Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms

Changjiang Gou, Anne Benoit, Mingsong Chen, Loris Marchal, Tongquan Wei

Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms

Changjiang Gou, Anne Benoit, Mingsong Chen, Loris Marchal, Tongquan Wei

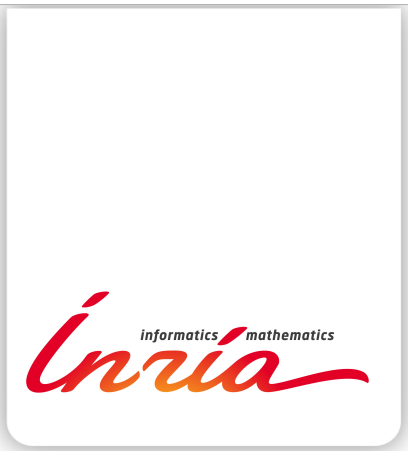
RESEARCH REPORT

N° 9346

June 2020

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9346--FR+ENG



Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms

Changjiang Gou*†, Anne Benoit*, Mingsong Chen†, Loris Marchal*, Tongquan Wei†

Project-Team ROMA

Research Report n° 9346 — June 2020 — 28 pages

Abstract: Streaming applications come from various application fields such as physics, and many can be represented as a series-parallel dependence graph. We aim at minimizing the energy consumption of such applications when executed on a hierarchical platform, by proposing novel mapping strategies. Dynamic voltage and frequency scaling (DVFS) is used to reduce the energy consumption, and we ensure a reliable execution by either executing a task at maximum speed, or by triplicating it. In this paper, we propose a structure rule to partition the series-parallel applications, and we prove that the optimization problem is NP-complete. We are able to derive a dynamic-programming algorithm for the special case of linear chains, which provides an interesting heuristic and a building block for designing heuristics for the general case. The heuristics performance is compared to a baseline solution, where each task is executed at maximum speed. Simulations demonstrate that significant energy savings can be obtained.

Key-words: Mapping, partitioning, scheduling, SPG, reliability, energy.

* LIP Laboratory, ENS Lyon, CNRS, France
† East China Normal University, China

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Placement fiable pour minimiser la consommation énergétique d'applications séries-parallèles sur des plates-formes hiérarchiques

Résumé : Les applications de type "workflow" viennent de divers domaines d'application (en particulier de la physique), et plusieurs applications peuvent être représentées avec un graphe de dépendances série-parallèle (SPG). Le but est de minimiser la consommation énergétique de telles applications lorsqu'elles sont exécutées sur une plate-forme hiérarchique. Pour cela, nous proposons de nouvelles stratégies de placement. Nous utilisons la technique DVFS (*Dynamic Voltage and Frequency Scaling*) pour réduire la consommation énergétique, et nous assurons une exécution fiable en exécutant chaque tâche soit à la vitesse maximum, soit en tripliquant son exécution. Nous proposons une règle pour partitionner les applications tout en gardant la structure série-parallèle, et nous prouvons que le problème d'optimisation associé est NP-complet. Pour le cas particulier des chaînes de tâches, nous proposons un algorithme de programmation dynamique, et une heuristique pour les graphes SPG s'appuie sur cet algorithme. Plusieurs heuristiques sont proposées, et leur performance est comparée à une solution gloutonne où toutes les tâches sont exécutées à la vitesse maximum. Des simulations démontrent que d'importants gains en énergie peuvent être obtenues grâce aux nouvelles heuristiques.

Mots-clés : Placement, partitionnement, ordonnancement, graphes séries-parallèles, fiabilité, énergie.

1 Introduction

Streaming data is continuously generated from applications in high energy physics, astronomy [1] and other scientific or industrial domains [2]. With the improvement of detector resolution, it is anticipated that the data volume will dramatically increase. For instance, the advanced light-source facility could generate 1.9 PB data each year and at a rate of 20 GB/sec in the near future [3]. Processing these data in real-time and then feedback key information to decision-making is critically useful, even if it demands intensive computing power. The use of large-scale hierarchical platforms can help parallelize the processing of this streaming data and process it in real time. This may also help reduce the overall energy consumption resulting from the intensive computing properties, for instance by using Dynamic Voltage and Frequency Scaling (DVFS): by dynamically tuning processor frequencies and voltages, DVFS enables task completions with a lower energy consumption.

Although DVFS techniques can save overall energy, they inevitably result in an increased arrival rate of transient faults [4,5]. This is because modern processors used by streaming applications are based on CMOS technology. Typically, a CMOS processor consists of billions of transistors, where one or more transistors form one logic bit holding binary values of either 0 or 1. Due to physical phenomena such as high energy cosmic particles or rays, the content of some logic bit can be flipped by mistake, resulting in the notorious soft errors. Although checkpointing with rollback-recovery can mitigate soft error effects, frequent utilization of such fault-tolerance mechanism is time-consuming. The unpredictable occurrences of soft errors may result in severe temporal violations. We consider in this study a reliability target not equal to 100%, but instead a small percentage of failures is acceptable, so that tasks running at maximum speed have a sufficient reliability. We also observe that triplicating tasks and performing a majority voting leads to a suitable reliability. This avoids overwhelming energy-consuming on applications that do not need an error-free level of reliability.

This results in a multi-objective optimization problem: mapping streaming applications onto a hierarchical computing platforms with the aim of saving energy, while meeting performance and reliability constraints. Scientific workflows are often represented as Directed Acyclic Graphs (DAGs), which model the computation needs of tasks and dependencies among tasks [6]. Most of the workflows corresponding to streaming applications exhibit a regular structure, such as linear chains, trees, fork-join graphs, or general series-parallel graphs. For instance, most of the StreamIt benchmarks [7] are series-parallel graphs. Hence, we focus on series-parallel applications. The platform on which we aim at executing such applications is a two-level platform, where several blocks, each with several cores, are available.

This paper makes the following major contributions:

1. We propose a formal reliability and energy-aware model for multi-objective optimization of allocation and scheduling of streaming tasks on a hierarchical platform, and prove that the optimization problem is NP-hard;

2. We design a dynamic programming approach for simple linear chains of streaming tasks, based on which allocation and scheduling heuristics for the general case can be built;

3. Extensive simulations on real applications show that our heuristics can achieve energy savings without degradation of performance and reliability, as compared to running all tasks at the maximum speed.

The rest of this paper is organized as follows. After the introduction of related works in Section 2, Section 3 formalizes both application and platform models and defines the MinEnergy optimization problem. Section 4 analyzes the problem complexity. Section 5 presents a dynamic programming-based solution for MinEnergy when dealing with linear chain applications, and Section 6 proposes heuristics for general series-parallel graphs. Section 7 evaluates the proposed algorithms. Finally, Section 8 concludes the paper and provides directions for future work.

2 Related work

In recent years, efficient parallel processing of streaming applications on hierarchical platforms has attracted growing research interest. Considering the calculation and communication cost of directed acyclic graph (DAG), Tang et al. [8] presented two heuristic strategies based on integer linear programming to reduce communication overhead and scheduling length. Flasskamp et al. [9] designed a performance estimator embedded in the compiler to partition and map streaming applications. For an MPSoC system consisting of a multi-core CPU and on-chip GPU, Vilches et al. [10] introduced a novel framework that can adaptively find the best mapping for multiple tasks to achieve better performance. Under real-time requirements and mapping constraints, Onnebrink et al. [11] proposed a DVFS-based effective heuristic algorithm for heterogeneous MPSoC systems to optimize energy consumption. For a set of periodic real-time tasks, Haque et al. [12] designed a static and dynamic two-stage algorithm to reduce the concurrent execution of given task replicas and reduce energy consumption while meeting the given reliability requirements. For heterogeneous real-time MPSoC systems, Zhou et al. [13] designed a two-stage thermal-aware task allocation strategy to optimize energy consumption and peak temperature. Although these works can effectively reduce energy consumption and improve performance, communication costs are not taken into account.

Recently, a considerable number of researchers have focused on improving communication costs, which is also a key factor in determining performance. Khandekar et al. [14] described an iterative algorithm that partitions streaming application graphs to balance the load and minimize the communication costs. Yu et al. [15] proposed a genetic algorithm to map a workflow onto utility grids to minimize execution time under a given budget constraint. In [16], Huang et al. divided the dependent DAG into many parts and mapped each part to the processor to achieve a balance between communication and workload. Wieczorek et al. [17] evaluated the performance of the three scheduling strategy for mapping scientific work onto the grid and the experimental results demonstrate that the HEFT algorithm is optimal for balanced and unbalanced applications. For streaming applications in hierarchical MPSoC systems, Kelly et al. [18] proposed a simulated annealing-based compiler to achieve better performance compared to the most advanced partitioning algorithms. However, these studies ignore the dependence or reliability requirements of streaming tasks.

3 Model

3.1 Applications

The application that is to be scheduled is a streaming application: it operates on a collection of data sets that are executed in a pipelined fashion. The period of the application, which is the inverse of the throughput, corresponds to the time interval between the arrival of two consecutive data sets. We assume that the period of the application (or the throughput) is given by the application and must be enforced. This target period is denoted by P_{t}.

We consider applications represented as a series-parallel graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, or SPG. Nodes of the graph correspond to different application tasks, and are denoted by T_i, with $1 \leq i \leq n$, where $n = |\mathcal{V}|$ is the size of the graph. For each precedence constraint in the application, say from task T_i to task T_j, we have an edge $L_{i,j} \in \mathcal{E}$, and we say that T_j is a successor of T_i, $j \in Succ(i)$. For $1 \leq i \leq n$, w_i is the computation requirement of task T_i, and for each $L_{i,j} \in \mathcal{E}$, with $1 \leq i, j \leq n$, $\delta_{i,j}$ is the volume of communication to be sent from T_i to T_j before T_j can start its computation.

An SPG is built from a sequence of compositions (parallel or series) of smaller-size SPGs, as illustrated in Figure 1. The smallest SPG consists of two nodes connected by an edge. The first node is the source of the SPG while the second is its sink. When composing two SPGs in series, we merge the sink of the first SPG with the source of the second SPG. For a parallel composition, the two sources are merged, as well as the two sinks. The source is also called a *fork* node, and the sink a *join* node.

Data sets arrive at a prescribed rate P_{t}, i.e., a new data set enters the system every P_{t} time units, and we must therefore be able to process at a throughput of at least $\frac{1}{P_{\mathrm{t}}}$. We will further discuss how to compute this processing rate in Section 3.6.

