Graph Transformations for Memory Peak Minimization by Scheduling

Pascal Fradet, Alain Girault, Alexandre Honorat

Graph Transformations for Memory Peak Minimization by Scheduling

PASCAL FRADET, Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, France
ALAIN GIRAULT, Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, France
ALEXANDRE HONORAT, Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, France

Many computing systems are constrained by a fixed amount of available shared memory. Modeling applications with task graphs or Synchronous DataFlow (SDF) graphs makes it possible to analyze and optimize their memory usage. The NP-complete problem studied here is to find a sequential schedule of dataflow graphs that minimizes their memory peak. We propose four task graph transformations that reduce the graph size while preserving the minimal memory peak. They are able to compress all Series-Parallel Directed Acyclic Graphs (SP-DAG) to a single node representing an optimal schedule (optimal in the sense that its memory peak is minimal). They also offer simple criteria to schedule optimally independent tasks and guided us to develop an optimal compositional scheduling analysis.

These transformations are quite effective and compress a large class of graphs into a single node representing an optimal schedule. When this is not the case, we propose an optimized branch and bound algorithm to complete the analysis of the reduced graphs. This algorithm is able to find the optimal schedule of graphs comprising up to 100 tasks. Our approach also applies to SDF graphs after converting them to task graphs. However, since this conversion may produce huge graphs, we also describe a new suboptimal method, similar to Partial Expansion Graphs, to reduce the graph and schedule sizes. We evaluate our approach on classic benchmarks, on which we always outperform the state-of-the-art.

CCS Concepts: • **Theory of computation** → **Streaming models**.

Additional Key Words and Phrases: dataflow, task graph, SDF, memory peak, sequential scheduling

ACM Reference Format:
Pascal Fradet, Alain Girault, and Alexandre Honorat. 2024. Graph Transformations for Memory Peak Minimization by Scheduling. *ACM Trans. Embedd. Comput. Syst.* 1, 1 (October 2024), 36 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Memory footprint is an important constraint to consider when developing software applications. In the domain of artificial intelligence, some neural networks (*e.g.,* GPT-3 [6]) cannot entirely fit in the memory of a single GPU. The modeling of neural networks with dataflow graphs of tasks [16] allows the memory requirements to be analyzed, and a schedule optimizing the memory to be produced [2, 3]. Similarly, embedded systems are subject to strict memory constraints (specially IoTs), and, here again, dataflow graphs of tasks help to efficiently schedule the tasks w.r.t. their consumed and produced amounts of data [4].

Authors' addresses: Pascal Fradet, Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, Grenoble, France, 38000, pascal.fradet@inria.fr; Alain Girault, Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, Grenoble, France, 38000, alain.girault@inria.fr; Alexandre Honorat, Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, Grenoble, France, 38000, alexandre.honorat@inria.fr.

ACM 1539-9087/2024/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The problem we address is to find a *sequential schedule* that minimizes the *memory peak* of a dataflow graph. This is directly useful for single processor applications as found frequently in the embedded context, but also for massively parallel applications where a GPU executes, using the Single Program Multiple Data model, the same sequence of tasks.

We take as input a dataflow graph with memory costs attached to edges. We consider two variants of dataflow graphs: *task graphs* and the more expressive *Synchronous DataFlow (SDF)* graphs [14]. In a task graph, each node is a task, *i.e.*, a piece of code executed atomically, and each edge between two nodes is a FIFO buffer. An SDF graph refines the task graph model by attaching two rates to each edge: the amount of data produced by its source node and the amount of data consumed by its destination node. An SDF graph can be transformed into a task graph, but this transformation potentially entails an exponential blow-up in the number of nodes and edges in the resulting task graph. In both cases, our goal is to find a static and sequential schedule of the dataflow graph whose memory peak is the minimal one among all possible schedules. The memory peak is the maximum memory needed to execute the dataflow graph according to a specific schedule. The memory peak minimization problem of task graphs is a variation of the pebble game [20], which is a NP-complete problem. A naive method to solve it is to generate all the linear extensions of the graph, but it has at least factorial complexity [17].

