Mapping series-parallel streaming applications on hierarchical platforms with reliability and energy constraints

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

Mapping series-parallel streaming applications on hierarchical platforms with reliability and energy constraints*

Changjiang Gou[b,a], Anne Benoit[a], Mingsong Chen[c], Loris Marchal[a], Tongquan Wei[c]

[a] *LIP Laboratory, ENS Lyon, CNRS, France*
[b] *Zhejiang Lab, Hangzhou, China*
[c] *East China Normal University, Shanghai, China*

Abstract

Streaming applications come from various application fields such as physics, where data is continuously generated and must be processed on the fly. Typical streaming applications have a series-parallel dependence graph, and they are processed on a hierarchical failure-prone platform, as for instance in miniaturized satellites. The goal is to minimize the energy consumed when processing each data set, while ensuring real-time constraints in terms of processing time. 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, so that the time to execute a data set without failure is bounded. We propose a structure rule to partition the series-parallel applications and map the application onto the platform, and we prove that the optimization problem is NP-complete. We design a dynamic-programming algorithm for the special case of linear chains, which is optimal for a special class of schedules. Furthermore, this algorithm provides an interesting heuristic and a building block for designing heuristics for the general case. The heuristics are compared to a baseline solution, where each task is executed at maximum speed. Simulations on realistic settings demonstrate the good performance of the proposed heuristics; in particular, significant energy savings can be obtained.

1. Introduction

Streaming data is continuously generated from applications in high energy physics [1], astronomy [2] and other scientific or industrial domains [3]. 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 [4]. Such streaming applications, also referred to as scientific workflows, are often represented as Directed Acyclic Graphs (DAGs). Indeed, the

*A shorter version of this paper appeared in the proceedings of SBAC-PAD'2020 under the title *Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms.*

Email addresses: gouchangjiang@gmail.com (Changjiang Gou), Anne.Benoit@ens-lyon.fr (Anne Benoit), mschen@sei.ecnu.edu.cn (Mingsong Chen), Loris.Marchal@ens-lyon.fr (Loris Marchal), tqwei@cs.ecnu.edu.cn (Tongquan Wei)

graph models the computation needs of tasks and dependencies among tasks [5]. 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 [6] are series-parallel graphs. Hence, we focus in this work on series-parallel applications.

In this context, processing the data in real-time, so that a feedback with key information for decision making can be obtained, usually requires a huge computing power. Hence, the use of large-scale hierarchical platforms can help parallelize the processing of this streaming data in real time. The platform on which we aim at executing such applications is a two-level platform, where cores are organized into computing blocks. The challenge consists in mapping the application onto this hierarchical platform, so that each data set is processed through each task without exceeding a given bound.

In order to succeed in finding such a mapping, a natural strategy consists in operating the platform at the highest possible speed. This has the advantage of ensuring that the platform is reliable; even though a few errors may strike the platform, a small percentage of failures is usually acceptable (we loose a very small number of data sets). However, the drawback is that it leads to a high energy consumption, and this might be critical for the target platform, as for instance satellites with a limited amount of energy.

Therefore, the objective is to minimize the energy consumption while ensuring a valid mapping (respecting the bound on processing time). The energy consumption can indeed be reduced by using Dynamic Voltage and Frequency Scaling (DVFS): by operating the cores at a lower frequency and voltage, we can reduce the energy consumption required to complete a task. However, the use of lower voltages may result in an increased arrival rate of transient faults [7, 8]. 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 the effects of soft errors, the frequent utilization of such fault-tolerance mechanisms is time-consuming and not appropriate to applications with real-time processing time constraints. Indeed, the unpredictable occurrences of soft errors may result in severe temporal violations. Similar soft errors occur on Xilinx UltraScale and UltraScale+ devices using ARM Cortex-A53 processors, in particular in the field of miniaturized satellites [9, 10]. Indeed, the use of radiation-hardened components is costly and difficult to implement, and it is necessary to protect from such errors even with Error-Correcting Code (ECC) memory components [11]. In fact, it has been shown that ECC only protects the integrity of the data stored in the SDRAMs and caches. Failures due to space radiation may cause temporary or permanent failures of memory integrated circuits, regardless of the memory technology used.

In order to deal with such errors, we rather use the standard technique of task triplication: tasks that are not executed at maximum speed, and hence subject to errors, are executed three times. Therefore, errors can be detected and corrected with a majority voting. Triplication has a major advantage over task duplication, which considers only two replicas: in task triplication, we do not need to recompute a task in case of different outcomes of two replicas. This allows each task to have a constant processing time, which is desirable for the steady-state scheduling of streaming applications. Recall that we take the standard assumption that the reliability is high enough when tasks are executed at maximum speed. The target optimization problem is therefore the following: the goal is to map a series-parallel streaming application on a hierarchical computing platform, with the aim at minimizing the energy consumption, while respecting constraints in terms of performance (the execution time should not exceed a prescribed bound), and of reliability (each task should be either executed at maximum speed, or triplicated).

We summarize our major contributions below:

1. We propose a formal model for the multi-objective optimization problem. In particular, we introduce a *structure rule* for simple and efficient mappings of series-parallel applications, we detail the reliability and processing time constraints, and we explain how the energy consumption is computed.

2. We prove that the corresponding optimization problem, MinEnergy, is strongly NP-complete.

3. We design a dynamic programming approach for applications consisting in a simple linear chain of streaming tasks, and prove the optimality of this approach for a particular class of schedules.

4. Building upon the dynamic programming algorithm, we design several mapping and scheduling heuristics for the general case.

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

The rest of this paper is organized as follows. Related work is discussed in Section 2. Then, Section 3 formalizes both application and platform models and defines the MinEnergy optimization problem. Section 4 establishes the strong NP-completeness of MinEnergy. 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, the efficient parallel processing of streaming applications on hierarchical platforms has attracted growing research interest. Considering the computation and communication costs of Directed Acyclic Graphs (DAGs), Tang et al. [12] presented two heuristic strategies based on integer linear programming to reduce communication overhead and scheduling length. Flasskamp et al. [13] designed a performance estimator embedded in the compiler to partition and map streaming applications. For a Multi-Processor System-on-Chip (MPSoC) system consisting of a multi-core CPU and on-chip GPUs, Vilches et al. [14] 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. [15] 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. [16] 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. [17] 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. [18] described an iterative algorithm that partitions streaming application graphs to balance the load and minimize the communication costs. Yu et al. [19] proposed a genetic algorithm to map a workflow onto utility grids to minimize execution time under a given budget constraint. In [20], Huang et al. partitioned a task graph with cyclic dependencies into parts, and mapped each part to a processor to achieve a balance between communication and workload. Wieczorek et al. [21] evaluated the performance of three scheduling strategies for mapping scientific workflows onto the grid, and the experimental results demonstrate that the HEFT algorithm is near-optimal for balanced and unbalanced applications. For streaming applications in hierarchical MPSoC systems, Kelly et al. [22] proposed a simulated annealing-based compiler to achieve better performance compared to the most advanced partitioning algorithms. However, none of these studies consider both dependent tasks and reliability requirements.

The optimization problem targeted in this paper is found for instance in the field of aboard satellites, especially miniaturized satellites. Instead of using radiation-hardened components that can be costly and without a prosperous software ecosystem, on-board computers use spatial redundancy of commercial off-the-shelf processors to make the whole system fault-tolerant, powerful and energy-efficient. Fuchs et al. [9, 10] investigated how to combine duplicating applications on independent cores, topological features, error correction coding and reconfiguration of MPSoC to achieve software-implemented fault-tolerance on a set of Xilinx

UltraScale and UltraScale+ devices using ARM Cortex-A53 processors. Another example of application can be found in [23], where the authors considered the reliability of avionic applications, based on redundancy and partitioning principles of multi- or many-core processors. To validate the approach, they used a 256-core commercially available processor as a test platform.

In this paper, we tackle the problem of streaming applications that would be executed on such failure-prone hierarchical platforms, with the objective of minimizing the energy consumption under several constraints. To the best of our knowledge, this is the first attempt to address this problem.