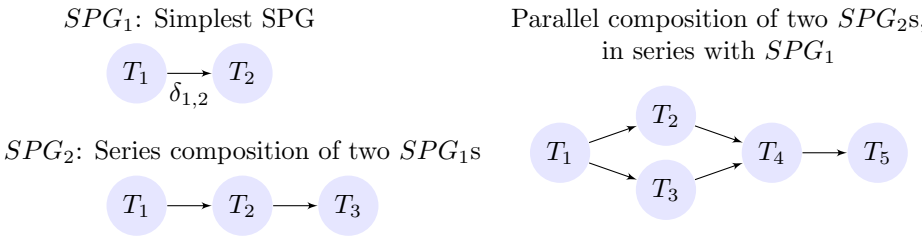


Figure 1: SPG examples.

3.2 Platforms

The computing platform targeted in this work has $c \times p$ homogeneous cores. Each core can run at a different speed, with a corresponding error rate and power consumption. We focus on the most widely used speed model, the discrete model, where cores have a discrete number of predefined speeds, which correspond to different voltages at which the core can be operating. Switching is not allowed during the execution of given tasks. The set of speeds is $\{s_{\min} = s_1, s_2, ..., s_k = s_{\max}\}$.

The cores are organized by a hierarchical communication system. It consists in c blocks, each of them containing p computing cores that are tightly coupled by a low-latency interconnect fabric. To have a system with hundreds of cores, blocks are connected by the next level network, which contains the route-tables and network parity checking logic. Computation and communication can hence process concurrently. The bandwidth between two cores in the same block and in different blocks are denoted respectively as β_1 and β_2. Communication among cores in the same block is cheaper than that among different blocks [19], i.e., $\beta_1 >> \beta_2$.

3.3 Graph partitioning and structure rule

In order to achieve load balance and to save communication, the application is partitioned into several connected parts. Tasks in a part are then allocated to the same core (and a core processes tasks from a single part), hence there is no communication cost to pay between tasks in the same part.

For the ease of the communication pattern, since we consider series-parallel graphs (SPGs), we aim at keeping the SPG structure when creating parts, hence the *structure rule*.

Definition 1 (Structure rule)**.** *A partition of the SPG follows the structure rule if and only if each part consists either of (i) a single task, (ii) a subgraph that is itself an SPG, or (iii) several tasks or SP subgraphs that share the same predecessor and successor (that is, a parallel composition of SP subgraphs).*

If we consider a simple linear chain with three tasks T_1, T_2, T_3, that is, a series composition of these tasks, to be mapped on two cores, this rule does not allow T_1 and T_3 to be mapped on the same core, while T_2 is on another core. Rather, we can either keep the three tasks on one core, or have two consecutive tasks on a core and the third task on another core. For such linear chains, this is similar to *interval mappings* [20].

The rule for parallel compositions is slightly more intricate: consider for instance a simple fork-join with source T_{fork} and sink T_{join} and inner tasks $T_1, \ldots, T_k$, as depicted on Fig. 2. Then, either all tasks of this fork-join are in a same part, or T_{fork} and T_{join} must both be in different parts, and none of inner tasks $T_1, \ldots, T_k$ can be in one of these two parts. However, several of them can be grouped in the same part, as they share the same predecessor T_{fork} and the same

successor T_{join}. For instance, T_1 and T_3 can be in the same part, while all other tasks $T_2, T_4, \ldots T_k$ can be in another part, as depicted in Fig. 2.

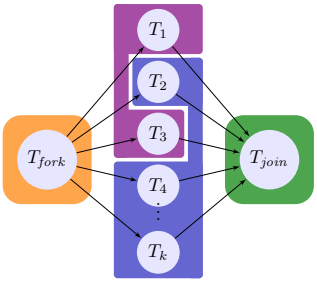


Figure 2: Fork-join graph and a partition following the structure rule.

A parallel composition of more complex subgraphs is depicted in Figure 3 between tasks T_1 and T_{15}. In the proposed partition, two subgraphs of the parallel composition are grouped together (green partition), which is allowed as they share the same predecessor T_1 and successor T_{15}. The other subgraph of this parallel composition is split into two parts. One of them, including T_2 and T_3, is made of two tasks sharing the same predecessor and successor, while the other one is a SP subgraph. Note that by construction, each part of a partition following the structure rule has either a single source vertex and sink vertex (in the cases (i) and (ii) of the definition), or it has a single predecessor and a single successor (case (iii)).

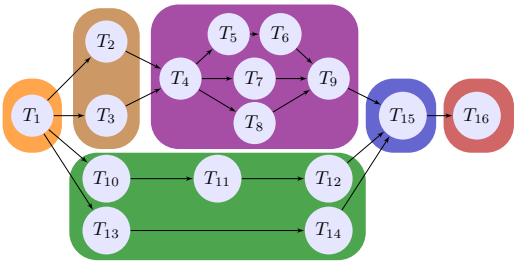


Figure 3: SPG partition following the structure rule.

Notations. The set of task indices that are mapped onto a core v is denoted by C_v, and all these tasks are executing at the same speed $S(v)$. Indices of tasks that are mapped on a block d of cores is denoted as set ℓ_d, and it is the union of the C_v's for all cores v in block d, i.e., $\ell_d = \cup_{v \in d} C_v$.

The sets $Source_v$ (resp. $Sink_v$) represent the indices of the source vertices (resp. sink vertices) mapped on core v. There is only one source and one sink, except for parallel SPGs mapped in a same part. Also, we define the set $PredC_v$ (resp. $SuccC_v$), which contains the core indices on which there are tasks that send outputs to tasks T_i, with $i \in Source_v$ (resp. receive inputs from tasks T_i with $i \in Sink_v$). By construction, either there is only one source and one sink ($|Source_v| = |Sink_v| = 1$), or there is only one predecessor and successor task.

3.4 Soft-errors and triplication

High performance computing platforms are subject to failures, and in particular transient errors caused by radiation. In our framework, we can choose the execution speed of a core. However, a very small decrease of speed leads to an exponential increase of failure rate [4, 5]. Indeed, radiation-induced transient failures follow a Poisson distribution, and the fault rate is given by:

$$\lambda(s) = \lambda_0 e^{d \frac{s_{\max} - s}{s_{\max} - s_{\min}}},$$

where $s \in [s_{\min}, s_{\max}]$ denotes the running speed, d is a constant that indicates the sensitivity to DVFS, and λ_0 is the average failure rate at speed $s_{\max}$. λ_0 is usually very small, of the order of

10^{-5} per hour [21]. Therefore, we can assume that the application is reliable enough when running at speed $s_{\max}$, and that there is no need of re-execution [22].

To save energy while having a reliable execution, we also propose a triplication of tasks: three copies of the same task (or group of tasks) are run simultaneously, and a majority voting determines the correct results. Such a scheme may fail only if two copies (among the three) fail simultaneously. For example, on the processor used for the simulation (see Section 7), and when considering that the failure rate at maximum speed is $\lambda_0 = 10^{-5}$ faults per hour, the failure rate at minimum speed is 5.46×10^{-4} per hour. Then, the probability for at least two copies failing is: $3 \times (5.46 \times 10^{-4})^2 = 8.94 \times 10^{-7}$ failures per hour, which is much smaller than the probability at maximum speed. We continue this example below to show that in some cases, triplication succeeds to reduce the energy consumption.

Therefore, after a partition of tasks is done (following the structure rule), in order to have a *reliable* execution, either we execute a whole part on a core at maximum speed without triplication (denoted by $m_i = 1$ for any task T_i in the part), or we triplicate the whole part on three different cores (denoted by $m_i = 3$ for any task T_i in the part). In this later case, the execution speed $S(v)$ used by the three cores is set to the minimum speed such that $S(v)P_{\mathrm{t}} \geq \sum_{i \in C_v} w_i$, so as to minimize the energy cost while respecting period bound. We further enforce that these three cores must be in the same block, since they need to communicate, in particular to do the majority voting and decide which result is correct. Note that if a part is triplicated, the majority voting occurs only for the last task of the part.

3.5 Energy

We follow a classical energy model, whose power estimation error in a case study is at most 9.4% on average, see for instance [11]. The energy consumption of executing a data item through all tasks is composed of static part and dynamic part: $E = E_s + E_d$. The static component represents the idle leakage current consumption, which is modeled as $E_s = I_s \times V_s \times P_{\mathrm{t}} \times c_a$, where I_s and V_s denote the leakage current and the minimum possible voltage of a core, and c_a denotes the actual number of cores used, since we assume that other cores can be switched off. Since a data item arrives every P_{t} time units, the static energy is consumed during a time P_{t} for each task, on each of the c_a cores.

For a single execution of task T_i running at speed $s(i)$, the dynamic component E_d^i is related to the operating frequency and voltage, $E_d^i = Cs^3(i) \times \frac{w_i}{s(i)} = Cw_i s^2(i)$, in which C denotes the switching capacitance. The supply voltage is scaled in almost linear fashion with the processing frequency [12]. After taking triplication into consideration, the energy cost of the whole application on one data item is therefore:

$$E = I_s V_s P_{\mathrm{t}} c_a + C \sum_{1 \leq i \leq n} m_i w_i s^2(i),$$

where $m_i = 3$ if T_i is triplicated, otherwise $m_i = 1$. Following up with the previous example, we show that triplicating a task may cost less energy than running it at the maximum speed. We use the values used in Section 7: $s_{\min}$ and $s_{\max}$ are 1.2 Ghz and 4 Ghz respectively, static power is 2W, $C = 1$. Assume that the task's weight is 1.2 and the period is 1 second. The energy needed for triplicating it at $s_{\min}$ is $3 * (2 + 1.2 * 1.2^2) = 11.184W$, while running it at $s_{\max}$ requires $2 + 1.2 * 4^2 = 21.2W$.

The energy cost of the communication is not negligible in our model. Within a block, communication among processor cores is realized through a remote memory access. Communication between two cores of different blocks is realized by routers on NoC. For a simple transfer of data on edge $L_{i,j}$, the energy cost can be represented by $E_c(L_{i,j}) = \alpha_{i,j}\delta_{i,j}$, where $\alpha_{i,j}$ is the energy cost for a unit of data sending. $\alpha_{i,j}$ depends on where tasks are located: if task T_i and T_j are allocated onto the same core, then $\alpha_{i,j} = 0$; $\alpha_{i,j} = \alpha_1 > 0$ if tasks are allocated onto two cores of the same block; otherwise $\alpha_{i,j} = \alpha_2$, and $\alpha_1 < \alpha_2$, see [23] for details.

Also, we must consider the influence of triplication. Given $L_{ij} \in \mathcal{E}$ such that $\alpha_{i,j} \neq 0$, i.e., T_i and T_j are mapped on different cores, the energy cost also depends on whether T_i and T_j are

triplicated or not. First, if T_i is triplicated, it does a majority voting before the communication occurs: two outputs from two different cores need to be sent to a core in the same block, hence the energy cost is $(m_i - 1)\alpha_1\delta_{i,j}$ (hence this cost is null if $m_i = 1$). Next, the communication between T_i and T_j must be done one or three times, depending whether T_j is triplicated or not, with a cost $m_j\alpha_{i,j}\delta_{i,j}$.