The heart of our contribution is a polynomial-time pre-processing of the task graph where we apply several *graph transformations* that compress the graph by fusing nodes such that: (i) the resulting graph's set of schedules is a subset of the initial graph's set of schedules, and (ii) the resulting graph's minimal memory peak is the same as the initial graph's. After this pre-processing, either the graph is reduced to a single node representing an optimal schedule, or we apply an optimized branch and bound (B&B) to find one on the reduced and usually much smaller graph.

Our main contributions consist of:

(1) local task graph transformations that compress the given task graph while preserving a schedule with the optimal memory peak;
(2) a simple technique to optimally schedule independent parallel tasks;
(3) a proof that our transformations always compress Series-Parallel Directed Acyclic Graphs (SP-DAGs) to a single node representing their optimal schedule;
(4) an optimal compositional analysis dealing with single-source single-sink subgraphs;
(5) an optimized B&B algorithm able to find optimal schedules for medium-to-large sized (50-100 nodes) task graphs;
(6) a suboptimal algorithm to reduce the size of the conversion of SDF graphs into task graphs;
(7) a complete implementation and experimental results that show that our transformations and algorithms outperform the state of the art.

The article is organized as follows. Sec. 2 defines the two considered models of computation: task graphs and SDF graphs. Sec. 3 introduces *schedule graphs*, an expressive representation of task graphs allowing us to reason on schedules and their memory usage. Sec. 4 presents four transformation rules compressing schedule graphs while preserving the existence of an optimal schedule (optimal w.r.t. the memory peak). In Sec. 5 we describe several properties and applications of our compression rules. In particular, we show that they are sufficient to compress any SP-DAG into a single node representing (one of) its optimal schedule. We present in Sec. 6 a compositional technique allowing to schedule separately some subgraphs while preserving optimality of the global schedule. Sec. 7 presents an optimized B&B algorithm to find optimal schedules of relatively large task graphs. Sec. 8 describes techniques to deal with the more general model of SDF graphs. We provide benchmark comparisons and other experimental results in Sec. 9. Finally, Sec. 10 presents related work and Sec. 11 concludes.

This article extends and revises the work presented in [10]. In particular, Secs. 5.1 to 5.3, 6, 7.1, 7.3, 8.1, 9.3 and 9.4 are entirely novel. Other sections have been extended (Secs. 3 and 10) or modified. A new appendix gathers the proofs of all properties omitted from the main text.

2 TASK GRAPHS AND SDF GRAPHS

Our goal is to find a static sequential schedule with the optimal (lowest) memory peak for task or SDF graphs. We present the characteristics of these two models in turn.

2.1 Task Graphs

The task graph model we consider consists of a Directed Acyclic Graph (DAG) where vertices represent tasks and edges represent FIFO communication buffers between tasks. The atomic execution of a task, referred to as a *firing*, consumes data from all its incoming edges (its inputs) and produces data to all its outgoing edges (its outputs). The data unit is called a *token*, and the same unit is used for all measures (production, consumption, memory peak). The number of tokens consumed (resp. produced) on a given edge at each firing is indicated on the edge and is called the consumption (resp. production) *rate*. A task can fire only when all its input edges contain enough tokens. It then reads them, execute its code and write its result as tokens to its outgoing edges according to their rates.

Fig. 1 presents a simple task graph with five tasks A, B, C, D, and E. When fired, task B consumes 1 token on its input edge and produces 2 and 4 tokens on its two output edges, which will be eventually consumed by C and D respectively when they will fire. Task A does not have any ingoing edges: it is a pure producer called a *source*. Task D is a pure consumer and is called a *sink*. There are three possible schedules for this task graph: $(A;B;C;E;D)$, $(A;E;B;C;D)$, and $(A;B;E;C;D)$. In general, for a connected graph of n tasks, there can be up to $(n-1)!$ schedules.

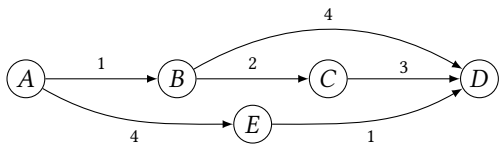


Fig. 1. A simple task graph example.

The memory occupied during execution by a task graph is the sum of all the tokens present on all its edges (*i.e.*, buffers). We assume that the FIFO buffers are allocated in the same global shared memory. Compared to implementations where each buffer is allocated independently in memory, the model we consider, called the *shared buffer model* [18], minimizes the memory requirement. The memory peak of a schedule is the maximum memory needed at any execution step of that schedule.

