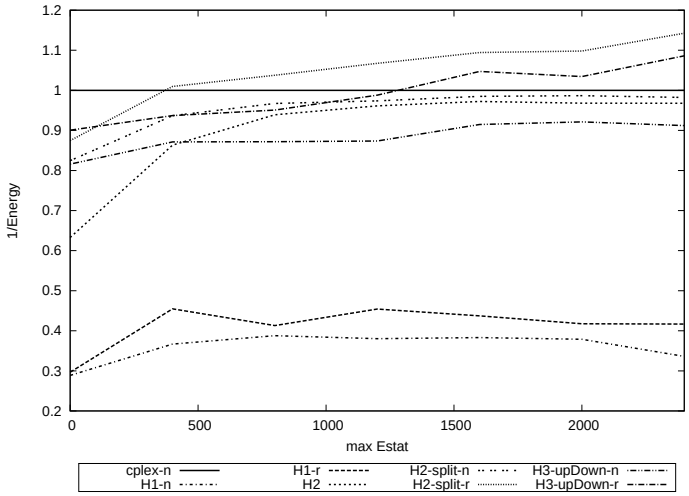

Figure 4: Experiment 3

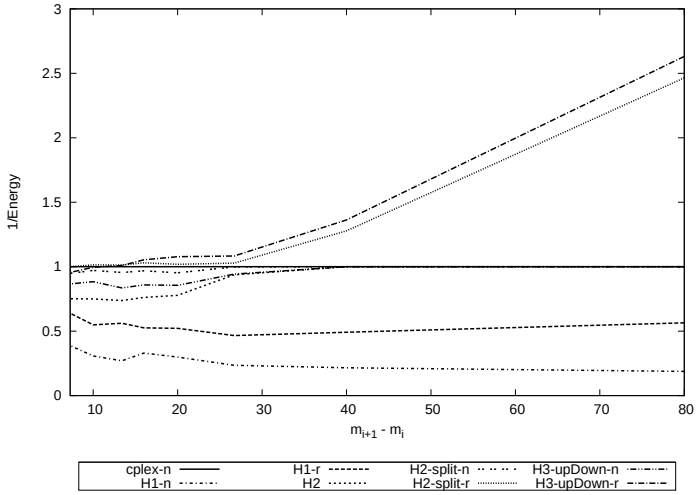

Figure 5: Experiment 4

The heuristics with reuse obtain much better results, in particular when the difference between modes is large. H3-upDown-r is constantly increasing and it is 2.6 times better than the optimal solution without reuse at the end. When the modes are very close, H3-upDown-r is not as competitive as the optimal solution without reuse, but it is still at 0.95. H2-split-r is almost as efficient as H3-upDown-r, and remains better than the optimal solution without reuse when the modes become closer.

More generally, resource sharing becomes interesting when the modes are not close to each other: the reuse allows us to fill up the high modes with stages of different applications.

4.3.4 Scalability

In this last set of experiments, we study the heuristics when the instances are bigger. For such real-life instances of the problem, the integer linear program cannot be used any more, due to its high complexity.

Experiment 5: Global increase

In the fifth experiment, we increase the number of processors with the number of applications, such that there are four times more processors than applications. This time, we represent the energy on the y-axis instead of its inverse, since we cannot normalize the plots with the optimal solution anymore. Therefore, the lower the plot the better the heuristic.

H2-split-r is the best on all platforms when there are many applications, before H3-upDown-r and H3-energy-r, which almost have the same efficiency, and H3-speed-r. The more applications, the better H2-split-r, compared to the other heuristics. But for 20 applications, all heuristics execute in less than 1 second, against 3 minutes for H2-split-r.

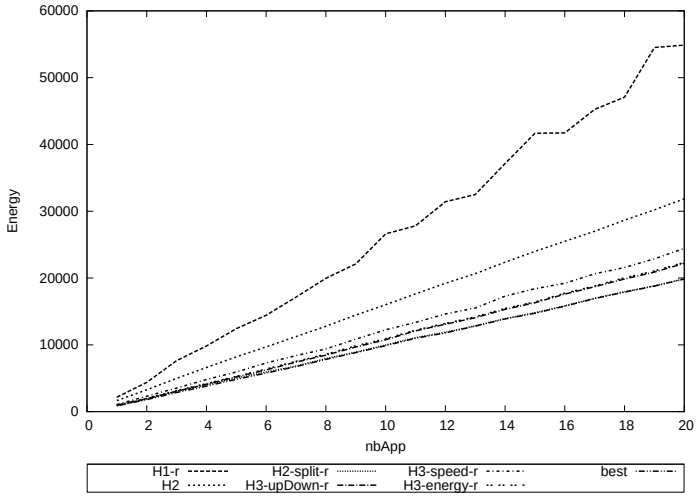

Figure 6: Experiment 5

Experiment 6: Complete comparison

In this last experiment, we study all heuristics for some large problem instances. The main characteristics of the heuristics are shown in Table 1. We report the number of failures in the first column, and how many times the concerned heuristic has been the best one in the second

column. For the last four columns, we normalize the power consumption found by each heuristic by the power consumption found by the best one and analyze the table of normalized power. The average is computed with the platforms for which the heuristic found a solution. The column "max" represents the worst case for each heuristic, this is why there is no numeric value for the heuristics which failed.