In total, the energy cost of the whole application on one data set is:

$$E = I_s V_s P_t c_a + C \sum_{1\leq i\leq n} m_i w_i s^2(i) + \sum_{L_{i,j}\in\mathcal{E}|\alpha_{i,j}\neq 0} ((m_i - 1)\alpha_1\delta_{i,j} + m_j\alpha_{i,j}\delta_{i,j}).$$

3.6 Timing definition and constraints

The actual time spent by tasks mapped on core v is:

$$T(v) = \max\Big(\frac{\sum_{i\in C_v} w_i}{S(v)} + (m_i - 1)\sum_{j\in Sink_v}\sum_{k\in Succ(j)}\frac{\delta_{j,k}}{\beta_1},$$
$$\max_{u\in SuccC_v}\sum_{j\in Sink_v, k\in Source_u}\frac{\delta_{j,k}}{\beta_{v,u}},$$
$$\max_{u\in PredC_v}\sum_{j\in Sink_u, k\in Source_v}\frac{\delta_{j,k}}{\beta_{u,v}}\Big),$$

where $\beta_{u,v} = \beta_{v,u}$ since communication channels are symmetrical, $\beta_{u,v} = \beta_1$ if u and v are on the same block, otherwise $\beta_{u,v} = \beta_2$. If tasks in C_v are triplicated, then $m_i = 3$, otherwise $m_i = 1$.

The first term in the maximum is the execution time plus the time required for majority voting if tasks are triplicated. Indeed, in this case, two copies of all outputs from task T_j, with $j \in Sink_v$, need to be sent to a core in the same block, since they are sent to the same place. The communication is sequentially executed to avoid potential contention, thus the time needed is two times ($m_i - 1 = 2$ in this case) a single transfer. The second and third terms are the time needed to send and receive datasets.

To execute a data item through all stages of $\mathcal{G}$, the time taken is therefore:

$$T(\mathcal{G}) = \max_{1\leq v\leq cp} T(v).$$

In order for the mapping to be valid, this has to be less than or equal to the target period:

$$T(\mathcal{G}) \leq P_t.$$

3.7 Optimization problem

The objective is to minimize the expected energy consumption per dataset of the whole workflow, while ensuring a reliable execution of the application. Hence, each task should either be executed at maximum speed, or triplicated. The goal is hence to decide which tasks to group in a same part, which parts to triplicate, at which frequency to operate each part, and on which core a part should be executed. More formally, the problem is defined as follows:

(MinEnergy) *Given a series-parallel graph composed of n tasks, a computing platform composed of c blocks, each equipped with p homogeneous processor cores that can be operated with a speed within set S, an intra-block (resp. inter-block) communication bandwidth β_1 (resp. β_2, with $\beta_1 >> \beta_2$), and a target period P_t, the goal is to partition the graph and decide, for each part, whether to triplicate it or not, at which speed to operate it, on which core to operate it, so that the total expected energy consumption is minimized, under the constraint that the actual period $T(\mathcal{G})$ should not exceed the period bound P_t (to ensure required performance), and that each task is either executed at maximum speed or triplicated (to ensure reliable execution).*

4 Problem complexity

As many partitioning problems, the MinEnergy optimization problem unsurprisingly turns out to be NP-complete. We establish its NP-completeness:

Theorem 1. *The decision version of* MinEnergy *problem is strongly NP-complete.*

Proof. First, we verify than given a mapping of the tasks on the processors, it is possible to verify that (i) each partition follow the structure rule, (ii) the constraint on the execution time is satisfied, and (iii) the required energy of the mapping does not exceed the bound.

To prove the problem NP-hard, we perform a reduction from 3-Partition, which is known to be NP-complete in the strong sense [24]. We consider the following instance $\mathcal{I}_1$ of the 3-Partition problem: let $\{a_1, \ldots, a_{3m}\}$ be $3m$ integers, and B the integer such that $\sum_{i=1}^{3m} a_i = mB$. We consider the variant of the problem, also NP-complete, where $\forall i, B/4 < a_i < B/2$. To solve $\mathcal{I}_1$, we need to solve the following question: does there exist a partition of the a_i's in m subsets $S_1, \ldots, S_m$, each containing exactly 3 elements, such that, for each S_k, $\sum_{i \in S_k} a_i = B$? We build the following instance $\mathcal{I}_2$ of MinEnergy: we consider a fork-join graph as depicted in Figure 2, where $w_{fork} = w_{join} = B$, and $w_i = a_i$. The data carried by edges are assumed of negligible size, and thus $\delta_{i,j} = 0$ for all $i, j \in \mathcal{E}$. We consider a platform with $c = 1$ block of $p = m + 2$ processors, with a set of possible speeds reduced to a single one: $s_{\min} = s_{\max} = 1$. The target period is $P_{\mathrm{t}} = B$. Since we consider the decision version of MinEnergy, we set a bound on the energy: $E \leq I_s \times V_s(m + 2)/B + C(m + 2)B$.

Assume first that there exists a solution to $\mathcal{I}_1$, i.e., that there are m subsets S_k of 3 elements with $\sum_{i \in S_k} a_i = B$. In this case, we build the following mapping as a solution for $\mathcal{I}_2$: T_{fork} and T_{join} are each mapped on a dedicated processor, while for each $1 \leq k \leq m$, the 3 tasks T_i with $i \in S_k$ are mapped on a dedicated processor (no triplication is used). On the whole, the mapping uses $m + 2$ processors. We verify that the computation load of each processor is B, which ensures that both the period bound and the energy bound are met. Besides, this mapping is similar to the one depicted in Figure 2 and thus follows the structure rule.

Reciprocally, assume that there exists a solution to problem $\mathcal{I}_2$, that is, a mapping of tasks that respects all bounds as well as the structure rule. We notice that the total computation load of $(m+2)B$ has to be perfectly balanced on the $m+2$ available processors to reach the period bound B, and that no triplication is possible. Hence, T_{fork} and T_{join} (each of computational weight B) must be mapped on dedicated processors, while m processors are available to compute the T_i's. Since $w_i > B/4$, each processor can accommodate at most 3 tasks. For each of these m processors $P_1, \ldots, P_m$, let S_k be the set of the indices of the 3 tasks mapped on P_k. Thanks to the period bound, we know that $\sum_{i \in S_k} a_i \leq B$ and as $\sum_{i=1}^{3m} a_i = mB$, we have $\sum_{i \in S_k} a_i = B$. Hence, the S_k's form a solution of $\mathcal{I}_1$. □

Since the problem is NP-complete, we first address the easier problem of linear chain applications in Section 5, before designing heuristics for the general case in Section 6.

5 Dynamic programming on a linear chain

If the application is a linear chain, we propose a dynamic programming algorithm to solve MinEnergy. According to the structure rule, the linear chain needs to be partitioned into sub-chains, each of them being assigned to one or three distinct cores, depending whether the sub-chain is triplicated or not. We further consider a *contiguous* allocation, where all cores from a same block are assigned to connected sub-chains (forming together a larger chain).

We consider that we have $c^* \leq c$ available blocks, where the $c^* - 1$ first blocks have p cores available, and the last block has $p^* \leq p$ cores available. We express recursively the minimum energy cost of scheduling tasks T_1 to T_i onto the remaining cores. Either all the tasks form a single part, or we create a part with tasks $T_{j+1}, \ldots, T_i$ and recursively partition the first j tasks.

Initially, we call $E(n, c, p)$, which partitions the whole chain with all blocks and all cores available. The recursion then writes:

$$\begin{aligned} E(i, c^*, p^*) = \min \Big\{ & E_m(1, i, c^*, p^*), \\ & E_t(1, i, c^*, p^*), \\ & \min_{1 \leq j < i} \big\{ E(j, c^*, p^* - 1) + E_c(j, \alpha_1, \beta_1, 1) + E_m(j+1, i, c^*, p^*), \\ & \qquad E(j, c^* - 1, p) + E_c(j, \alpha_2, \beta_2, 1) + E_m(j+1, i, c^*, p^*), \\ & \qquad E(j, c^*, p^* - 3) + E_c(j, \alpha_1, \beta_1, 3) + E_t(j+1, i, c^*, p^*), \\ & \qquad E(j, c^* - 1, p) + E_c(j, \alpha_2, \beta_2, 3) + E_t(j+1, i, c^*, p^*) \big\} \Big\}, \end{aligned} \tag{1}$$

where $E_m(i, j, c^*, p^*)$ (resp. $E_t(i, j, c^*, p^*)$) is the energy cost of executing tasks between T_i and T_j included, at the maximum speed (resp. triplicating the tasks) if there are c^* blocks of cores available, the last one having p^* cores available. Also, $E_c(j, \alpha, \beta, m)$ denotes the energy cost of transferring data of size $\delta_{j,j+1}$ if T_j and T_{j+1} are in different parts: α and β are the energy costs of transferring a unit of data and the bandwidth respectively (their values depend on whether tasks are in a same block or not), and m indicates whether task T_{j+1} is triplicated or not (we pay the communication either three times, or only once).

In the recursive formula $E(i, c^*, p^*)$, we consider all possible situations: either the subchain $T_1, \ldots, T_i$ is mapped in a same part, at maximum speed or triplicated (two first lines), or we cut the chain after T_j. In this case, tasks $T_{j+1}, \ldots, T_j$ are in a same part, triplicated or not, and we consider whether there are in the same block as T_j or in a different block, hence resulting in four different cases.

It remains to express E_m, E_t, and E_c. For E_m, we compute the energy cost as described in Section 3.5:

$$E_m(i, j, c^*, p^*) = \begin{cases} +\infty & \text{if } \sum_{i \leq k \leq j} w_k > P_{\mathrm{t}} s_{\max} \\ & \text{or } p^* < 1 \text{ or } c^* < 1, \\ I_s V_s P_{\mathrm{t}} + C s_{max}^2 \sum_{i \leq k \leq j} w_k & \text{otherwise.} \end{cases} \tag{2}$$

Note that the energy cost is infinite if the period bound is not respected, or if there is no available core ($c^* < 1$ or $p^* < 1$). The expression of E_t relies on s_s, the minimum speed among speeds at which the execution time of tasks between T_i and T_j is not larger than P_{t} (see Section 3.4). We add the energy cost of the majority voting within the same block ($2\alpha_1 \delta_{j,j+1}$), see Section 3.5. The period is infinite if there are less than three cores available, or no block left, or if the period bound cannot be matched:

$$E_t(i, j, c^*, p^*) = \begin{cases} +\infty & \text{if } \sum_{i \leq k \leq j} w_k > P_{\mathrm{t}} s_{\max} \\ & \text{or } p^* < 3 \text{ or } c^* < 1, \\ 3(I_s V_s P_{\mathrm{t}} + C s_s^2 \sum_{i \leq k \leq j} w_k) + 2\alpha_1 \delta_{j,j+1} & \text{otherwise.} \end{cases} \tag{3}$$

Finally, for E_c, the energy is infinite if the communication time is larger than the period, otherwise it is computed as indicated in Section 3.5:

$$E_c(j, \alpha, \beta, m) = \begin{cases} +\infty & \text{if } \delta_{j,j+1} > \beta P_{\mathrm{t}}, \\ m\alpha\delta_{j,j+1} & \text{otherwise.} \end{cases} \tag{4}$$

5.1 Case studies to show that it is not optimal

In this section, we provide an example to show that the method proposed above is not optimal, because of the contiguous assignment of blocks. Consider a platform with $c = 2$ blocks, each

with $p = 4$ cores. Each core can run at a speed in set $S = \{1, 2, 4\}$, with the corresponding operating voltage in set $V = \{1, 2, 4\}$. The characteristics of communications on-chip are given by $\alpha_1 = 1,\ \alpha_2 = 2$ (energy cost) and $\beta_1 = 2,\ \beta_2 = 1$ (bandwidth). The static energy cost of a core of a period is $1P_{\mathrm{t}}$ (i.e., $I_s V_s = 1$), constant C is set to 1. The application is a linear chain with four tasks, the task weights of T_1 to T_4 are $\{4, 4, 1, 1\}$ respectively, and the size of all edges are 0.1. The period bound is $P_{\mathrm{t}} = 1$.