3. Model

We first detail the application model in Section 3.1. We then describe the target computing platforms in Section 3.2. The key to our approach is to partition the graph, while preserving its structure, as explained in Section 3.3. Section 3.4 focuses on the errors and explains how the use of triplication helps dealing with soft errors. The energy model is detailed in Section 3.5, and we introduce the target period (constraint on execution time) in Section 3.6. Finally, we formalize the optimization problem in Section 3.7.

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, as in typical workflow applications [24], for instance the StreamIt benchmark [6]. 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, which is a very common assumption for streaming applications. Several applications are even simpler and consist in a linear chain of task. The nodes of the graph correspond to different application tasks, and they 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. $Succ(i)$ is the set of successors of T_i, hence $j \in Succ(i)$ if and only if $L_{i,j} \in \mathcal{E}$. For $1 \leq i \leq n$, w_i is the computation requirement of task T_i (in floating point operations), 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 $\frac{1}{P_t}$. We will further discuss how to compute this processing rate in Section 3.6.

3.2. Platforms

The target computing platform consists of homogeneous cores, where 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. Each core is operating at a constant speed; we assume that we cannot change the speed of the core during the execution, but different cores may operate at different speeds. The set of speeds is $\{s_{\min} = s_1, s_2, ..., s_k = s_{\max}\}$.

The cores are organized by a hierarchical communication system: there are c blocks, with p computing cores per block, hence a total of $c \times p$ cores. Within a block, the cores 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 (resp. in different blocks) is denoted as β_1 (resp. β_2). Communication among cores in the same block is cheaper than that among different blocks [25], i.e., $\beta_1 >> \beta_2$.

3.3. Graph partitioning and structure rule

In order to achieve load balance and 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).*

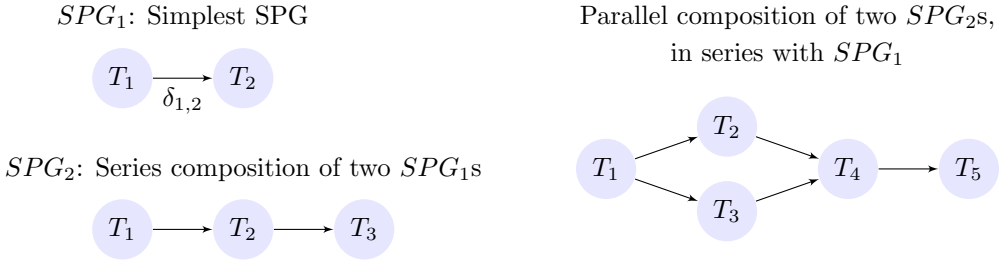


Figure 1: SPG examples.

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* [24].

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 the 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 $T_2, T_4, \ldots T_k$ are in another part, as depicted in Fig. 2.

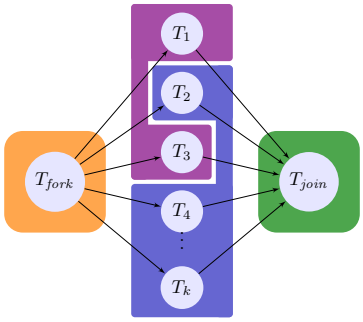


Figure 2: Fork-join graph and a partition following the structure rule.

A parallel composition of more complex subgraphs is depicted in Figure 3 between tasks T_1 and T_{16}. In the proposed partition, two subgraphs of the parallel composition are grouped together (in the 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 series-parallel subgraph. Note that, by construction, each part of a partition following the structure rule has either a single source vertex and sink vertex (i.e., in the cases (i) and (ii) of the definition), or it has a single predecessor and a single successor (i.e., case (iii)).

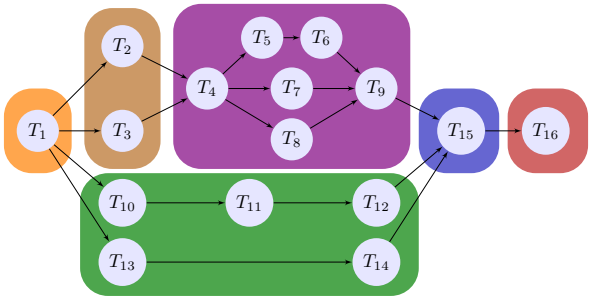


Figure 3: SPG partition following the structure rule.

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 $S(v)$. The set of indices of tasks that are mapped on a block d of cores is denoted as ℓ_d. This set ℓ_d 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. 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 (i.e., $|Source_v| = |Sink_v| = 1$), 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. We consider the use of DVFS (Dynamic Voltage and Frequency Scaling), hence it is possible to decide at which speed a core is operating. However, a very small decrease of speed leads to an exponential increase of failure rate [7, 8]. 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, and d is usually set as 4 or 5 [26]. 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 [27].

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, with $d = 4$, the failure rate at minimum speed is 5.46×10^{-4} per hour. As a result, 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.

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 the 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 the bound on the period, P_{t}. 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 [15]. The energy consumption of executing a data item through all tasks is composed of both 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 [16]. After taking triplication into consideration, the energy cost of the whole application on one data item is as follows:

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

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 from Section 7: $s_{\min}$ and $s_{\max}$ are 1GHz and 2.5GHz respectively, the static power is 0.02W, and $C = 1$. Assume that the task's computational requirement is 1.2 GFLOP and the period is 1.2 second (hence, the task can be executed at speed $s_{\min}$ within the target period of 1.2 second). The energy needed for triplicating the task at speed $s_{\min}$ is $3 \times (0.02 + 1.2 \times 1^2) = 3.66W$, while running the task only once at speed $s_{\max}$ requires an energy of $0.02 + 1.2 \times 2.5^2 = 7.52W$.

The energy cost of the communication cannot be neglected in our model. Within a block, communications among processor cores are done through remote memory accesses. Communications between two cores of different blocks are done through routers on the 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. This cost $\alpha_{i,j}$ depends on where tasks are located: if tasks 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 [28] for details.

Meanwhile, we need to 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 on 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. Also, $\beta_{u,v} = \beta_1$ 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. 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 twice the time of a single transfer ($m_i - 1 = 2$ in this case). The second and third terms are the time needed to send and receive datasets.

The throughput of the application $\mathcal{G}$ is therefore constrained by the maximum time taken on a processor, corresponding to the period of the mapping, and expressed as:

$$T(\mathcal{G}) = \max_{1 \leq v \leq c \times p} T(v).$$

In order for the mapping to be valid, this period has to be smaller 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 therefore to decide which tasks to group in a same part, and then, for each part, whether it is triplicated or not, on which core (or cores if triplicated) it is executed, and at which speed. More formally, the problem is defined as follows:

Definition 2. (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, on which core(s) and at which speed it is executed, so that the total expected energy consumption is minimized. There are two constraints: the actual period $T(\mathcal{G})$ should not exceed the period bound P_t (to ensure required performance), and each task is either executed at maximum speed or it is triplicated (to ensure reliable execution).

4. Problem complexity

Similarly to existing partitioning problems, the MinEnergy optimization problem unsurprisingly turns out to be NP-complete. We consider the decision version of MinEnergy, i.e., with a bound on the total expected energy consumption. Then, we establish its NP-completeness as follows:

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

Proof. First, note that given a mapping of the tasks on the processors, it is possible to verify in polynomial time that (i) each partition follows 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. Hence, the problem is in NP.

To prove the NP-hardness of the problem, we perform a reduction from 3-Partition, which is known to be NP-complete in the strong sense [29]. We consider the following instance $\mathcal{I}_1$ of the 3-Partition problem: let $\{a_1, \ldots, a_{3m}\}$ be $3m$ integers, and B be 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 answer 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 three tasks T_i with $i \in S_k$ are mapped on a same processor (no triplication is used). Overall, 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. This mapping is similar to the one depicted in Figure 2 (with three tasks per part except for the source and sink tasks), and therefore, it follows the structure rule.

Reciprocally, assume that there exists a solution to problem $\mathcal{I}_2$, i.e., 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 to 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 $\sum_{i \in S_k} a_i \leq B$. Finally, since $\sum_{i=1}^{3m} a_i = mB$, we have $\sum_{i \in S_k} a_i = B$. Hence, the subsets S_k $(1 \leq k \leq m)$ provide a solution to $\mathcal{I}_1$. □

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 in the form of 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 first describe the dynamic programming algorithm in Section 5.1. Next, we show through some examples in Section 5.2 that it might not be optimal in some cases, since we focus on contiguous allocation. However, we prove optimality of the dynamic programming algorithm for a special class of mappings, called *monotonic mappings*, as shown in Section 5.3.