There exist two classic local memory models for tasks:

- the consumed-before-produced (CBP) model where a task first reads and consumes its input tokens, then executes its code, and finally produces its result as output tokens;
- the produced-before-consumed (PBC) model where a task reads and keeps its input tokens, executes its code and produces its result before consuming (*i.e.*, freeing) its input.

The memory peak of a schedule depends on the chosen task memory model. For the task graph of Fig. 1, the minimal memory peak is 8 assuming the CBP model and 10 with the PBC model. Both are obtained with the schedule $A;E;B;C;D$ at the firing of task C. Our approach can take those two local task models (and others) into account.

2.2 SDF Graphs

The SDF [14] dataflow model generalizes task graphs by allowing different consumption and production rates on a given edge. SDF nodes are called *actors* and have the same conditions for firing as tasks in task graphs.

Heterogeneous rates imply that each actor must be fired multiple times to balance the production and consumption of tokens on each edge. For instance, in the SDF graph $A \xrightarrow{2} \xrightarrow{1} B$, each firing of A produces 2 tokens and each firing of B consumes 1 token; hence, each firing of A must be eventually followed by two firings of B.

More generally, each edge $A \xrightarrow{r} \xrightarrow{s} B$ gives rise to the following *balance equation*, with $\#X$ denoting the number of firings of actor X:

$$\#A \cdot r = \#B \cdot s \tag{1}$$

An SDF graph is said to be *consistent* if its system of balance equations has a non null solution. In other words, there exists a set of firings, called an *iteration*, which consumes exactly all tokens produced on each edge.

For the basic SDF graph $A \xrightarrow{2} \xrightarrow{1} B$, the minimal solution of the balance equation is $\#A = 1$ and $\#B = 2$ and the minimal iteration is $\{A, B, B\}$ (see Fig. 13 in Sec. 8 for another SDF example). Consistent SDF graphs can be executed indefinitely with bounded memory and, as in most work, we consider only consistent SDF graphs.

Finally, consistent SDF graphs can be converted into task graphs (also called Single-Rate SDF (SRSDF) or Homogeneous SDF (HSDF) [4]) by duplicating each actor into n tasks, n being its number of firings in the minimal iteration. This conversion produces a number of tasks equals to the length of the iteration which can be, in pathological cases, exponential in terms of number of actors.

3 SCHEDULE GRAPHS

We first define more precisely schedules and the key notions of *memory peak* and *impact*. We then introduce *schedule graphs*, a representation of task graphs that serves as the foundation for our transformations and analyses.

For a given task graph G, a sequential schedule is either a single task A (a node of G) or the sequence of two schedules as formalized by the following grammar:

$$S ::= A \mid S_1; S_2$$

For our purposes, a schedule S has two key attributes:

- its *peak*, a natural number denoting the maximal memory peak reached during its execution;
- its *impact*, an integer denoting the final number of tokens added or removed after its execution.

A schedule's peak and impact are expressed independently of the state of the memory before its execution. These attributes are either denoted directly on the schedule as $S^{\binom{\pi_S}{\iota_S}}$ or referred to as π_S and ι_S. For any schedule $S^{\binom{\pi_S}{\iota_S}}$, we have $\iota_S \leq \pi_S$ with $\pi_S \in \mathbb{N}$ and $\iota_S \in \mathbb{Z}$.

Consider an execution where p is the current memory peak already reached and n is the current number of tokens in memory. After the execution of $S^{\binom{\pi_S}{\iota_S}}$, the new peak will be $\max(p, n + \pi_S)$ and the new number of tokens will be $n + \iota_S$.

A task A that consumes s tokens and produces r tokens is represented by:

$A^{\binom{\max(0,r-s)}{r-s}}$	in the CBP model
$A^{\binom{r}{r-s}}$	in the PBC model

Both models entail the same impact, but the peak of a task in the PBC model is always the number of tokens it produces whereas, in the CBP model, the number of consumed tokens is first subtracted and the peak might be null.

Schedule graphs are a quite expressive representation. For instance, applications where some tasks require more memory to perform their internal computation can be expressed by adjusting their peaks accordingly.