The optimal partition and mapping is: break all edges, each task is a part. The first two tasks are running at the maximum speed and are mapped onto two different blocks, each on a core. The third and forth tasks are triplicated, they both run at speed 1 and are mapped onto two different blocks, each task on three cores. T_2 and T_3 are mapped onto the same block. Then, the energy cost is 137.1 (energy cost of running tasks are $64.1, 64.1, 3.9, 3.9$ for T_1 to T_4, and communication energy cost are $0.2, 0.3, 0.6$ between them).

The optimal partition and mapping proposed above is not a contiguous allocation, and hence it will not be considered by the dynamic programming algorithm. Indeed, since triplication of T_4 uses 3 cores, if T_3 is triplicated as well, it has to move to another block, and the core available in the block with T_4 will never be used. Hence, there is no core left for T_1 (indeed, T_2 and T_1 cannot be in a same part without exceeding the period bound). The minimum energy cost by the dynamic algorithm is 148.4, which is larger than 137.1. It is obtained by having T_1 and T_2 in the first block, T_3 and T_4 in the second block, and by triplicating T_4 only.

There are even cases where no contiguous allocation is possible, and hence the dynamic programming algorithm fails at finding a valid mapping. Consider for instance a linear chain application with eight tasks, all have weight 4, edges between T_2 and T_3, T_6 and T_7 have size 1, other edges have size 2. Other configurations are the same as before. Each task should be mapped onto a different core and operated at maximum speed. Tasks between T_3 and T_6 (both included) should be mapped onto the same block, otherwise the communication time between them will exceed the period. Hence, the dynamic programming algorithm cannot find the solution.

5.2 Condition for optimality

For tasks from T_1 to T_n, if indices of blocks at which tasks are assigned to are monotonically increasing or decreasing, we call this mapping monotonic. More formally, it is defined below:

Definition 2 (Monotonic mapping)**.** *In a monotonic mapping, for any tasks T_i and T_j with $1 \leq i < j \leq n$ and blocks d and f such that $i \in \ell_d$ and $j \in \ell_f$, then $d \leq f$.*

Lemma 1. *The previous dynamic program producing $E(n, c, p)$ finds a mapping whose energy cost is minimal among monotonic mappings.*

Proof. We prove that for any i, c^*, p^* with $i \in \mathbb{N}$, $c^* \in \mathbb{N}$, $p^* \in \mathbb{N}$, $E(i, c^*, p^*)$ finds a mapping whose energy cost is minimal among monotonic mappings. Then $E(n, c, p)$ is naturally the optimal solution.

We first prove that for an application composed of a single task T_1, solution given by the formula is optimal. $E(1, c^*, p^*) = min(E_m(1, 1, c^*, p^*), E_t(1, 1, c^*, p^*))$. The solution returned is the minimum between the energy cost of running T_1 at the maximum speed on processor p^* of block c^* and that of triplicating T_1 at the speed s_s on processors p^*, $p^* - 1$, $p^* - 2$ of block c^*. These two situations include all possibilities: running T_1 at the maximum speed on a core or triplicating T_1 at speed s_s on three cores. Since cores and blocks are homogeneous, taking any of them costs the same energy. So it takes the minimum of all possibilities, which is apparently the optimal solution.

Then we assume that for applications that has at most k tasks, $k \leq i - 1$, $E(k, c', p')$ returns an optimal monotonic solution, $c' \leq c^*$ and $p' \leq p^*$; then we need to prove that $E(i, c^*, p^*)$ is optimal among monotonic mappings for applications that have i tasks with c^* block and p^* cores on each block.

We consider an optimal monotonic mapping M_{opt}, in which we assume the last edge broken is $L_{j,j+1}$, $(1 \leq j \leq i - 1)$. That is to say, tasks from T_{j+1} to T_i are to be mapped onto the same

processor. Assume task T_j is assigned to block c_t, $c_t \leq c^*$. Since the mapping is monotonic, tasks between T_{j+1} to T_i can use only block c_t or c_t+r, $r = 1, 2, 3, \ldots$. Then we consider all possibilities of mapping.

The energy cost of tasks between T_1 and T_j is denoted by E_{left}.

- if T_{j+1} to T_i are running at the maximum speed on a processor of block c_t, then the energy cost of this part is $E_m(j+1, i, c_t, p'+1)$, the communication energy cost between T_j and T_{j+1} is $E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 1)$. In total, the energy cost of the application is $E_m(j+1, i, c_t, p'+1) + E_{left} + E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 1)$;
- if T_{j+1} to T_i are running at the maximum speed on a processor of block $c_t + r$, then the energy cost of this part is $E_m(j+1, i, c_t + r, p^*)$, the communication energy cost between T_j and T_{j+1} is $E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 1)$. In total, the energy cost of the application is $E_m(j+1, i, c_t + r, p^*) + E_{left} + E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 1)$;
- if T_{j+1} to T_i are triplicated on processors of block c_t, then the energy cost of this part is $E_t(j+1, i, c_t, p'+3)$, plus the energy cost on communication $E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 3)$, the total energy cost is $E_t(j+1, i, c_t, p'+3) + E_{left} + E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 3)$;
- if they are triplicated on processors of block $c_t + r$, the energy cost of this part is then $E_t(j+1, i, c_t + r, p^*)$, the communication energy cost is $E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 3)$, plus the energy cost of tasks between T_1 and T_j, the total energy cost is $E_t(j+1, i, c_t + r, p^*) + E_{left} + E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 3)$.

The optimal solution is among them and it costs the least energy. As assumed above, the optimal solution of tasks between T_1 and T_j can be represented as $E(j, c', p')$, $E_{left} \geq E(j, c', p')$. The optimal solution can be rewritten as:

$$\begin{aligned} E_{opt} = \min(&E_m(j+1, i, c_t, p'+1) + E(j, c', p') + E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 1), \\ &E_m(j+1, i, c_t + r, p^*) + E(j, c', p') + E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 1), \\ &E_t(j+1, i, c_t, p'+3) + E(j, c', p') + E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 3), \\ &E_t(j+1, i, c_t + r, p^*) + E(j, c', p') + E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 3)). \end{aligned}$$

Note that cores and blocks are homogeneous and $c_t \leq c' \leq c^*$, $p' \leq p^*$, hence if we keep the relative place of blocks and cores where tasks are allocated. Then it can be rewritten as:

$$\begin{aligned} E_{opt} = \min(&E_m(j+1, i, c^*, p^*) + E(j, c^*, p^* - 1) + E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 1), \\ &E_m(j+1, i, c^*, p^*) + E(j, c^* - 1, p^*) + E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 1), \\ &E_t(j+1, i, c^*, p^*) + E(j, c^*, p^* - 3) + E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 3), \\ &E_t(j+1, i, c^*, p^*) + E(j, c^* - 1, p^*) + E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 3)). \end{aligned}$$

Formula $E(i, c^*, p^*)$ considers all situations above, and it further includes running all tasks on one core with the maximum speed or triplicating them on three cores of the same block, and it returns the minimum of all them, so it returns a result that is not larger than the optimal solution, which shows that it is optimal. □

6 Heuristics for series-parallel graphs

For general series-parallel graphs, we first propose a naive baseline heuristic in Section 6.1, which will be used to evaluate the performance of the proposed sophisticated heuristics. Other heuristics use a two-step approach to map the SPG onto the platform. The first step is to partition the graph into many parts, and the second step is to map these parts onto computing resources. We propose two heuristics that focus on partitioning the graph into many parts, and select the most energy efficient way of execution, while the baseline heuristic executes all tasks at maximum speed (Sections 6.2 and 6.3). The mapping heuristic is described in Section 6.4.

6.1 Baseline heuristic – MaxS

We first outline a baseline heuristic, MaxS, that will serve as a comparison point. It consists in having each task executed at the maximum speed $s_{\max}$, and then mapping greedily tasks to cores. A set L stores a depth-first traversal of $\mathcal{G}$. At each step, we pop up the first node from L and map it onto current core v until total work load on v, $\sum_{i \in C_v} w_i$, is larger than $P_{\mathrm{t}} s_{\max}$. To respect the *structure rule*, if the node is a fork, we map the whole fork-join onto the current core, otherwise if the workload is already too large, we map the fork onto current core, and its successors onto other cores. We first use all cores of the current block before using cores of the next block.

Algorithm 3 describes this heuristic. We start from the last core p on the last block c, and move to the next core on the same block if it has any, otherwise we move to the core p on next block $c-1$.

Algorithm 1 $NextCore(c^*, p^*)$

if $p^* > 1$ **then**
  $p^* \leftarrow p^* - 1$;
else if $c^* > 1$ **then**
  $c^* \leftarrow c^* - 1$, $p* \leftarrow p$;
else
  return $< 0, 0, \emptyset >$;
end if
return $< c*, p*, \emptyset >$;

Algorithm 2 $MapNodesOn(T_i, T_j, c, v)$

for all nodes T_k from T_i to T_j **do**
  set $m_k \leftarrow 1$;
  put T_k into C_v;
  map T_k onto block c and core v;
end for

6.2 Partitioning heuristic – GroupCell

Heuristic GroupCell partitions the graph in a bottom-up way. It first breaks all edges, except (i) edges that have a large communication cost that cannot be done within the period, i.e., $\delta_{i,j} \geq \beta_1 P_{\mathrm{t}}$; and (ii) all edges in a parallel composition when one of the *fork*'s output edges or *join*'s input edges is too large. Indeed, according to the *structure rule*, edges inside this parallel composition should not be broken. For each resulting part, the most energy efficient choice between running at maximum speed or triplicating is selected. Parts stored in vector V_{maxs} are those that are supposed to run at the maximum speed, while others that are supposed to be triplicated are in V_{trip}. For two neighbor parts, if they are both in V_{maxs}, merging them will save the communication. We hence merge parts in V_{maxs} if they are neighbors and if the merged part fits within the period bound. In this process, we respect the *structure rule*, i.e., the resulted part should be either an SPG or a combination of parallel branches, see Section 3.3 for details. If the number of processors requested for the whole graph then exceeds the capacity, we merge parts in V_{trip}, starting with the one with largest input edge weight. This heuristic is described in Algorithm 4.