5.1. Dynamic programming (DP) formulation

Assume that we have $c^* \leq c$ available blocks, where $c^* - 1$ blocks are fully available (p cores available), while the last block has $p^* \leq p$ cores available. We recursively express the minimum energy cost of scheduling tasks T_1 to T_i onto these available cores, denoted as $E(i, c^*, p^*)$. 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.

Hence, $E(n, c, p)$ is the minimum energy cost of scheduling all tasks, when all blocks and all cores are

available. The recursion can be formulated as:

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

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. Hence, the two first lines represent the cases where all remaining tasks are mapped as a single part, at maximum speed or triplicated. The third line aims at creating a part with tasks T_{j+1} to T_i, and recursively mapping the j first tasks.

$E_c(j, \alpha, \beta, m)$ 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 is the energy cost of transferring a unit of data, and β is the bandwidth (these values depend on whether tasks are in a same block or not), and m indicates whether task T_{j+1} is triplicated (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 they 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:

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

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 energy cost is infinite if there are less than three cores available, or no block left, or if the period bound cannot be matched:

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

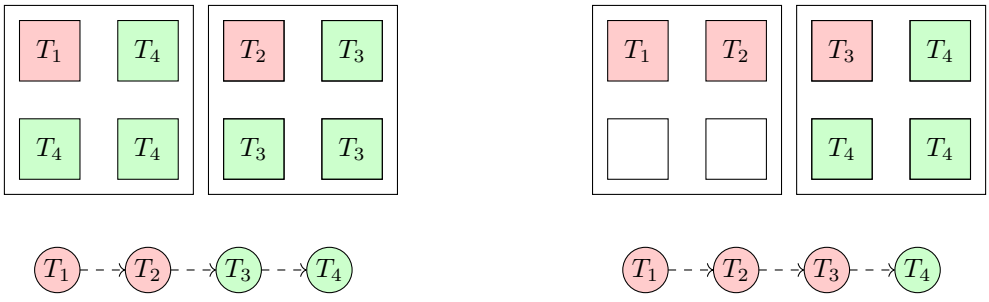
Finally, for E_c, the energy cost is infinite if the communication time is larger than the period, otherwise it is computed as indicated in Section 3.5:

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

5.2. Examples where DP 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 on-chip communications 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., $I_s V_s = 1$), and the 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 sizes of all edges are 0.1. The period bound is $P_{\mathrm{t}} = 1$.

The optimal partition and mapping, as shown in Fig. 4a, is creating one part per task (a communication is done between each task). The first two tasks are running at the maximum speed and are mapped onto two different blocks, each on a core. The third and fourth 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 costs of running tasks are $64.1, 64.1, 3.9, 3.9$ for T_1 to T_4, and communication energy costs are $0.2, 0.3, 0.6$ between tasks).



(a) The optimal mapping.

(b) Mapping returned by DP.

Figure 4: Example where DP is not optimal. Red tasks are executed at the maximum speed, while green tasks are triplicated and run at speed s_s.

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 the 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. As shown in Fig. 4b, 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 having 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 a solution.

5.3. Condition for optimality

We first define a special class of mappings: if the indices of blocks where tasks are mapped are monotonically increasing with the task indices (tasks $T_1, \ldots, T_n$), the mapping is said to be *monotonic*:

Definition 3 (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 $d \leq f$.*

We prove that the dynamic programming algorithm (computing $E(n, c, p)$) returns an optimal monotonic mapping, even though there might be a better non-monotonic mapping as highlighted in Section 5.2.

Lemma 1. *The previous dynamic program returns a mapping whose energy cost is minimal among monotonic mappings.*

Proof. We prove that for any i, c^*, p^* with $1 \leq i \leq n$, $0 \leq c^* \leq c$, $0 \leq p^* \leq p$, $E(i, c^*, p^*)$ finds a mapping whose energy cost is minimal among monotonic mappings. In particular, $E(n, c, p)$ hence returns an optimal monotonic mapping. The proof is done by induction.

We first prove that for an application composed of a single task T_1, the solution given by the formula is optimal. We have $E(1, c^*, p^*) = \min(E_m(1, 1, c^*, p^*), E_t(1, 1, c^*, p^*))$, hence the solution returned is the minimum between the energy cost of running T_1 at the maximum speed, and that of triplicating T_1 at speed s_s. Since cores and blocks are homogeneous, the mapping can be done on any core for the same energy, and we consider all possibilities hence obtain the optimal mapping.

Next, we assume that for applications that have at most k tasks, $k \leq i-1$, $E(k, c^*, p^*)$ returns an optimal monotonic solution (for any $c^* \leq c$ 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 available on the last block.

We consider an optimal monotonic mapping M_{opt}. If there is a single part in this optimal mapping (i.e., all tasks are mapped on a single core or on three cores), then the solution either triplicates this part, or executes it at maximum speed, which corresponds to the two first lines of the DP, and hence $E(i, c^*, p^*) \leq E_{opt}$, proving the optimality of DP.

Otherwise, we assume that the last edge separating tasks mapped on different cores is $L_{j,j+1}$ (with $1 \leq j \leq i-1$). Therefore, tasks from T_{j+1} to T_i form a part that is mapped on one or three cores. The energy cost of this mapping is E_{opt}. We assume that task T_j is assigned to block c_t ($c_t \leq c^*$), and tasks $T_1, \ldots, T_j$ use p_t cores on this block. The energy cost of tasks $T_1, \ldots, T_j$ in M_{opt} is denoted by E_{left}, and by induction, since the mapping returned by $E(j, c_t, p_t)$ is optimal, we have $E_{left} \geq E(j, c_t, p_t)$.

Since the mapping is monotonic, in M_{opt}, tasks T_{j+1} to T_i can use only a block with index higher than or equal to c_t. We then consider the different ways these tasks can be mapped.

- If T_{j+1} to T_i are running at the maximum speed on a core of block c_t, assuming $p - p_t \geq 1$ (or $p^* - p_t \geq 1$ if $c_t = c^*$), then the energy cost of this part is $E_m(j+1, i, c^*, p^*)$. 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 $E_{left} + E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 1) + E_m(j+1, i, c^*, p^*)$.

- If T_{j+1} to T_i are running at the maximum speed on a core of a block c_r, with $c_t < c_r \leq c^*$, then the energy cost of this part is $E_m(j+1, i, c^*, p^*)$. This can happen if $c_t < c^*$, and if $c_r = c^*$, we also must have $p^* \geq 1$ (at least one core available on a block to the right of c_t). In this case, 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 $E_{left} + E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 1) + E_m(j+1, i, c^*, p^*)$.

- If tasks $T_{j+1}, \ldots, T_i$ are triplicated on block c_t (if there are at least three cores available), we obtain similarly a total energy cost $E_{left} + E_c(\alpha_1, \beta_1, \delta_{j,j+1}, 3) + E_t(j+1, i, c^*, p^*)$.

- If tasks $T_{j+1}, \ldots, T_i$ are triplicated on block c_r, with $c_t < c_r \leq c^*$, with at least three cores available on this block, then we have $E_{left} + E_c(\alpha_2, \beta_2, \delta_{j,j+1}, 3) + E_t(j+1, i, c^*, p^*)$.

These are the only possibilities for mapping tasks $T_{j+1}, \ldots, T_i$, hence the optimal solution consists of one of the four cases above. These cases are exactly the ones considered in the DP formulation, and DP selects the choice with minimum total energy. Furthermore, recall that by induction, $E_{left} \geq E(j, c_t, p_t)$. Finally, since cores and blocks are homogeneous, DP will return a mapping that does not use more energy than the optimal one. □

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. The other heuristics use a two-step approach to map the SPG onto the platform. The first step is to partition the graph into parts, and the second step consists in mapping these parts onto the computing resources. In Sections 6.2 and 6.3, we propose two heuristics that focus on partitioning the graph into parts, and select the most energy efficient way of execution (maximum speed vs triplication), while the baseline heuristic executes all tasks at maximum speed. 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 basis for comparison. Each task is executed at the maximum speed $s_{\max}$, and then tasks are greedily mapped 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 the total workload 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 the 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; it is calling the procedures from Algorithms 1 and 2. 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 block $c-1$.