In the sequel, we represent a task graph as a *schedule graph* where each node represents a schedule (a single task being a schedule of length 1) decorated with its peak and impact, as in Fig. 2.

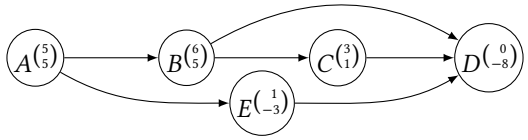


Fig. 2. The schedule graph for the task graph of Fig. 1 in the PBC model.

The memory peak of a complete schedule can be computed using the following associative operation:

$$A^{\binom{\pi_a}{\iota_a}}; B^{\binom{\pi_b}{\iota_b}} = (A;B)^{\binom{max(\pi_a, \pi_b+\iota_a)}{\iota_a+\iota_b}} \tag{2}$$

The impact of the sequential execution of nodes A and B is the sum of their impacts, whereas its memory peak is the maximum of the peak reached during A and the peak reached during B taking into account the impact of A.

For example, the peak of the schedule $A;E;B;C;D$ of the graph of Fig. 2 can be computed as:

$$\begin{aligned} A^{\binom{5}{5}}; E^{\binom{1}{-3}}; B^{\binom{6}{5}}; C^{\binom{3}{1}}; D^{\binom{0}{-8}} &= (A;E)^{\binom{6}{2}}; (B;C)^{\binom{8}{6}}; D^{\binom{0}{-8}} \\ &= (A;E;B;C)^{\binom{10}{8}}; D^{\binom{0}{-8}} \\ &= (A;E;B;C;D)^{\binom{10}{0}} \end{aligned}$$

Each intermediate step in the above computation can be represented as a schedule graph: *e.g.*, the *rhs* of the first equation can be seen as a schedule graph where the schedule node $(A;E)^{\binom{6}{2}}$ has two outgoing edges towards $(B;C)^{\binom{8}{6}}$ and $D^{\binom{0}{-8}}$.

A task graph, or a consistent SDF graph, consumes exactly the number of tokens it produces: its global schedule starts with a null memory peak and impact, and completes with an arbitrary peak but always with a null impact. In the above schedule, the final impact is null and the global peak (10) is reached while executing task C.

In the remainder of this article, we use additional concepts related to schedules. These are illustrated in Fig. 3 whose peaks and valleys depicts memory usage during execution of the schedule $S = X_1;X_2;X_3;X_4$.

- *Valleys.* The valley of X_i in S, denoted by $\Upsilon^S_{X_i}$, is the memory usage *after* the execution of X_i. It is computed as the sum of previous impacts: $\Upsilon^S_{X_i} = \iota_{X_1} + \ldots + \iota_{X_i}$.
- *Relative peaks.* The relative peak of X_i in S is the memory peak reached *during* the execution of X_i. It is computed as $\pi_{X_i} + \Upsilon^S_{X_{i-1}}$. The highest relative peak of a schedule is its global memory peak.
- *Drops.* The drop of X_i in S is the local decrease of memory between its peak and impact. It is computed as $\pi_{X_i} - \iota_{X_i}$.

The *mirror image* of a schedule is the schedule of the reversed graph where drops and peaks are exchanged, while positive impacts become negative and vice versa. This symmetry is used in several instances throughout our work.