6.3 Partitioning heuristic – BreakFJ-DP

This second partitioning heuristic builds upon the dynamic programming algorithm that was designed for linear chains. It partitions the graph in a top-down way. First, BreakFJ-DP breaks all input edges of *join* nodes and output edges of *fork* nodes so that resulting parts are either linear

Algorithm 3 MaxS($\mathcal{G}, c, p, P_{\mathrm{t}}$)

$L \leftarrow$ a depth-first traversal of $\mathcal{G}$;
$b \leftarrow c$; $v \leftarrow p$; $C_v \leftarrow \emptyset$; $T_{mapped} \leftarrow L[1]$;
while L is not empty **do**
  $T_i \leftarrow$ pop up the first element of L;
  if $C_v \neq \emptyset$ **then**
    if T_i is not a successor of T_{mapped} or T_{mapped} is a fork **then**
      $< b, v, C_v > \leftarrow NextCore(b, v, C_v)$;
      if $b < 1$ then return fail;
    end if
  end if
  if T_i is a fork node **then**
    $w \leftarrow$ sum of weight of all nodes of fork-join of T_i;
    if $w + \sum_{j \in C_v} w_j > P_{\mathrm{t}} s_{\max}$ **then**
      if $w_i + \sum_{j \in C_v} w_j > P_{\mathrm{t}} s_{\max}$ **then**
        $< b, v, C_v > \leftarrow NextCore(b, v)$;
        if $b < 1$ then return fail;
      end if
      $MapNodesOn(T_i, T_i, b, v)$;
      $T_{mapped} \leftarrow T_i$;
    else
      $T_j \leftarrow$ join of fork-join of T_i;
      $MapNodesOn(T_i, T_j, b, v)$;
      $T_{mapped} \leftarrow T_j$;
      remove nodes of fork-join of T_i from L;
    end if
  else
    if T_i is a join node **then**
      $< b, v, C_v > \leftarrow NextCore(b, v)$;
    end if
    if $w_i + \sum_{j \in C_v} w_j > P_{\mathrm{t}} s_{max}$ **then**
      $< b, v, C_v > \leftarrow NextCore(b, v)$;
    end if
    if $b < 1$ then return fail;
    $MapNodesOn(T_i, T_i, b, v)$;
    $T_{mapped} \leftarrow T_i$;
  end if
end while

Algorithm 4 GROUPCELL($\mathcal{G}, c, p, P_{\mathrm{t}}$)

parts $\leftarrow$ break all edges except the one whose $\delta_{i,j} > \beta_1 P_{\mathrm{t}}$;
$V_{maxs} \leftarrow$ parts in *parts* for which running at the maximum speed costs less energy than triplication;
$V_{trip} \leftarrow$ *parts*$\backslash V_{maxs}$;
sort V_{maxs} by an non-increasing order of input edge size;
for $i = 1$ to $i = |V_{maxs}|$ **do**
  if part $V_{maxs}[i]$'s predecessor is also in V_{maxs} **then**
    if sum of weight of $V_{maxs}[i]$ and its predecessor $\leq P_{\mathrm{t}} s_{\max}$ **then**
      restore the broken edge between $V_{maxs}[i]$ and its predecessor;
      replace $V_{maxs}[i]$ by the combination of $V_{maxs}[i]$ and its predecessor;
    end if
  end if
end for
sort V_{trip} by an non-increasing order of input edge size;
while $|V_{maxs}| + 3|V_{trip}| > cp$ **do**
  part $\leftarrow$ pop up the first element of V_{trip};
  merge *part* into its predecessor;
end while
for all *part* in V_{trip} **do**
  move it into V_{maxs} if running at the maximum speed costs less energy;
end for
for all tasks T_i in V_{maxs} **do**
  set $m_i = 1$;
end for
for all tasks T_i in V_{trip} **do**
  set $m_i = 3$;
end for

chains or single nodes. Dynamic programming algorithm from Section 5 is then called on each of them with the same number of cores and blocks given as BreakFJ-DP.

Note that on a linear chain application, BreakFJ-DP is similar than calling the dynamic programming algorithm on the whole chain, except that mapping the parts to the cores is not done in the dynamic program but in a second step, using the mapping heuristic.

This heuristic is detailed in Algorithm 5.

Algorithm 5 BreakFJ-DP($\mathcal{G}, c, p, P_{\mathrm{t}}$)

set $L \leftarrow$ all *fork* and *join* nodes of $\mathcal{G}$;
Parts $\leftarrow$ break output edges of *fork* and input edges of *join* in L;
$C \leftarrow \emptyset$; /*edges broken*/
repeat
  part $\leftarrow$ pop up the first element of *Parts*;
  $< i, j > \leftarrow$ source node and sink node of *part*;
  $< E, C_{cur} > \leftarrow DP(i, j, c, p)$;
  if $E == +\infty$ **then**
    return failure;
  end if
  $C \leftarrow C \cup C_{cur}$;
until *Parts* is empty

6.4 Mapping heuristic

Once a partition has been returned by GroupCell or BreakFJ-DP, one still needs to map the parts onto the cores. The mapping heuristic first maps parts that need to communicate a large amount of data onto a same block, whenever possible. In a second step, the remaining parts are mapped to the cores following the topology of the graph: a depth-first traversal of the parts is created, and parts are mapped in this order to the available cores. If available cores on the current block are not enough for mapping the current part, then starting using cores from a new block. Some parts may be merged into its predecessor or its parallel part when there are no available cores.

MapRank is the mapping heuristic that considers mapping first parts that are connected by edges of size $\delta_{i,j} > \beta_2 P_{\mathrm{t}}$. A part may have more than one large edge, so parts connected by these edges should all be mapped onto the same block. They are represented as vectors of set L in the first for loop of MapRank. Parts in the same vector should be mapped onto a same block. If number of processors needed by a group exceeds the capacity p, we select the part with the smallest computation weight and execute it at the maximum speed on one processor. This process is repeated until the requirement fits the capacity. According to their demand, parts are sequentially assigned to processors $|Sets[b_{cur}]|$, $|Sets[b_{cur}]|+1$ and so on. MapTopology is called afterwards to map the remaining parts in a sequential order of them. The MapRank algorithm is detailed in Algorithm 6.

In MapTopology (see Algorithm 7), a part is at first mapped onto the same block as its first predecessor, if it is possible. Otherwise, it will be mapped onto the block with the closest index that has enough cores. If the input edge is too large to communicate between two blocks, and current block does not have enough cores, MapTopology first tries to move some parts already mapped onto the current block to the next block, and then continues the mapping from the next block. If it does not work, MapTopology then merges linear chains already mapped from the smallest size until there is enough space.

Algorithm 6 MapRank($\mathcal{G}, C$)

$b_{cur} \leftarrow c$; $L \leftarrow \emptyset$;
construct quotient graph Q by breaking edges in C;
initialize $Sets$ with c empty vectors;
$C' \leftarrow$ edges of Q whose $\delta_{i,j} > \beta_2 P_{\mathrm{t}}$;
for $i = 1$ to $i = |C'|$ **do**
  $< T_p, T_s > \leftarrow$ nodes connected by edge $C'[i]$;
  if T_p is in L but not T_s **then**
    push T_s into the same vector as T_p;
  end if
  if T_s is in L but not T_p **then**
    push T_p into the same vector as T_s;
  end if
  if neither T_p nor T_s is in L **then**
    initialize a vector with them, push it into L;
  end if
end for
while $L \neq \emptyset$ **do**
  $vec \leftarrow$ pop up the first vector of L;
  $nbr \leftarrow \sum_{i \in vec} m_i$; /*cores requested*/
  if $nbr > p$ **then**
    sort vec by an non-decreasing order of weight;
    $idx \leftarrow 1$;
    while $nbr > p$ **do**
      set the node $vec[idx]$ to run on only one core;
      $idx \leftarrow idx + 1$; recaculate nbr;
      if $idx > |vec|$ then return failure;
    end while
  end if
  if p - $|Sets[b_{cur}]| < nbr$ then $b_{cur} \leftarrow b_{cur} - 1$;
  for $i = 1$ to $i = |vec|$ **do**
    if node $vec[i]$ needs 3 processors **then**
      push it into vector $Sets[b_{cur}]$ three times;
    else
      push it into vector $Sets[b_{cur}]$;
    end if
  end for
end while
MapTopology($Q, Sets$);

Algorithm 7 MAPTOPOLOGY($Q, Sets$)

$L \leftarrow$ a depth-first traversal of Q;
repeat
  $T_i \leftarrow$ pop up L; $b \leftarrow c$;
  if T_i has not been mapped yet **then**
    if T_i has a predecessor **then**
      $b_{cur} \leftarrow$ which block the first predecessor of T_i mapped onto;
    end if
    if $p - |Sets[b]| < m_i$ and the size of first input edge is larger than $\beta_2 P_{\mathrm{t}}$ **then**
      $h \leftarrow$ the index such that all input edges of $Sets[b][h]$ are not larger than $\beta_2 P_{\mathrm{t}}$;
      if h exists and $b > 1$ **then**
        move elements between $Sets[b][h]$ and the end of $Sets[b]$ to $Sets[b-1]$;
        $b = b - 1$;
      else
        while $p - |Sets[b]| > m_i$ **do**
          $part_j \leftarrow$ the smallest part of $Sets[b]$ who has only one predecessor that is not a fork;
          merge $part_j$ to its predecessor and remove $part_j$ from $Sets[b]$;
        end while
      end if
    end if
    while $p - Sets[b] \geq m_i$ and $b > 1$ **do**
      $b = b - 1$;
    end while
    if $m_i == 3$ and $p - |Sets[b]| \geq 3$ **then**
      put $part_i$ into $Sets[b]$ three times;
    else
      put $part_i$ into $Sets[b]$;
    end if
  end if
until L is empty

7 Experimental evaluation of the heuristics

In this section, we evaluate all proposed algorithms through extensive simulations on real applications. For reproducibility purposes, the code is available at github.com/gouchangjiang/Stream_HPC.