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

```
if p* > 1 then
    p* ← p* − 1;
else if c* > 1 then
    c* ← c* − 1, p* ← p;
else
    return < 0, 0, ∅ >;
end if
return < c*, p*, ∅ >;
```

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

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

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

6.2. Partitioning heuristic – GROUPCELL

Heuristic GROUPCELL partitions the graph in a bottom-up way. It first breaks all edges, except (i) edges with a large communication cost, which cannot be done within the period, i.e., such that $\delta_{i,j} \geq \beta_1 P_{\mathrm{t}}$; 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 cost. 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 resulting 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.

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

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 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 to 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($\mathcal{G}, c, p, P_{\mathrm{t}}$)

```
1: set L ← all fork and join nodes of G;
2: Parts ← break output edges of fork and input edges of join in L;
3: C ← ∅; /*edges broken*/
4: repeat
5:    part ← pop up the first element of Parts;
6:    < i, j >← source node and sink node of part;
7:    < E, C_cur >← DP(i, j, c, p);
8:    if E == +∞ then
9:       return failure;
10:   end if
11:   C ← C ∪ C_cur;
12: 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 the 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

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

```
1: b_cur ← c; L ← ∅;
2: construct quotient graph Q by breaking edges in C;
3: initialize Sets with c empty vectors;
4: C' ← edges of Q whose δ_{i,j} > β_2 P_t;
5: for i = 1 to i = |C'| do
6:    < T_p, T_s > ← nodes connected by edge C'[i];
7:    if T_p is in L but not T_s then
8:       push T_s into the same vector as T_p;
9:    end if
10:   if T_s is in L but not T_p then
11:      push T_p into the same vector as T_s;
12:   end if
13:   if neither T_p nor T_s is in L then
14:      initialize a vector with them, push it into L;
15:   end if
16: end for
17: while L ≠ ∅ do
18:   vec ← pop up the first vector of L;
19:   nbr ← Σ_{i∈vec} m_i; /*cores requested*/
20:   if nbr > p then
21:      sort vec by an non-decreasing order of weight;
22:      idx ← 1;
23:      while nbr > p do
24:         set the node vec[idx] to run on only one core;
25:         idx ← idx + 1; recaculate nbr;
26:         if idx > |vec| then return failure;
27:      end while
28:   end if
29:   if p - |Sets[b_cur]| < nbr then b_cur ← b_cur − 1;
30:   for i = 1 to i = |vec| do
31:      if node vec[i] needs 3 processors then
32:         push it into vector Sets[b_cur] three times;
33:      else
34:         push it into vector Sets[b_cur];
35:      end if
36:   end for
37: end while
38: MapTopology(Q, Sets);
```

on. MapTopology is called afterwards to map the remaining parts in a sequential order. 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 is 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 fails, MapTopology then merges linear chains already mapped from the smallest size until there is enough space.

Algorithm 7 MapTopology($Q, Sets$)

```
1: L ← a depth-first traversal of Q;
2: repeat
3:    T_i ← pop up L; b ← c;
4:    if T_i has not been mapped yet then
5:       if T_i has a predecessor then
6:          b_cur ← which block the first predecessor of T_i mapped onto;
7:       end if
8:       if p − |Sets[b]| < m_i and the size of first input edge is larger than β_2 P_t then
9:          h ← the index such that all input edges of Sets[b][h] are not larger than β_2 P_t;
10:         if h exists and b > 1 then
11:            move elements between Sets[b][h] and the end of Sets[b] to Sets[b − 1];
12:            b = b − 1;
13:         else
14:            while p − |Sets[b]| > m_i do
15:               part_j ← the smallest part of Sets[b] who has only one predecessor that is not a fork;
16:               merge part_j to its predecessor and remove part_j from Sets[b];
17:            end while
18:         end if
19:      end if
20:      while p − Sets[b] ≥ m_i and b > 1 do
21:         b = b − 1;
22:      end while
23:      if m_i == 3 and p − |Sets[b]| ≥ 3 then
24:         put part_i into Sets[b] three times;
25:      else
26:         put part_i into Sets[b];
27:      end if
28:   end if
29: 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 [6] for testing the **StreamIt** compiler. It collects a wide spectrum of 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. Here, 44 applications are selected, in which 10 of them are linear chains, and the list of applications with their main parameters is available in Table 1.

We base the platform parameters on the characteristics of the ARM's Cortex-A15 Processor [30, 31]: the possible core frequencies (GHz) are $\{s_{\min} = 1, 1.4, 1.5, 1.8, 2, 2.5 = s_{\max}\}$, and the idle power of each core is 0.02W. The Cortex-A15 is a multi-core processor, which can have up to 4 blocks, and each block has up to 4 cores. Numerous SoCs from mobile products, such as Samsung Exynos series, HiSilicon K3V3 and Texas Instruments Sitara feature Cortex-A15 cores [32]. Some other Cortex-A series processors can have up to 8 cores per block, and according to the target applications, a SoC can have several (different) ARM processors. Furthermore, some SoCs for multimedia or avionic applications on embedded systems can have up to 16×16 cores: 16 blocks, each with up to 16 cores [33], or a combination of 2 blocks, each with 32 cores [34]. Hence, we consider in this section a computing system that has up to 4 blocks, with up to 64 cores in each block. To simulate applications with various communication to computation ratios (CCRs), 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 $\beta_2 = \beta_1/16$. We do not set the units of the edge sizes in Table 1 and bandwidth since we focus on the CCR. The parameters α_1 and α_2 are set to $0.05W$ and $0.2W$ respectively. Finally, the switching capacitance C is set to 1.

For each application, we set the period bound $P_{\mathrm{t}} = a + \kappa(b - a)$. 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 maximum time needed to process all tasks on a core at the minimum speed or a data transfer at speed β_2 ($b = \max(\sum_{1 \le i \le n} w_i/s_{\min}, \delta_{i,j}/\beta_2)$, corresponding to a very loose period bound. We set κ to values from 0.1 to 0.9, with an increment of 0.1. 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. Results on the number of such failures are depicted in Section 7.2.2. For the other results, 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.

Name	Nb Nodes	Nb 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	

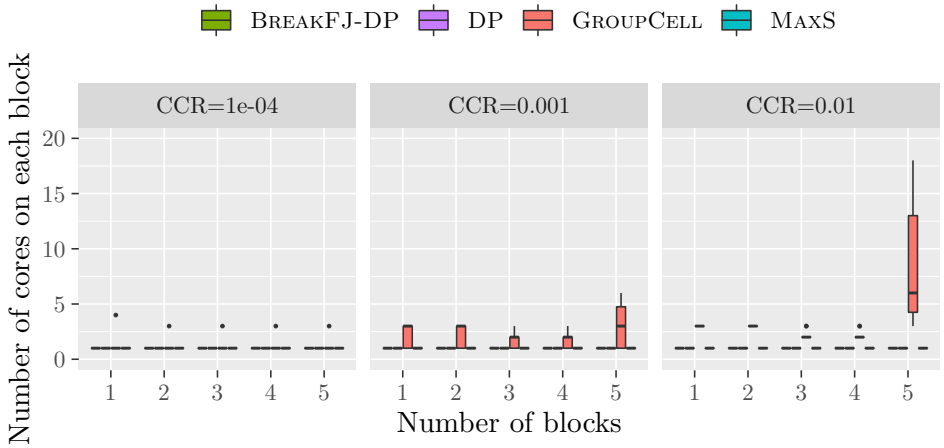
Table 1: Parameters of the streaming applications. The unit of nodes' weight is GFLOP.