	Fail/Best	average	min	max
H1-r	114/0	2.625483	1.541424	FAIL
H1-n	286/0	2.570121	1.511875	FAIL
H2	0/0	1.558267	1.257960	1.954439
H2-split-r	0/3710	1.008385	1	1.226330
H2-split-n	0/514	1.022594	1	1.226330
H3-upDown-r	0/164	1.100380	1	1.338159
H3-upDown-n	0/98	1.113033	1	1.504697
H3-speed-r	0/4	1.228998	1	1.974289
H3-speed-n	0/3	1.244661	1	2.180104
H3-energy-r	0/58	1.114920	1	1.374722
H3-energy-n	0/37	1.126718	1	1.504697
H3-sort-upDown-r	0/712	1.056324	1	1.251331
H3-sort-upDown-n	0/62	1.118340	1	1.453929
H3-sort-speed-r	0/37	1.170503	1	1.902925
H3-sort-speed-n	0/5	1.210323	1	2.017221
H3-sort-energy-r	0/239	1.071706	1	1.271649
H3-sort-energy-n	0/25	1.128880	1	1.470972

Table 1: Experiment 6

The random heuristics are the only ones which fail on some drawn platforms, and, as expected, they have the largest variability. H2-split-r is clearly the best heuristic: it finds about four times out of five a better solution than the other heuristics, and when it does not, it is not so bad, because it is on average at 0.8% of the best solution.

The variants "sort" of H3 are significantly better than the regular ones. H3-sort-upDown-r finds the best solution more often than H2-split-n, but it is worse on average. Because they do not evaluate the static energy of the processors, the variants of H3-speed do not avoid the processors with high static energy, thus they have a bigger variability and a worse average than the other variants of H3.

5 Conclusion

In this paper, we have studied the following scheduling problem: given several pipelined applications with period and latency thresholds, determine the mapping on a platform composed of multi-modal processors (and the speed at which each processor should run), in order to minimize the total energy consumed by the platform.

We first established the complexity of this problem for different mapping strategies (interval mappings without reuse and general mappings with reuse), and different platform types (processor-homogeneous and processor-heterogeneous platforms). Thanks to a combination of two dynamic programming algorithms, we showed that finding an optimal interval mapping without reuse on processor-homogeneous platforms can be done in polynomial time. On the contrary, finding an optimal general mapping on any platform type, or finding any optimal

mapping on speed heterogeneous platforms, are NP-complete problems.

We have also been interested in providing polynomial-time solutions for speed heterogeneous platforms. We wrote an integer linear program to compute the optimal solution (either interval-based or general) in possibly exponential time. Then we have designed several heuristics, which we compared to each other, and to the optimal solution found by the linear program on small instances. At least on those small instances, the best heuristic always achieves at least 90% of the best solution. The comparison of heuristics with and without processor sharing does confirm that sharing is more useful when (i) the modes are not close to each other, and (ii) the static energy is high.

As for future directions, we would like to search for approximation algorithms, or to derive inapproximability results. Indeed, even though the performance of the heuristics was experimentally shown pretty good, we have no theoretical guarantee. With the tri-criteria approach of this paper, with thresholds on performance-related criteria, we could formulate the problem as follows: given three parameters α_P, α_L and α_E, does there exist a polynomial algorithm $\mathcal{A}$ such that the energy found by $\mathcal{A}$ on the problem TriCriteria$(E[\alpha_P P_a, \alpha_L L_a])$ is less than α_E times the optimal energy consumption of the problem TriCriteria$(E[P_a, L_a])$? Finding such approximation algorithms for some values of α_P, α_L and α_E seems quite a challenging problem.

References

[1] K. Agrawal, A. Benoit, L. Magnan, and Y. Robert. Scheduling algorithms for workflow optimization. Research Report 2009-22, LIP, ENS Lyon, France, July 2009. Available at `http://graal.ens-lyon.fr/~yrobert/`. To appear in IPDPS'2010.

[2] K. Agrawal, A. Benoit, and Y. Robert. Mapping linear workflows with computation/communic ation overlap. In *ICPADS'2008, the 14th IEEE International Confer ence on Parallel and Distributed Systems*, pages 195–202. IEEE CS Press, 2008.

[3] A. Benoit, P. Renaud-Goud, and Y. Robert. Performance and energy optimization of concurrent pipelined applications. In *International Parallel and Distributed Processing Symposium IPDPS'2010*. IEEE Computer Society Press, 2010.

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

[5] Cplex. ILOG CPLEX: High-performance software for mathematical programming and optimization. `http://www.ilog.com/products/cplex/`.

[6] DataCutter. DataCutter Project: Middleware for Filtering Large Archival Scientific Datasets in a Grid Environment. `http://www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm`.

[7] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22:973–996, 2001.

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

[10] 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*, 2003.

[11] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In *International Parallel and Distributed Processing Symposium IPDPS'2006*. IEEE Computer Society Press, 2006.

[12] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202. ACM Press, 1998.

[13] 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.

[14] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[15] 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.

[16] 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.

[17] 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.