7.1 Simulation setup

We use a benchmark proposed in [7] for testing the **StreamIt** compiler. It collects many applications from various representative domains, such as video processing, audio processing and signal processing. The stream graphs in this benchmark are mostly parametrized, i.e., graphs with different lengths and shapes can be obtained by varying the parameters. Some applications, such as time-delay equalization, are more computation intensive than others. 44 applications are selected, in which 10 of them are chains, and the list of applications is available in Table 1.

We base our platform parameters on the characteristics of the Intel Skylake-SP Processor [25]: the possible core frequencies are $\{s_{\min} = 1.2, 2.1, 2.4, 2.6, 3.0, 3.7 = s_{\max}\}$, and the idle power of each core is 2.17W. To simulate applications with various communication to computation ratio (CCR), we choose three values of β_1, leading to a CCR (defined as the total time spent on communications over the total time spent on computations) of 10^{-4}, 10^{-3}, or 10^{-2}, while $\beta_2 = \beta_1/16$. α_1 and α_2 are set as 0.2 and 0.8 respectively. C in section 3.5 is set as 1.

For each application, we set the period bound $P_{\mathrm{t}} = a + (b-a)/\kappa$. The value of a is set to the minimum time spent on a task or a data transfer at speed β_1 ($a = \max(w_i/s_{\max}, \min(\delta_{i,j}/\beta_1))$), which corresponds to a very tight period bound. On the contrary, b is set to the time needed to process all tasks on a core at the minimum speed ($b = \sum_{1\leq i\leq n} w_i/s_{\min}$), corresponding to a very loose period bound. We set κ to values from 2 to 10, by increments of 2. Note that it may happen that an application cannot meet the period bound, for instance if an edge between two tasks and the sum of computation cost of these tasks both cannot fit within P_{t}: in that case, all heuristics will fail to produce an appropriate mapping.

Since some heuristics mail fail to produce an acceptable mapping, for each plot described below, we select a subset of applications on which all considered heuristics succeed to produce a mapping, and we plot the average result of the heuristics on this common subset.

7.2 Simulation results

7.2.1 Minimum number of cores request

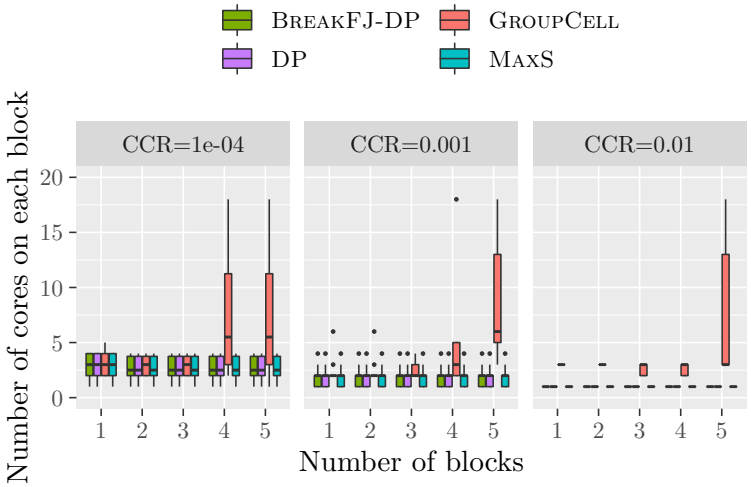
After removing the cases where heuristic does not find a valid solution under a given number of blocks and cores, Fig. 4 shows the minimum number of cores on a block requested by each heuristic, with various number of blocks provided, where κ is set to 4, a median value. On linear chains, as shown in Fig. 4a, dynamic programming (denoted as DP), MaxS as well as BreakFJ-DP have the same performance, they require the least number of cores. GroupCell requires averagely 2.4 times more cores than DP. On general SPGs, as shown in Fig. 4b, BreakFJ-DP uses far more cores than GroupCell and MaxS. For instance, with CCR=10^{-3}, BreakFJ-DP uses 5 times more cores than GroupCell in average.

7.2.2 Energy cost

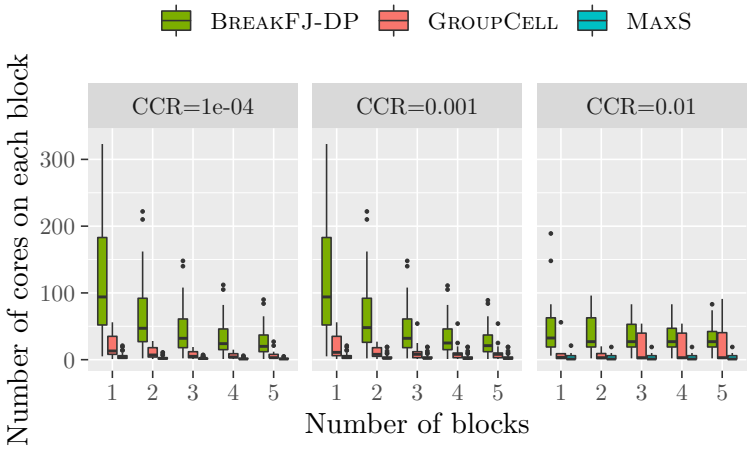
Figures 5 and 6 depict the energy cost as a function of κ, where a larger κ represents a tighter period, with different number of blocks and cores given. For linear chains, see Fig. 5, when 2 blocks and 2 cores on each block are given, 2, 3 and 1 applications are considered for CCR=10^{-4}, 10^{-3} and 10^{-2} respectively, since at least one heuristic fails to get a valid mapping on other applications. The number of applications included are 10, 8, 9 for CCR=10^{-4}, 10^{-3} and 10^{-2} respectively, when 4 cores are given on each block. BreakFJ-DP and DP reduce by 33% on average compared to MaxS, and around 60% when communications are expensive. It leads to the same conclusion when 2 blocks, each with 8 cores, are provided. BreakFJ-DP and DP reduce the energy by 44% on average compared to MaxS. Note that when CCR=10^{-4}, the results only

Name	Nbr Nodes	Nbr Edges	Max Degree	Max Node Weight	Min Node Weight	Max Edge Size	Min Edge Size	Chain
B10cholesky	80	95	2	9088	8	136	1	
B11CP	22	38	17	272	96	257	1	
B13DCT	66	80	2	240	12	16	2	
B13GPP	214	313	32	36771	3	1344	1	
B14DCT2D	86	108	4	128	12	16	2	
B15DCT2D	24	38	8	4864	384	64	8	
B16DES	197	229	2	1024	192	96	32	
B19FFT	13	13	1	2464	632	128	128	T
B20FFT	283	469	32	4035	4	128	4	
B22FFT	50	62	2	448	192	2	1	
B25FMRadio	43	54	6	1434	8	60	1	
B27fliterbank	85	100	8	11312	6	64	1	
B280211a	132	177	12	44928	6	1728	1	
B36IDCT	97	128	8	128	12	8	1	
B37IDCT2D	110	140	4	128	12	8	1	
B38IDCT2D	24	38	8	4864	384	8	1	
B39IDCT2D	4	4	1	1576	1104	1	1	T
B3audiobeam	20	34	15	140	22	15	1	
B40IDCT2D	22	36	8	138	138	8	1	
B41insertionsort	6	6	1	745	96	1	1	T
B44lattice	46	55	2	29	6	2	1	
B45matrixmult	43	63	9	27648	72	468	12	
B46matrixmult	54	93	12	3528	72	2592	9	
B47mergesort	31	38	2	208	96	16	2	
B49mp3	180	295	32	414144	96	9216	16	
B4autocar	12	19	8	579	48	32	1	
B50MPD	165	211	32	3274750	259	140	0	
B53OFDM	16	19	4	181500	24	3300	0	
B54oversampler	10	10	1	11360	11	16	1	T
B56Radar	53	67	12	5076	332	12	0	
B57radixsort	13	13	1	208	96	1	1	T
B58ratecovert	5	5	1	19836	32	2	0	T
B5bitonicsort	6	6	1	265	96	16	16	T
B60raytracer2	5	5	1	473	8	1	1	T
B61SAR	44	45	2	6541490000	3	167316	1	
B63serpent	234	267	2	3336	68	256	4	
B64TDE	29	29	1	36960	12840	1920	1080	T
B65targetdetect	12	15	4	3306	8	4	1	
B66vectadd	6	7	2	10	6	2	1	
B67vocoder	116	151	15	9105	6	60	1	
B6bitonicsort	170	240	8	126	14	16	2	
B7bitonicsort	152	201	8	128	6	16	1	
B8bubblesort	18	18	1	23	6	1	1	T
B9channelvocoder	57	73	16	65055	251	1	0	

Table 1: Set of streaming applications.



(a) Linear chain



(b) General SPG

Figure 4: Number of cores requested by each block with different CCR and number of blocks provided.

include 3 applications out of 10, since GroupCell fails on other applications because of shortage of cores. For CCR=10^{-3} and 10^{-2}, 9 applications are included.

The gains are also very impressive for general SPGs, see Fig. 6. With 2 blocks, each with 128 cores given, both heuristics save more than 50% of energy in all settings, with BreakFJ-DP being better for tighter periods and larger CCRs. Note however that the results for CCR=10^{-2} are computed only on a small subset of applications, 6 out of 34, since the heuristics failed on the other applications: the period bound could not be met because of the high communication cost on some edges. For CCR=10^{-3} and 10^{-4}, 31 and 32 applications are included respectively.

With 2 blocks, each equipped with 64 cores, both heuristics also save more than 50% of energy in all settings, with BreakFJ-DP being better at least 5% in most cases. 27, 20 and 4 applications are included for CCR = 10^{-4}, 10^{-3} and 10^{-2} respectively, since at least one heuristic does not return a valid mapping on other applications.

With 2 blocks, each equipped with 32 cores, BreakFJ-DP and GroupCell still outweigh MaxS by around 44% when CCR=10^{-4}. BreakFJ-DP is still slightly better than GroupCell. 17, 11 and 2 applications are included when CCR=10^{-4}, 10^{-3} and 10^{-2} respectively.

7.2.3 Failure cases

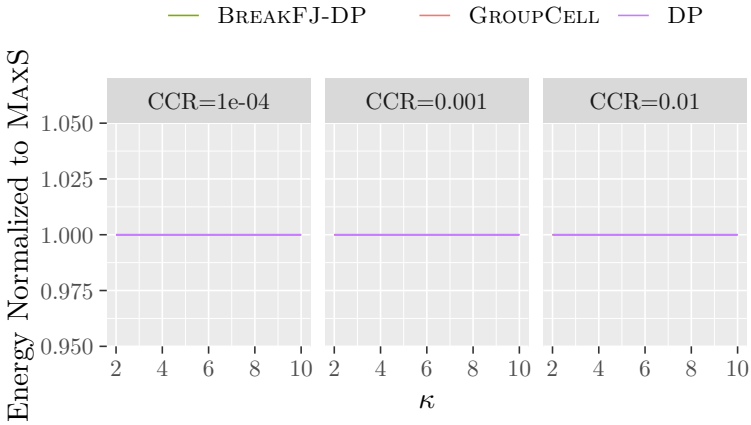
We report percentage of failure cases in Figures 7 and 8. Note that, in total, there are 34 general SPGs and 10 linear chains. On linear chains (Fig. 7), there are several scenarios where GroupCell fails more than other heuristics, in particular with few cores per blocks. However, most heuristics have no more than 10% of failures, i.e., at most one application did fail.

