# 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[1,2], Anne Benoit[1], Mingsong Chen[2], Loris Marchal[1], Tongquan Wei[2]
1. LIP Laboratory, ENS Lyon, CNRS, France
2. East China Normal University, China

November 25, 2020

**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.

## 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. Due to space limitation, further related work is discussed in the companion research report [**?**].

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. Section 2 formalizes both application and platform models and defines the MinEnergy optimization problem. Section 3 presents a dynamic programming-based solution for MinEnergy when dealing with linear chain applications, and Section 4 proposes heuristics for general series-parallel graphs. Section 5 evaluates the proposed algorithms. Finally, Section 6 concludes the paper and provides directions for future work.

# 2 Model

## 2.1 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 $P_{\mathrm{t}}$.

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. 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 $P_{\mathrm{t}}$, i.e., a new data set enters the system every $P_{\mathrm{t}}$ 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 2.6.

## 2.2 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 $\beta_1$ and $\beta_2$. Communication among cores in the same block is cheaper than that among different blocks [19], i.e., $\beta_1 >> \beta_2$.

## 2.3 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. 1. 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. 1.

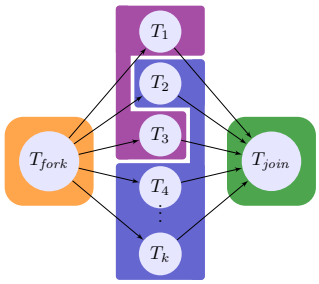



Figure 1: Fork-join graph and a partition following the structure rule.

A parallel composition of more complex subgraphs is depicted in Figure 2 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)).

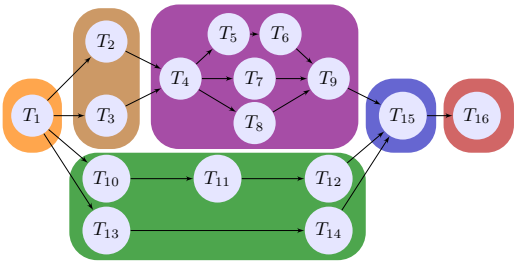



Figure 2: SPG partition following the structure rule.

**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 $\ell_d$, 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.

## 2.4 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 $\lambda_0$ is the average failure rate at speed $s_{\max}$. $\lambda_0$ 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 5), 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 $5.46 \times 10^{-4}$ 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.

## 2.5 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 $P_{\mathrm{t}}$ time units, the static energy is consumed during a time $P_{\mathrm{t}}$ 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:

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

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 5: $s_{\min}$ 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:

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

## 2.6 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, i.e., $T(\mathcal{G}) \leq P_{\text{t}}$.

### 2.7 Optimization problem and complexity

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 $\beta_1$ (resp. $\beta_2$, 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).*

MinEnergy is NP-complete in the strong sense, as we prove in the companion research report [**?**] through a reduction from 3-partition [24]. The reduction builds a fork-join application and ensures that no triplication is used, hence each task is run at maximum speed. The platform has a single block and tasks must be grouped three by three to ensure that the period bound is respected, given a tight number of cores.

Since the problem is NP-complete, we first address the easier problem of linear chain applications in Section 3, before designing heuristics for the general case in Section 4.

## 3 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: $E(i, c^*, p^*) = \min \Big\{$

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

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: $\alpha$ and $\beta$ 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 2.5:

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

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 $P_{\mathrm{t}}$ (see Section 2.4). We add the energy cost of the majority voting within the same block ($2\alpha_1 \delta_{j,j+1}$), see Section 2.5. The period is infinite if there are less than three cores available, or no block left, or if the period bound cannot be matched:

$E_t(i, j, c^*, p^*) =$

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

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 2.5:

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

We prove in the companion research report [**?**] that this dynamic programming algorithm is optimal for contiguous mappings, since we explore all possible valid solutions in the recursive formula for $E(i, c^*, p^*)$. However, it may happen that the optimal solution is not contiguous. For instance, if there are four tasks, two blocks with four cores, and we should triplicate the third and fourth tasks, then the optimal solution maps tasks 1 and 4 on a block, and tasks 2 and 3 on the other block. Details can be found in [**?**].

# 4 Heuristics for series-parallel graphs

For general series-parallel graphs, we first propose a naive baseline heuristic, MaxS, 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 (GroupCell and BreakFJ-DP). Finally, we describe the mapping heuristic. All pseudo-codes and additional heuristic variants are available in the companion research report [**?**].