7.2. Simulation results

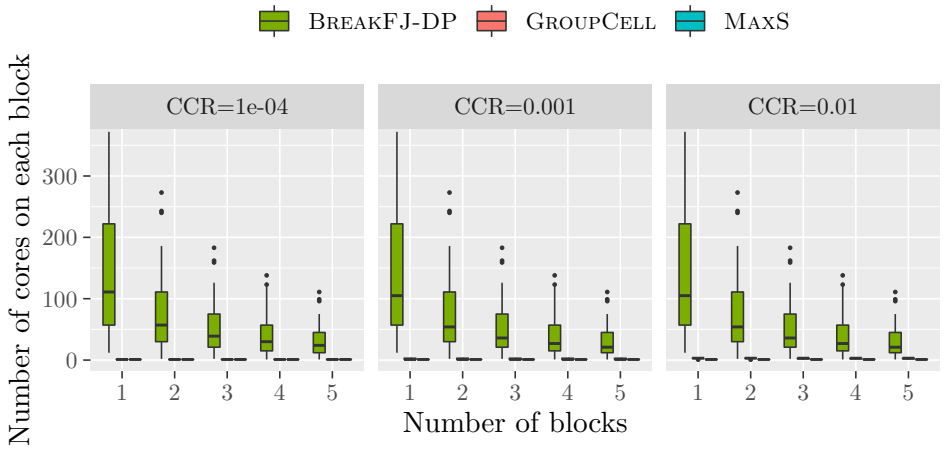
We first plot the number of cores that are used by the heuristics. Next, we report the percentages of failure cases of the heuristics, i.e., cases where heuristics do not succeed to find a valid solution, because some constraints could not be respected. We then present the key results, which are the energy costs of the heuristics as a function of the parameters. Finally, we report the execution time of the heuristics.

7.2.1. Minimum number of cores requested by the heuristics

After removing the cases where a heuristic does not find a valid solution, given a number of blocks and cores, Fig. 5 shows the minimum number of cores on a block requested by each heuristic, with various number of blocks provided, where κ is set to 0.5, which corresponds to a median value. This provides information whether some heuristics need more cores than others to be able to provide a valid solution.



(a) Linear chain



(b) General SPG

Figure 5: Number of cores requested by each block with different CCRs and number of blocks provided.

On linear chains, as shown in Fig. 5a, dynamic programming (denoted as DP), MaxS as well as BreakFJ-DP have the same performance, they require the least number of cores, one core for all cases. GroupCell requires on average 2.51 times more cores than DP.

On general SPGs, as shown in Fig. 5b, BreakFJ-DP uses far more cores than GroupCell and MaxS. For instance, with CCR=10^{-3}, BreakFJ-DP uses 39 times more cores than GroupCell on average. GroupCell requests 2 cores on average. Similarly as for linear chains, MaxS requests only one core in all cases.

7.2.2. Failure cases

We report percentage of failure cases in Figures 6 and 7. Recall that, in total, there are 34 general SPGs and 10 linear chains. On linear chains (Fig. 6), GroupCell fails far more than the other heuristics, in particular with few cores per blocks, cheap communication costs, or a loose period bound. However, the other heuristics have very few failure cases. MaxS, BreakFJ-DP and DP never fail, except in some cases where the period bound is very tight ($\kappa = 0.1$), with two blocks and two or more cores per block.

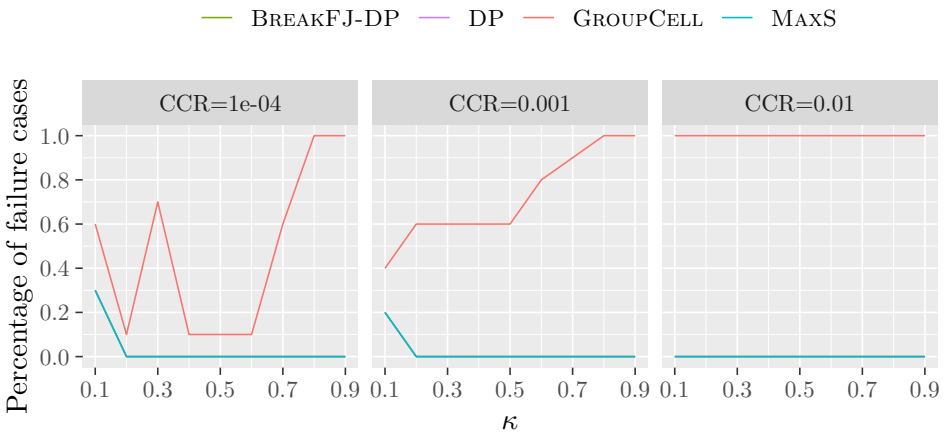
The percentages of failure cases on general SPGs are shown in Fig. 7. BreakFJ-DP is the heuristic that fails the most, since it requests more cores. When communications are not so expensive (CCR=10^{-4} or 10^{-3}), the percentage of failure cases of BreakFJ-DP is around 69% with 16 cores per block, and it decreases to 47% with 32 cores, and then to 21% with 64 cores. The same happens for the other heuristics: if there are more cores on a block, the heuristics fail on fewer applications. Even with a tight communication bandwidth (CCR=10^{-2}) and a tight period bound ($\kappa \leq 0.2$), less than 50% of the applications fail when using GroupCell. Here again, MaxS has very few failure cases.

Note that a *failure* means that the heuristic does not succeed in finding a mapping of the application such that all constraints are respected. One should then consider increasing the platform bandwidth if the application has high communication requirements, or relaxing the bound on the period.

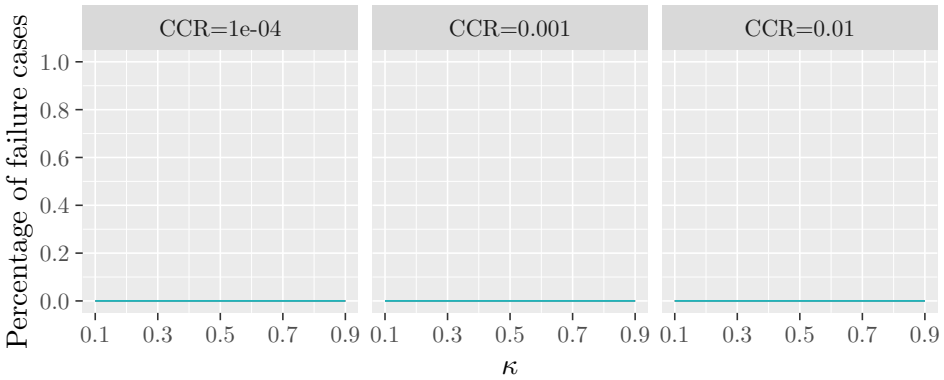
7.2.3. Energy cost

Figures 8 and 9 depict the energy cost as a function of κ, where a smaller κ represents a tighter period, with different number of blocks and cores given. The value $\kappa = 0.5$ represents a median period bound between the tightest one a and the loosest one b.

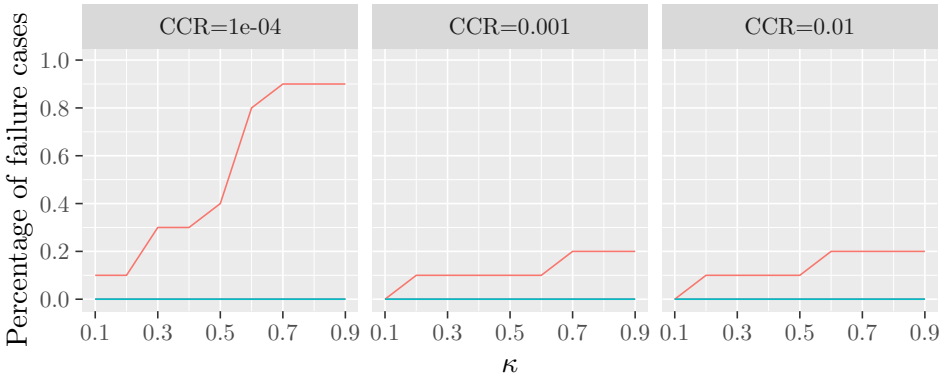
We first focus on linear chains, see Fig. 8. Recall that there are ten linear chain applications, and we consider a platform with two blocks. BreakFJ-DP and DP always return the same result. We do not consider two cores per block, since there were too many failures (see Figure 6a). When there are four cores per block (Fig. 8a), all 10 applications are included. BreakFJ-DP (resp. GroupCell) reduces the energy cost by 42% (resp. 28%) average compared to MaxS, and around 52% (resp. 49%) when communications are expensive. As expected, the algorithm building on the dynamic programming formulation produces very good results, even though it restricts to contiguous mappings. With more cores (8 cores per block, see