Percentage of failure cases on general SPGs are shown in Fig. 8. BreakFJ-DP is the one that fails more than other heuristics since it requests more cores. When communication is not so expensive (CCR=10^{-4} or 10^{-3}), the percentage of failures cases of BreakFJ-DP is 40% with 32 cores per block, and it decreases to 20% with 64 cores, and then to zero with 128 cores. The same happens for other heuristics: if there are more cores on a block, the heuristics fail on less applications. With a tight communication bandwidth (CCR=10^{-2}) and a tight period bound ($8 \leq \kappa \leq 10$), more than 50% of the applications fail. Note that a *failure* means that the application cannot be executed on the platform while meeting all constraints, and one should then consider increasing the platform bandwidth if the application has high communication requirements, or relaxing the bound on the period.

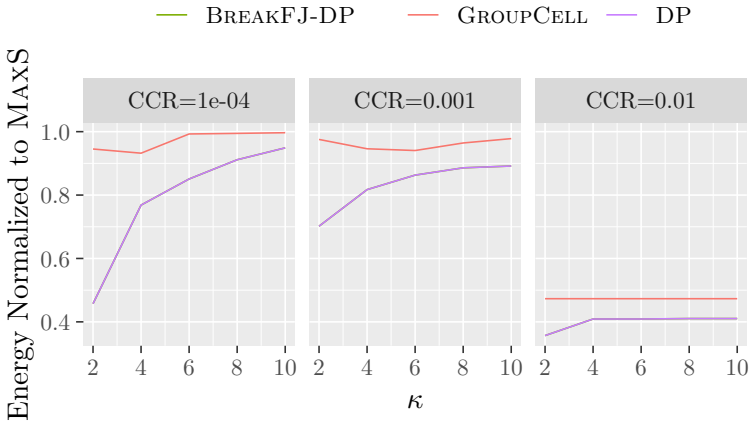
8 Conclusion

We have addressed the problem of mapping streaming SPG applications onto a hierarchical two-level platform, with the goal of minimizing the energy consumption, while ensuring performance (a period bound should not be exceeded) and a reliable execution (each task should either be executed at maximum speed or triplicated). We have formalized the problem and proven its NP-completeness, and provided practical solutions building upon a dynamic programming algorithm, which returns the optimal *contiguous* mapping for a linear chain. Heuristics are proposed for general SPGs, and the BreakFJ-DP heuristic that builds upon the DP algorithm provides significant savings in terms of energy consumption, with more than 61% savings, in particular when the period bound is not too tight. With tighter period bounds, we still achieve 57% savings. However, this heuristic may fail with limited number of cores per blocks. In this case, our GroupCell heuristic is an interesting alternative, with only a slightly greater energy consumption for a reduced number of cores used.

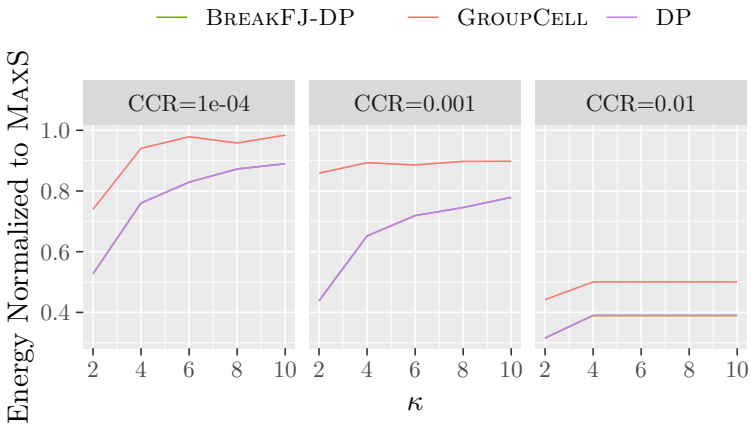
An interesting open question is whether the proposed dynamic program is an approximation algorithm: even though it is not optimal in the general case, it works well in practice and it would be interesting to provide a guarantee on its performance.



(a) BreakFJ-DP, GroupCell and DP give the same results, hence only DP is visible, 2 blocks, each with 2 cores provided.

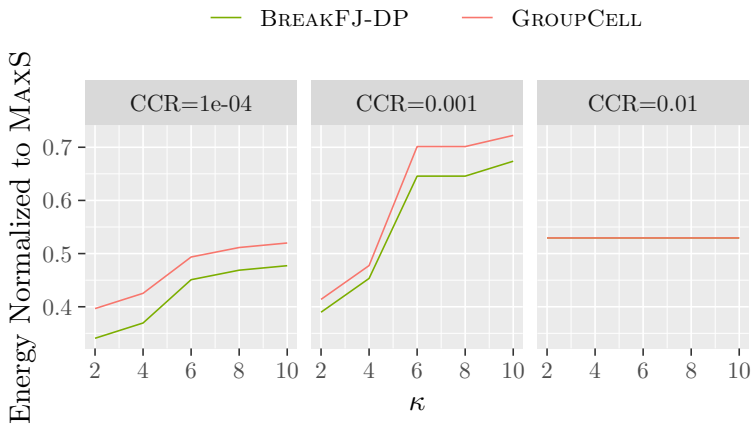


(b) BreakFJ-DP and DP give the same results, hence only DP is visible, 2 blocks, each with 4 cores provided.

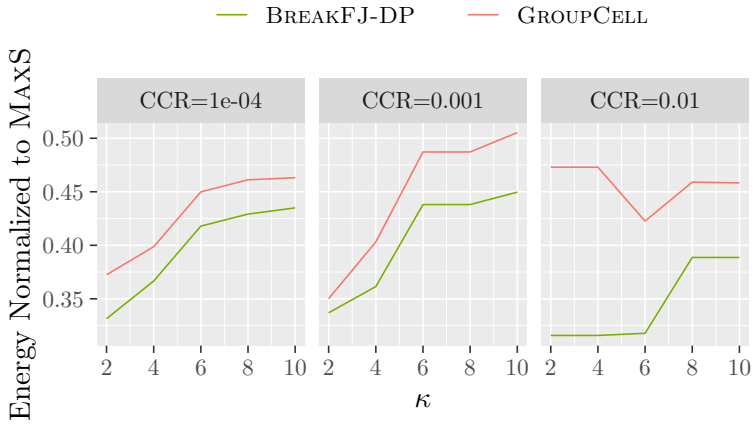


(c) BreakFJ-DP and DP give the same results, hence only DP is visible, 2 blocks, each with 8 cores provided.

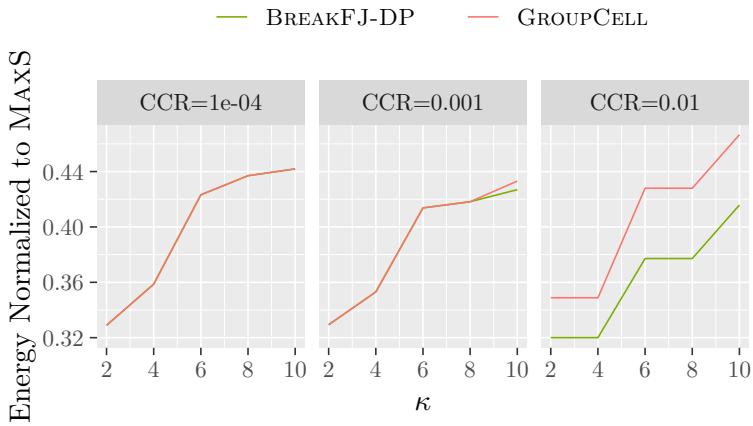
Figure 5: Energy consumption on linear chains relative to MaxS as a function of the period bound tightness κ.



(a) For $CCR = 10^{-2}$, both heuristics give the same results, 4 blocks, each with 32 cores provided.

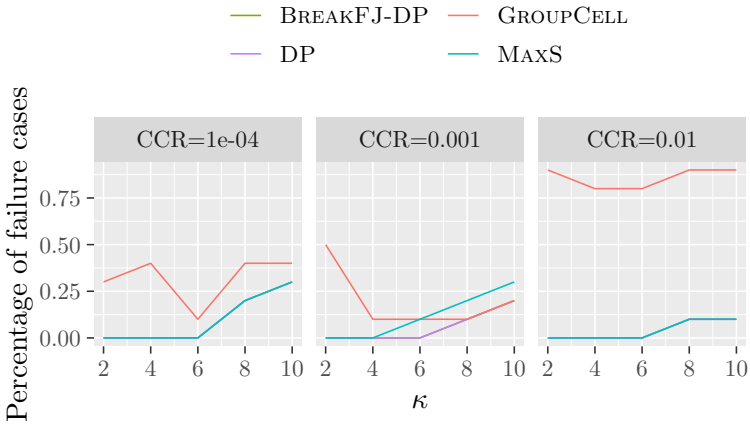


(b) 4 blocks, each with 64 cores provided.

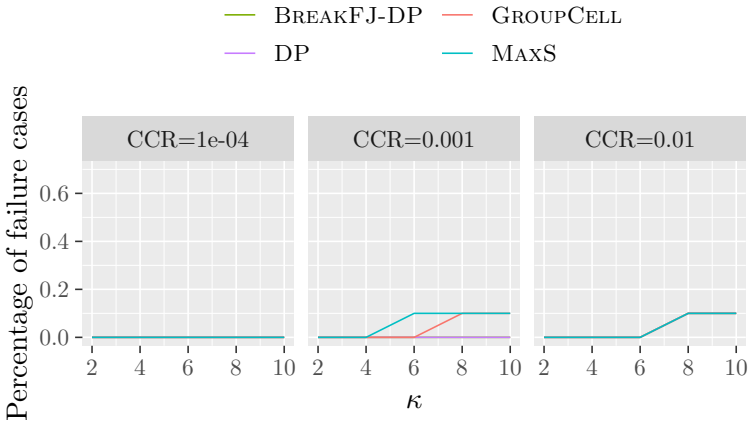


(c) For $CCR = 10^{-4}$, both heuristics give the same results, 4 blocks, each with 128 cores provided.