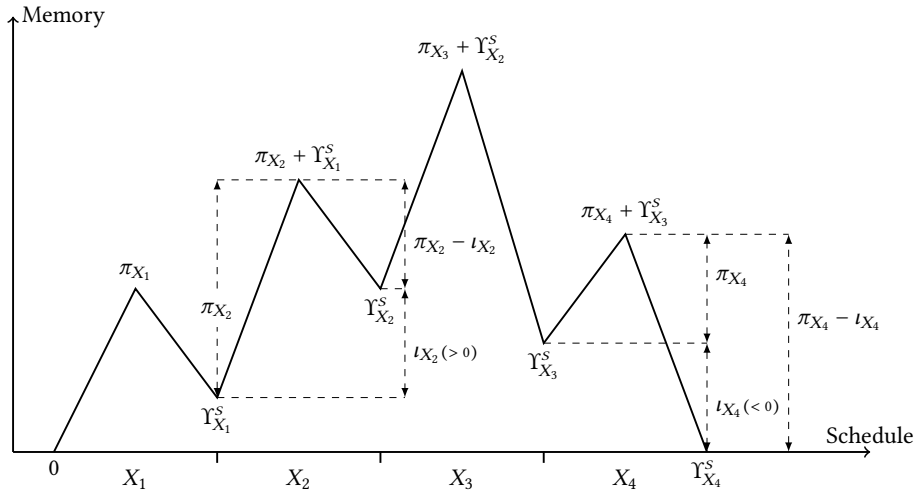


Fig. 3. Numerical features of a schedule $S = X_1; X_2; X_3; X_4$.

We make also use of the following order relation:

$$S_1 \binom{\pi_{S_1}}{\iota_{S_1}} \preceq_p S_2 \binom{\pi_{S_2}}{\iota_{S_2}} \Leftrightarrow \pi_{S_1} \leq \pi_{S_2} \tag{3}$$

Our goal is to find minimal schedules according to this $\preceq_p$ order.

4 PEAK PRESERVING TRANSFORMATION RULES

We present transformation rules that simplify the schedule graph (in number of nodes and edges) while preserving its minimal memory peak. We *reduce*, *cluster*, and *sequentialize* the graph but ensure that there still exists a schedule with the same minimal memory peak as the original graph.

Combined and applied repeatedly, these local transformations compress substantially most task graphs. They even compress many graphs (and, among them, all SP-DAGs) to a single node representing an optimal schedule. In general, a compressed graph may remain large but it has usually much less possible schedules than the original graph.

We first present the *reduction* transformation in Sec. 4.1, followed by *clustering* Sec. 4.2, and *sequentialization* in Sec. 4.3. The global compression algorithm is presented in Sec. 4.4.

In the following, for X a node of the schedule graph S, we write $\text{Succ}(X)$ for its set of immediate successors, $\text{Pred}(X)$ for its set of immediate predecessors, $\text{Succ}^+(X)$ and $\text{Pred}^+(X)$ for the sets of all its successors and predecessors respectively in the transitive closure of S.

4.1 Transitive Reduction

The first graph transformation suppresses useless edges. It does not directly suppress possible schedules but makes the subsequent transformations more effective.

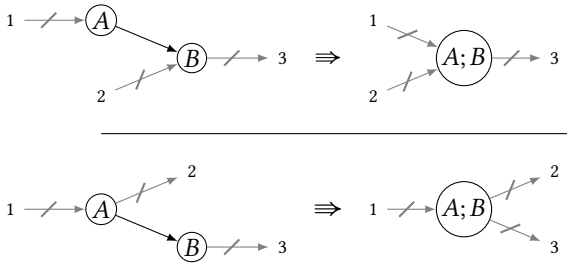
In our schedule graphs, edges do not carry other information than dependencies. If removing an edge does not modify the impact nor the peak of the nodes that it connects it might however modify the scheduling constraints. Consider for instance the nodes B, C, D in Fig. 2: The edge $B \to D$ can be removed because it does not add any scheduling constraint. Those three nodes must be executed in the order $B; C; D$ (possibly interleaved with some sub-schedules), the edge $B \to D$ being present or not.

This transformation is the classic transitive reduction [1]. It returns the graph with the fewest edges having the same reachability relation.

4.2 Node Clustering

Let A and B be two connected nodes such that $\text{Succ}(A) = \{B\}$ and $\text{Pred}(B) = \{A\}$. The $A \to B$ dependency implies that any sequential schedule of the graph is of the form $\ldots; A; S; B; \ldots$ with S a schedule node made of nodes that do not depend on A nor B. Therefore, S could also be executed before A or after B. If we can prove that, for any S, the memory peak of $A; S; B$ is greater than or equal to the peak of $S; A; B$ or $A; B; S$, then there is no gain in interleaving S between A and B. In that case, A and B can be clustered in a single node denoted by $[A; B]$. The resulting graph has much less schedules since we got rid of all the schedules that interleaved tasks between A and B as they could not lead to a strictly better peak.