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

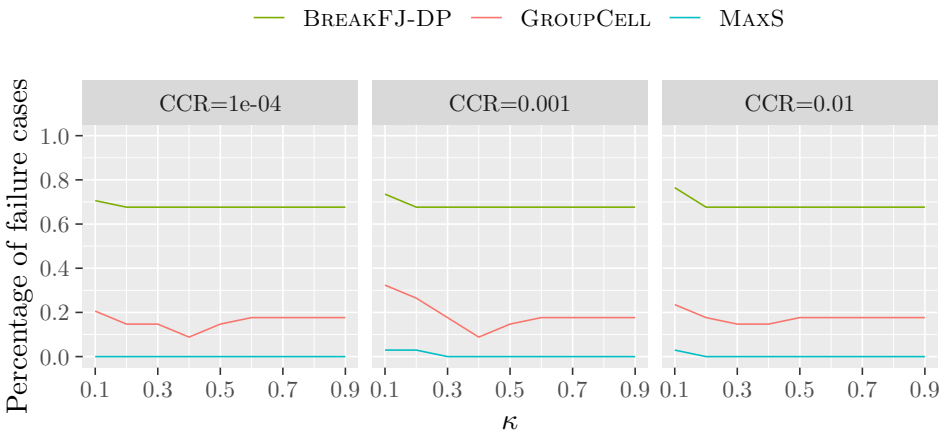


(b) Linear chains. 2 blocks, each with 4 cores provided. BreakFJ-DP and DP are covered by MaxS.

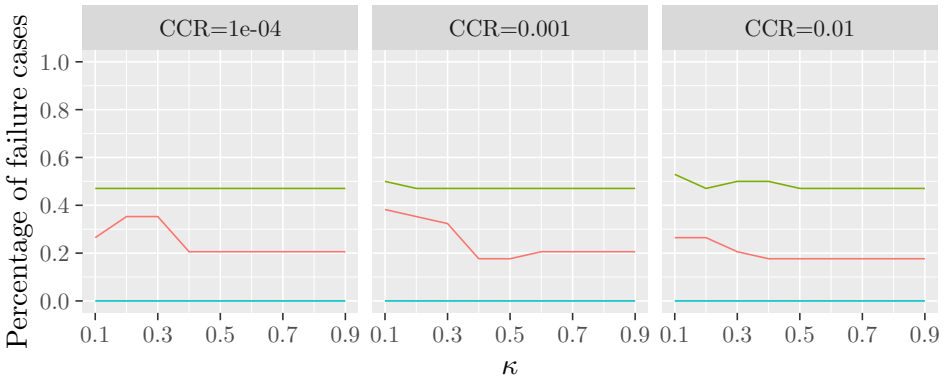


(c) Linear chains. 2 blocks, each with 8 cores provided. BreakFJ-DP and DP are covered by MaxS.

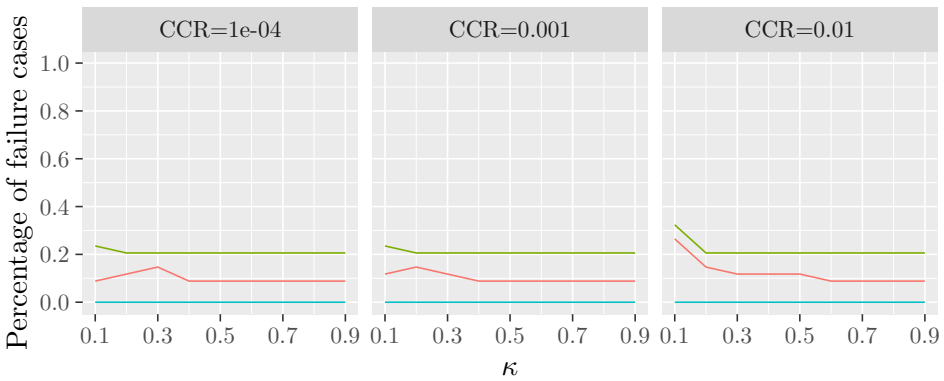
Figure 6: Percentage of failure cases on linear chains as a function of κ.



(a) General SPGs. 4 blocks, each with 16 cores provided.

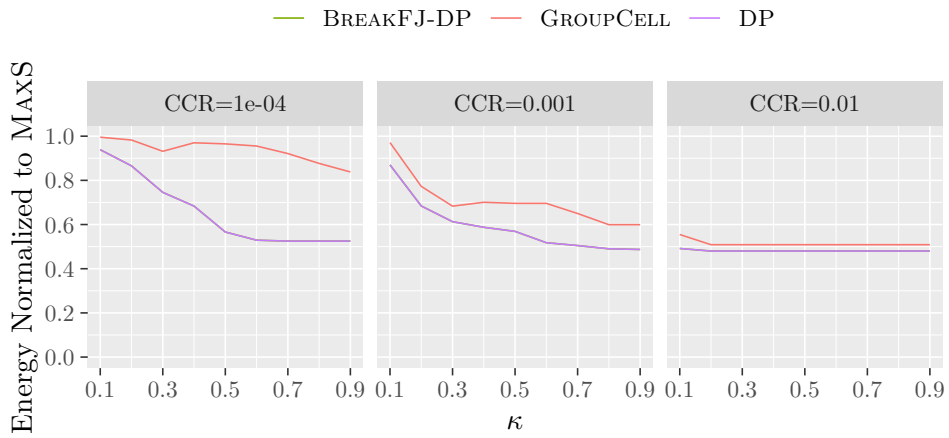


(b) General SPGs. 4 blocks, each with 32 cores provided.

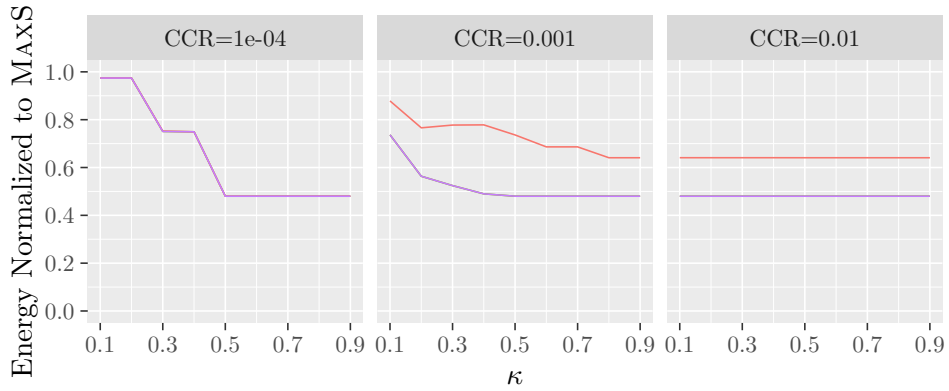


(c) General SPGs. 4 blocks, each with 64 cores provided.

Figure 7: Percentage of failure cases on general SPGs as a function of κ.



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

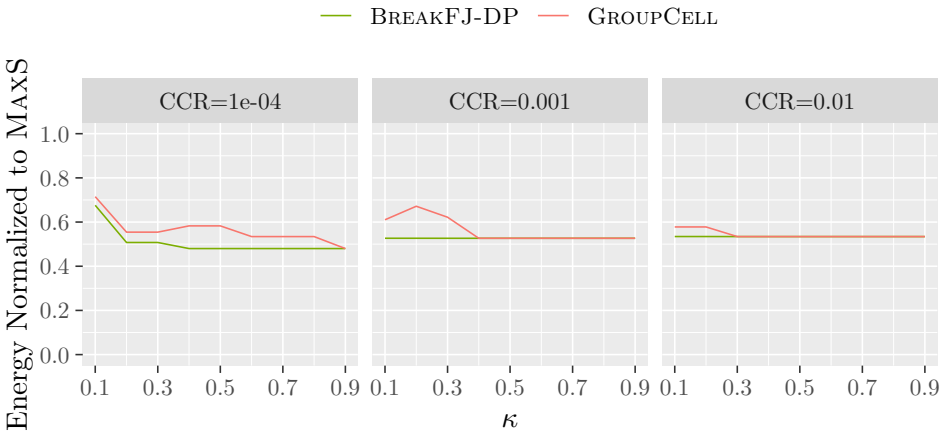


(b) BreakFJ-DP and DP give the same results, hence only DP is visible, 2 blocks, each with 8 cores provided.