**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.

**GroupCell partitioning heuristic** – 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 2.3 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.

**BreakFJ-DP partitioning heuristic** – 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 chains or single nodes. Dynamic programming algorithm from Section 3 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.

**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.

# 5 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.

## 5.1 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 are mostly parametrized, i.e., graphs with different lengths and shapes can be obtained by varying the parameters. 44 applications are selected, in which 10 of them are chains, more details can be found in [**?**].

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 $\beta_1$, 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 $\beta_2 = \beta_1/16$. $\alpha_1$ and $\alpha_2$ are set as 0.2 and 0.8 respectively. $C$ is set as 1.

For each application, we set the period bound $P_{\text{t}} = a + (b-a)/\kappa$. The value of $a$ is set to the minimum time spent on a task or a data transfer at speed $\beta_1$ ($a = \max(w_i/s_{\max}, \min(\delta_{i,j}/\beta_1))$), 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 $\kappa$ 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 $P_{\text{t}}$: 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.

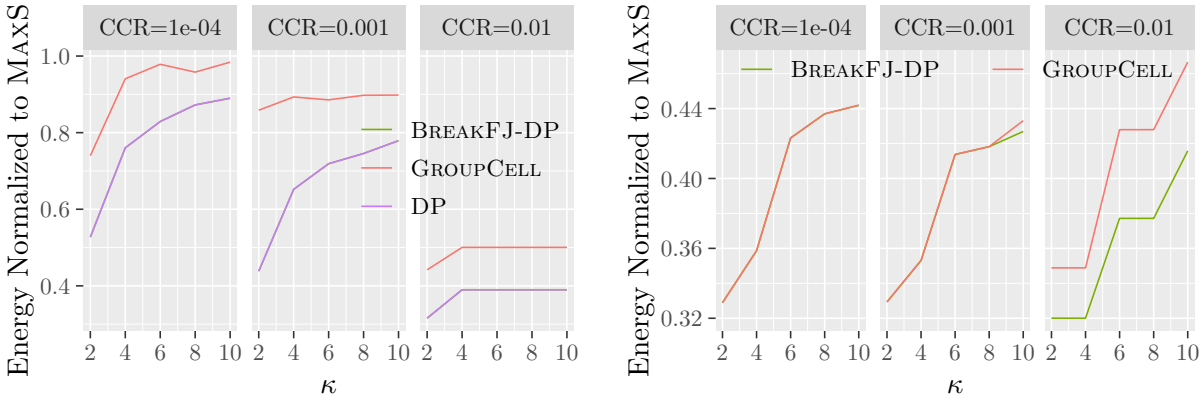
### 5.2 Simulation results

Fig. 3 depicts the energy cost as a function of $\kappa$, where a larger $\kappa$ represents a tighter period. The platform is composed of $c = 4$ (resp. 2) blocks, each equipped with $p = 128$ (resp. 8) cores for general SPGs (resp. linear chains). For linear chains, BreakFJ-DP and DP reduce the energy by 44% on average compared to MaxS, and around 60% when communications are expensive. Note that when CCR=$10^{-4}$, the results only 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, where 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.

Results are quite similar with fewer cores per blocks, see [**?**] for detailed results, but the number of cases where the heuristics cannot find a solution increases, because heuristics may not be able to avoid the costly communication between blocks. Also, the difference between BreakFJ-DP and GroupCell can be observed even for smaller CCRs in these cases, with BreakFJ-DP saving slightly more energy than GroupCell. We also report in [**?**] detailed numbers of cases when heuristics fail to produce an appropriate mapping, together with a study about the minimum number of cores that are required for each heuristic. The execution time of all heuristics can be found in [**?**]. Even though BreakFJ-DP leads to the most interesting energy savings, it requires more cores to run successfully, hence GroupCell may be a better alternative in some cases.

## 6 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,




(a) Linear chains (BreakFJ-DP and DP give the same results, hence only DP is visible).

(b) General SPGS (for $CCR = 10^{-4}$, both heuristics give the same results).

Figure 3: Energy consumption relative to MaxS as a function of the period bound tightness $\kappa$.

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.

## Acknowledgement

This work is supported by National Key Research and Development Program of China 2018YFB2101300, and the Natural Science Foundation of China 61872147.

## 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 et 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] C. Gou and et al., "Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms," INRIA, Research Report RR-9346, 2020.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.