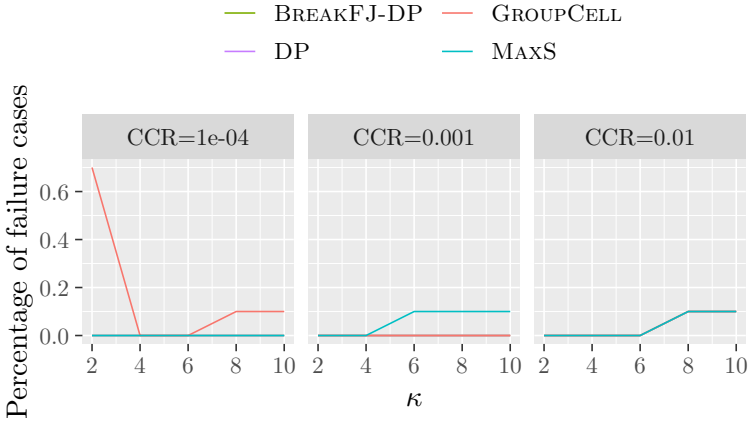
Figure 6: Energy consumption on general SPGs relative to MaxS as a function of the period bound tightness κ.



(a) Linear chains. 2 blocks, each with 2 cores provided. BreakFJ-DP and DP are sometimes covered by MaxS.

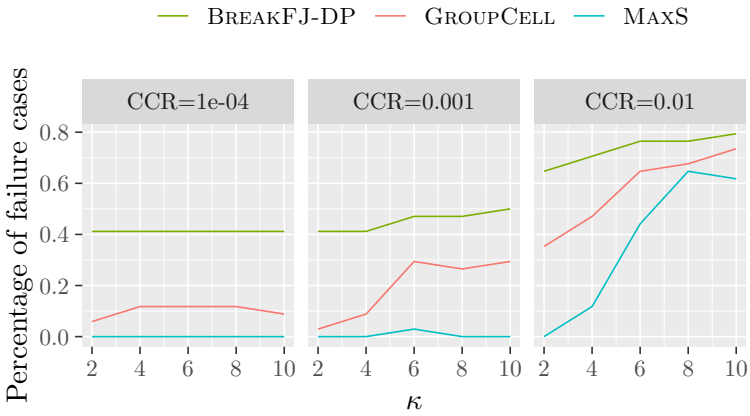


(b) Linear chains. 2 blocks, each with 4 cores provided. BreakFJ-DP is covered by DP when CCR=10^{-3}.

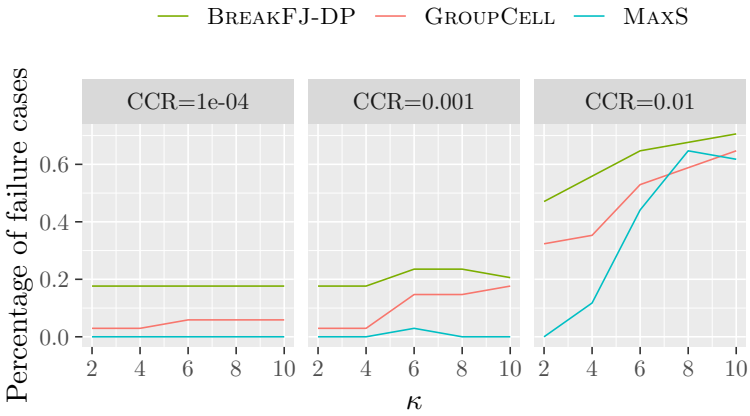


(c) Linear chains. 2 blocks, each with 8 cores provided. BreakFJ-DP and DP are covered by MaxS for CCR=10^{-4}. They are covered by GroupCell for CCR=0.001.

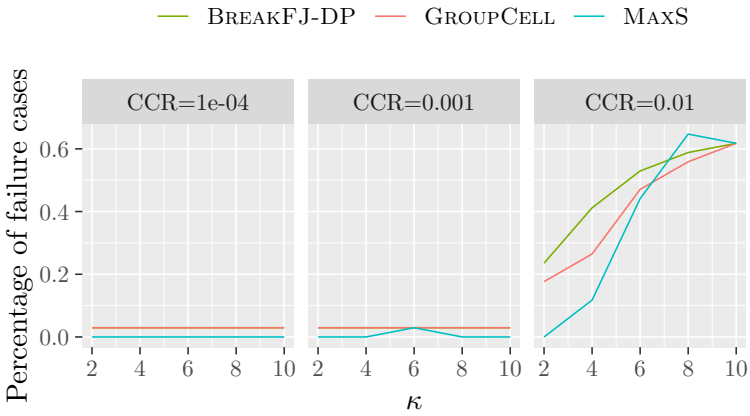
Figure 7: Percentage of failure cases on linear chains as a function of κ.



(a) General SPGs. 4 blocks, each with 32 cores provided.



(b) General SPGs. 4 blocks, each with 64 cores provided.



(c) General SPGs. 4 blocks, each with 128 cores provided. BreakFJ-DP has the same number of failure cases as GroupCell when CCR=10^{-4} and 10^{-3}.

Figure 8: Percentage of failure cases on general SPGs as a function of κ.

References

[1] J. Deslippe, A. Essiari, S. J. Patton, T. Samak, C. E. Tull, A. Hexemer, D. Kumar, D. Parkinson, and P. Stewart, "Workflow management for real-time analysis of lightsource experiments," in *Proc. of WORKS'14*, 2014, pp. 31–40.

[2] CMS Collaboration, "CMS data processing workflows during an extended cosmic ray run," *Journal of Instrumentation*, vol. 5, no. 03, pp. T03 006–T03 006, mar 2010.

[3] A. Luckow, G. Chantzialexiou, and S. Jha, "Pilot-streaming: A stream processing framework for high-performance computing," 2018.

[4] D. Zhu, "Reliability-aware dynamic energy management in dependable embedded real-time systems," *ACM Trans. on Embedded Computing Systems*, vol. 10, pp. 26:1–26:27, 2011.

[5] D. Zhu, R. Melhem, and D. Mosse, "The effects of energy management on reliability in real-time embedded systems," in *Proc. of ICCAD'04*, USA, 2004, pp. 35–40.

[6] R. Maciulaitis and al, "Support for HTCondor high-throughput computing workflows in the REANA reusable analysis platform," CERN, Tech. Rep. CERN-IT-2019-004, Sep 2019.

[7] W. Thies, "Language and compiler support for stream programs," Ph.D. dissertation, MIT, Cambridge, MA, USA, 2009.

[8] Q. Tang, S. Wu, J. Shi, and J. Wei, "Optimization of duplication-based schedules on network-on-chip based multi-processor system-on-chips," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 826–837, March 2017.

[9] M. Flasskamp, G. Sievers, J. Ax, C. Klarhorst, T. Jungeblut, W. Kelly, M. Thies, and M. Porrmann, "Performance estimation of streaming applications for hierarchical mpsocs," in *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '16. New York, NY, USA: ACM, 2016, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/2852339.2852342

[10] A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, and M. J. Garzarán, "Mapping streaming applications on commodity multi-cpu and gpu on-chip processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1099–1115, April 2016.

[11] G. Onnebrink, F. Walbroel, J. Klimt, R. Leupers, G. Ascheid, L. G. Murillo, S. Schürmans, X. Chen, and Y. Harn, "DVFS-enabled power-performance trade-off in MPSoC SW application mapping," in *SAMOS'17*, 2017, pp. 196–202.

[12] M. A. Haque, H. Aydin, and D. Zhu, "On reliability management of energy-aware real-time systems through task replication," *IEEE TPDS*, vol. 28, no. 3, pp. 813–825, March 2017.

[13] J. Zhou, T. Wei, M. Chen, J. Yan, X. S. Hu, and Y. Ma, "Thermal-aware task scheduling for energy minimization in heterogeneous real-time mpsoc systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 8, pp. 1269–1282, 2016.

[14] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik, "Cola: Optimizing stream processing applications via graph partitioning," in *Middleware 2009*, J. M. Bacon and B. F. Cooper, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 308–327.

[15] J. Yu and R. Buyya, "A budget constrained scheduling of workflow applications on utility grids using genetic algorithms," in *2006 Workshop on Workflows in Support of Large-Scale Science*, June 2006, pp. 1–10.

[16] K. Huang, S. Xiu, M. Yu, X. Zhang, R. Yan, X. Yan, and Z. Liu, "Software pipeline–based partitioning method with trade-off between workload balance and communication optimization," *ETRI Journal*, vol. 37, no. 3, pp. 562–572, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.15.0114.0502

[17] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of scientific workflows in the askalon grid environment," *SIGMOD Rec.*, vol. 34, no. 3, pp. 56–62, Sep. 2005. [Online]. Available: https://doi.org/10.1145/1084805.1084816

[18] W. Kelly, M. Flaßkamp, G. Sievers, J. Ax, J. Chen, C. Klarhorst, C. Ragg, T. Jungeblut, and A. Sorensen, "A communication model and partitioning algorithm for streaming applications for an embedded mpsoc," in *2014 International Symposium on System-on-Chip (SoC)*, Oct 2014, pp. 1–6.

[19] V. Cavé, R. Clédat, P. Griffin, A. More, B. Seshasayee, S. Borkar, S. Chatterjee, D. Dunning, and J. Fryman, "Traleika glacier: A hardware-software co-designed approach to exascale computing," *Parallel Computing*, vol. 64, pp. 33 – 49, 2017.

[20] A. Benoit and Y. Robert, "Mapping pipeline skeletons onto heterogeneous platforms," *J. Parallel and Distributed Computing*, vol. 68, no. 6, pp. 790–808, 2008.

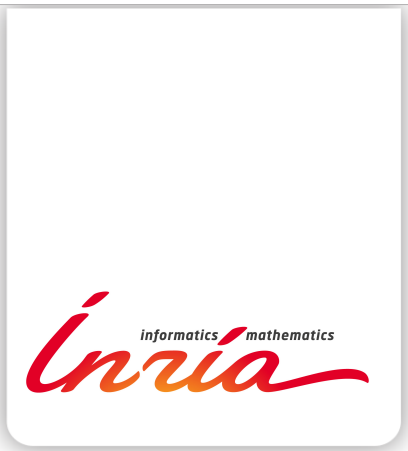
[21] I. Assayad, A. Girault, and H. Kalla, "Tradeoff exploration between reliability, power consumption, and execution time for embedded systems," *International Journal on Software Tools for Technology Transfer*, vol. 15, pp. 229–245, 2013.

[22] G. Aupy and A. Benoit, "Approximation algorithms for energy, reliability, and makespan optimization problems," *Parallel Process. Lett.*, vol. 26, no. 1, pp. 1–23, 2016.

[23] H. Xu, R. Li, C. Pan, and K. Li, "Minimizing energy consumption with reliability goal on heterogeneous embedded systems," *J. of Parallel and Distributed Computing*, vol. 127, pp. 44 – 57, 2019.

[24] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. London (UK): W.H. Freeman and Co, 1979.

[25] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, and D. Hackenberg, "Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance," *CoRR*, vol. abs/1905.12468, 2019.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803