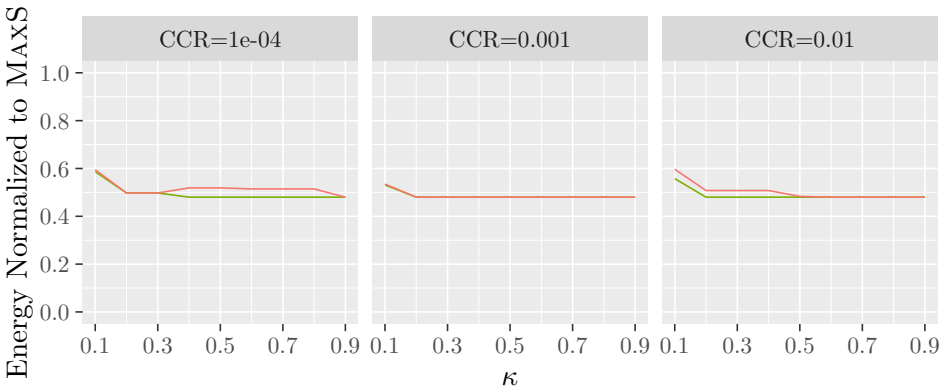
Figure 8: Energy consumption on linear chains relative to MaxS as a function of the period bound tightness κ.

Fig. 8b), the ranking of heuristics is the same. BreakFJ-DP and GroupCell reduce the energy cost by 49% and 32% respectively on average compared to MaxS. For CCR=10^{-3} and 10^{-2}, eight applications are included. However, note that when CCR=10^{-4}, the results only include one application out of ten. Indeed, for these applications, GroupCell creates five parts to be triplicated (hence requesting a total of 15 cores, less than the total of $2 \times 8 = 16$ cores), but does not account for the fact that only 4 parts can be triplicated on the platform. Indeed, recall that triplication cannot be shared between two blocks. Hence, surprisingly, GroupCell has more failures than with fewer cores, where less parts are generated.

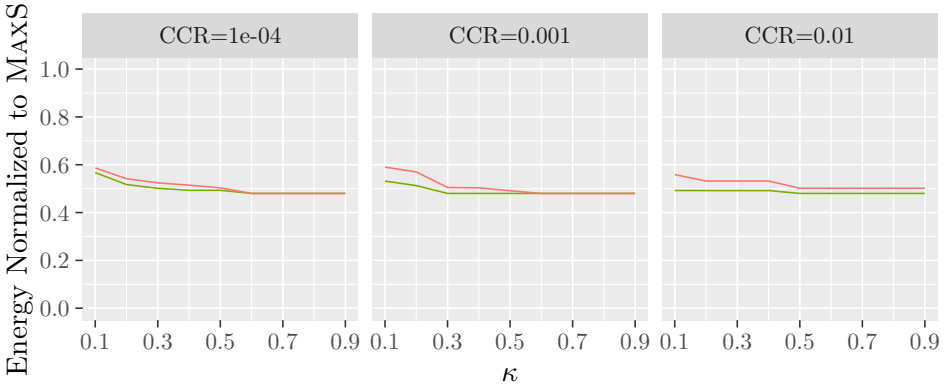
The gains are also very impressive for general SPGs, see Fig. 9. We now consider a platform with four blocks. With 64 cores per block (Fig. 9c), both heuristics save more than 41% of energy cost in all settings, with BreakFJ-DP being slightly better by at least 2.4% in most cases. 23, 21, and 20 applications (out of



(a) 4 blocks, each with 16 cores provided.



(b) 4 blocks, each with 32 cores provided.



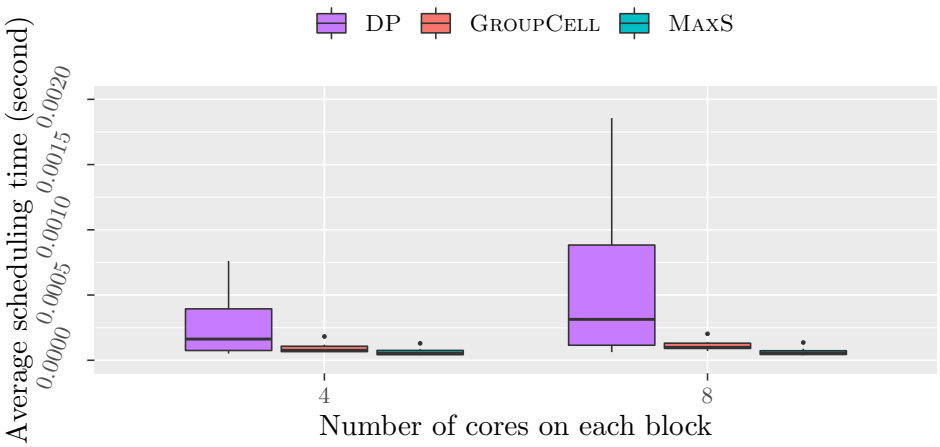
(c) 4 blocks, each with 64 cores provided.

Figure 9: Energy consumption on general SPGs relative to MaxS as a function of the period bound tightness κ.

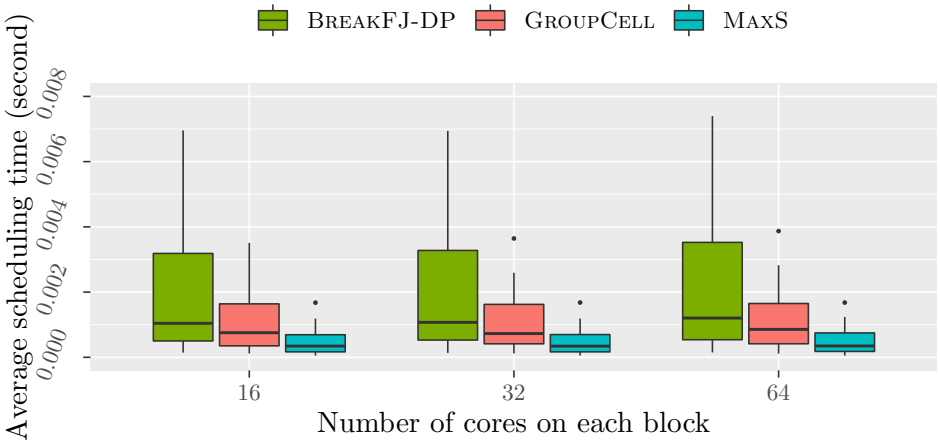
the 34 applications that are not chains) are included for CCR = 10^{-4}, 10^{-3}, and 10^{-2} respectively, since at least one heuristic does not return a valid mapping on other applications. With 32 cores per block (Fig. 9b), BreakFJ-DP and GroupCell still outweigh MaxS by around 50% when CCR=10^{-4}. BreakFJ-DP is still slightly better than GroupCell. 11, 9, and 11 applications are included when CCR=10^{-4}, 10^{-3}, and 10^{-2} respectively. We observe the same trend for the setting with 16 cores per block (Fig. 9a). 7, 7, and 6 applications are included when CCR=10^{-4}, 10^{-3}, and 10^{-2} respectively.

7.2.4. Execution time

The average execution time of all heuristics is shown in Fig. 10 as a function of the number of cores



(a) Linear chains.



(b) General SPGs.

Figure 10: Average scheduling time on 27 cases (9 period bounds and 3 different CCRs).

on each block. The time is an average on 27 running cases: for a given graph, we run each heuristic for a combination setting of 9 period bound options and 3 CCRs. The time cost is of the order of 10^{-4} (resp. 10^{-3}) seconds for chains (resp. general SPGs), which is obviously very fast. MaxS is the fastest heuristic, then followed by GroupCell, and DP or BreakFJ-DP is the slowest one. On linear chains, the execution time of DP increases with the number of cores as the search space is increasing. Running all experiments takes less than one minute.

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 47% savings overall. When the period bound is not too tight, it achieves 68% savings on average over all experiments. However, this heuristic may fail with limited number of cores per blocks. In this case, the GroupCell heuristic is a promising alternative, with only a slightly greater energy consumption for a reduced number of cores used.

As future work, we plan to further study the dynamic programming algorithm, to determine whether it is an approximation algorithm. Indeed, 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. On the practical side, we plan to propose a variant of GroupCell to avoid cases where no mapping can be found, because too many parts should be triplicated and cores may not be used. We should hence be able to design a more robust heuristic, building on the ideas proposed in the present work. Other promising research directions would be to extend the study to general applications, rather than series-parallel graphs, and consider general mappings instead of the structure rule. While it might be difficult to derive theoretical results for these extensions, some efficient heuristics could be designed and analyzed. An experimental validation on a real platform would also be very insightful.

Acknowledgements: We would like to thank the anonymous reviewers for their comments and suggestions, which greatly helped improve the paper. This work is supported by National Key Research and Development Program of China 2018YFB2101300, and the Natural Science Foundation of China 61872147.

References

References

[1] A. Dolgert, L. Gibbons, C. D. Jones, V. Kuznetsov, M. Riedewald, D. Riley, G. J. Sharp, P. Wittich, Provenance in high-energy physics workflows, Computing in Science Engineering 10 (3) (2008) 22–29. doi:10.1109/MCSE.2008.81.

[2] J. Deslippe, A. Essiari, S. J. Patton, T. Samak, C. E. Tull, A. Hexemer, D. Kumar, D. Parkinson, P. Stewart, Workflow management for real-time analysis of lightsource experiments, in: Proc. of WORKS'14, 2014, pp. 31–40. doi:10.1109/WORKS.2014.9.

[3] CMS Collaboration, CMS data processing workflows during an extended cosmic ray run, Journal of Instrumentation 5 (03) (2010) T03006–T03006. doi:10.1088/1748-0221/5/03/t03006.

[4] A. Luckow, G. Chantzialexiou, S. Jha, Pilot-streaming: A stream processing framework for high-performance computing (2018). arXiv:1801.08648.

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

[6] W. Thies, Language and compiler support for stream programs, Ph.D. thesis, MIT, Cambridge, MA, USA (2009).

[7] D. Zhu, Reliability-aware dynamic energy management in dependable embedded real-time systems, ACM Trans. on Embedded Computing Systems 10 (2011) 26:1–26:27.

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

[9] C. M. Fuchs, P. Chou, X. Wen, N. M. Murillo, G. Furano, S. Holst, A. Tavoularis, S.-K. Lu, A. Plaat, K. Marinis, A Fault-Tolerant MPSoC For CubeSats, in: 2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2019, pp. 1–6. doi:10.1109/DFT.2019.8875417.

[10] C. M. Fuchs, N. Murillo, A. Plaat, E. V. der Kouwe, D. Harsono, T. Stefanov, Fault-tolerant nanosatellite computing on a budget (2019). arXiv:1903.08781.

[11] A. Samaras, F. Bezerra, E. Lorfevre, R. Ecoffet, CARMEN-2: In flight observation of non destructive single event phenomena on memories, in: 2011 12th European Conference on Radiation and Its Effects on Components and Systems, 2011, pp. 839–848. doi:10.1109/RADECS.2011.6131314.

[12] Q. Tang, S. Wu, J. Shi, J. Wei, Optimization of Duplication-Based Schedules on Network-on-Chip Based Multi-Processor System-on-Chips, IEEE Transactions on Parallel and Distributed Systems 28 (3) (2017) 826–837. doi:10.1109/TPDS.2016.2599166.

[13] M. Flasskamp, G. Sievers, J. Ax, C. Klarhorst, T. Jungeblut, W. Kelly, M. Thies, 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, RAPIDO '16, ACM, New York, NY, USA, 2016, pp. 3:1–3:6. doi:10.1145/2852339.2852342.
URL http://doi.acm.org/10.1145/2852339.2852342

[14] A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, M. J. Garzarán, Mapping Streaming Applications on Commodity Multi-CPU and GPU On-Chip Processors, IEEE Transactions on Parallel and Distributed Systems 27 (4) (2016) 1099–1115. doi:10.1109/TPDS.2015.2432809.

[15] G. Onnebrink, F. Walbroel, J. Klimt, R. Leupers, G. Ascheid, L. G. Murillo, S. Schürmans, X. Chen, Y. Harn, DVFS-enabled power-performance trade-off in MPSoC SW application mapping, in: SAMOS'17, 2017, pp. 196–202. doi:10.1109/SAMOS.2017.8344628.

[16] M. A. Haque, H. Aydin, D. Zhu, On reliability management of energy-aware real-time systems through task replication, IEEE TPDS 28 (3) (2017) 813–825. doi:10.1109/TPDS.2016.2600595.

[17] J. Zhou, T. Wei, M. Chen, J. Yan, X. S. Hu, 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 35 (8) (2016) 1269–1282.

[18] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, B. Gedik, COLA: Optimizing Stream Processing Applications via Graph Partitioning, in: J. M. Bacon, B. F. Cooper (Eds.), Middleware 2009, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 308–327.

[19] J. Yu, 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, 2006, pp. 1–10. doi:10.1109/WORKS.2006.5282330.

[20] K. Huang, S. Xiu, M. Yu, X. Zhang, R. Yan, X. Yan, Z. Liu, Software pipeline–based partitioning method with trade-off between workload balance and communication optimization, ETRI Journal 37 (3) (2015) 562–572. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.4218/etrij.15.0114.0502, doi:10.4218/etrij.15.0114.0502.

[21] M. Wieczorek, R. Prodan, T. Fahringer, Scheduling of Scientific Workflows in the ASKALON Grid Environment, SIGMOD Rec. 34 (3) (2005) 56–62. doi:10.1145/1084805.1084816.

[22] W. Kelly, M. Flasskamp, G. Sievers, J. Ax, J. Chen, C. Klarhorst, C. Ragg, T. Jungeblut, A. Sorensen, A communication model and partitioning algorithm for streaming applications for an embedded MPSoC, in: 2014 International Symposium on System-on-Chip (SoC), 2014, pp. 1–6. doi:10.1109/ISSOC.2014.6972436.

[23] V. Vargas, P. Ramos, J.-F. Méhaut, R. Velazco, NMR-MPar: A Fault-Tolerance Approach for Multi-Core and Many-Core Processors, Applied Sciences (2018) 8 (3). doi:10.3390/app8030465.
URL https://www.mdpi.com/2076-3417/8/3/465

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

[25] V. Cavé, R. Clédat, P. Griffin, A. More, B. Seshasayee, S. Borkar, S. Chatterjee, D. Dunning, J. Fryman, Traleika glacier: A hardware-software co-designed approach to exascale computing, Parallel Computing 64 (2017) 33 – 49. doi:https://doi.org/10.1016/j.parco.2017.02.003.

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

[27] G. Aupy, A. Benoit, Approximation algorithms for energy, reliability, and makespan optimization problems, Parallel Process. Lett. 26 (1) (2016) 1–23. doi:10.1142/S0129626416500018.

[28] H. Xu, R. Li, C. Pan, K. Li, Minimizing energy consumption with reliability goal on heterogeneous embedded systems, J. of Parallel and Distributed Computing 127 (2019) 44 – 57. doi:https://doi.org/10.1016/j.jpdc.2019.01.006.

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

[30] ARM Cortex-A15 Technical Reference Manual (September 2021) [cited 30 September 2021].
URL https://developer.arm.com/documentation/ddi0438/i/

[31] K. Nikov, J. L. Nunez-Yanez, M. Horsnell, Evaluation of Hybrid Run-Time Power Models for the ARM Big.LITTLE Architecture, in: 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing, 2015, pp. 205–210. doi:10.1109/EUC.2015.32.

[32] ARM Cortex-A15 wikipedia page (September 2021) [cited 30 September 2021].
URL https://en.wikipedia.org/wiki/ARM_Cortex-A15

[33] NXP layerscape-processors LX2160A (September 2021) [cited 30 September 2021].
URL https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-processors/layerscape-lx2160a-lx2120a-lx2080a-processors:LX2160A#collapse9

[34] T. Miyamori, H. Xu, T. Kodaka, H. Usui, T. Sano, J. Tanabe, Development of low power many-core soc for multimedia applications, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '13, EDA Consortium, San Jose, CA, USA, 2013, p. 773–777.