This simple case is generalized by the two transformations Rules (C_1) and (C_2) of Fig. 4, which cluster two nodes A and B according to topological and arithmetic conditions.



$$\text{Succ}(A) = \{B\} \wedge \big(0 \le \iota_A\big) \wedge \big(\pi_A - \iota_A \le \pi_B\big) \qquad (C_1)$$

$$\text{Pred}(B) = \{A\} \wedge \big(\iota_B \le 0\big) \wedge \big(\pi_B \le \pi_A - \iota_A\big) \qquad (C_2)$$

Fig. 4. Clustering rules for single successor (top) and single predecessor (bottom). Striked through arrows ⇝ represent 0 or more incoming/outgoing edges.

• Rule (C_1). The topological condition is $\text{Succ}(A) = \{B\}$, that is, B is the unique successor of A but B may have several predecessors. Here, all nodes that can be executed between A and B can also be executed before A. In general, they cannot all be executed after B since B has predecessors that must be executed before. The arithmetic conditions are $0 \le \iota_A \wedge \pi_A - \iota_A \le \pi_B$, that is, the impact of A is positive and the drop of A is less than or equal to the peak of B.

If both topological and arithmetic conditions are met, the two nodes can be clustered into a single node. The correctness of this transformation is ensured by the following property.

Property 1 (Rule (C_1)). *Let A and B be two nodes of a schedule graph.*

$$\forall S,\ 0 \le \iota_A \wedge \pi_A - \iota_A \le \pi_B \ \Leftrightarrow\ (S; A; B) \preceq_p (A; S; B)$$

Proof.

($\Rightarrow$) $0 \le \iota_A \wedge \pi_A \le \pi_B + \iota_A$ is a sufficient condition. Indeed,

$$\begin{cases} 0 \le \iota_A & \Rightarrow \quad \pi_S \le \pi_S + \iota_A \\ \pi_A - \iota_A \le \pi_B & \Rightarrow \quad \pi_A + \iota_S \le \pi_B + \iota_A + \iota_S \end{cases}$$

therefore

$$\max\{\pi_S, \pi_A + \iota_S, \pi_B + \iota_S + \iota_A\} \le \max\{\pi_A, \pi_S + \iota_A, \pi_B + \iota_A + \iota_S\}$$

which is the expanded form of $(S; A; B) \preceq_p (A; S; B)$ according to Eqs. (2) and (3).

($\Leftarrow$) We show that if the condition does not hold, *i.e.*,

$$\iota_A < 0 \ \vee\ \pi_B < \pi_A - \iota_A$$

then there exists a schedule node S whose execution between A and B yields a smaller peak, *i.e.*,

$$(A; S; B) \prec_p (S; A; B) \ \textit{ i.e., } \max\{\pi_A, \pi_S + \iota_A, \pi_B + \iota_A + \iota_S\} < \max\{\pi_S, \pi_A + \iota_S, \pi_B + \iota_S + \iota_A\}$$

Let S be a node of the schedule graph s.t. $(\pi_S = \pi_A - \iota_A) \wedge (\iota_S = \pi_S - \pi_B)$. Since the impact of a node is always less or equal than its positive peak, π_S is a valid possible peak (*i.e.*, $0 \leq \pi_S$). It follows that $\pi_A = \max\{\pi_A, \pi_S + \iota_A, \pi_B + \iota_A + \iota_S\}$. If the condition does not hold, either $\iota_A < 0$ and we have $\pi_A < \pi_S = \pi_A + \iota_A$, or $\pi_B + \iota_A < \pi_A \Rightarrow \pi_A < \pi_A + \iota_S$. In both cases it implies $\pi_A < \max\{\pi_S, \pi_A + \iota_S, \pi_B + \iota_S + \iota_A\}$ and so $(A;S;B) \prec_p (S;A;B)$. □

• Rule (C_2). The topological condition is $\mathrm{Pred}(B) = \{A\}$. That is, A is the unique predecessor of B, but A may have several successors. This case is the dual of the previous one: all nodes that can be executed between A and B can also be executed after B. In general they cannot all be executed before A since A has other successors that must be executed after. The arithmetic conditions are $\iota_B \leq 0 \wedge \pi_B \leq \pi_A - \iota_A$, that is, the impact of B is negative and the peak of B is less than or equal to the drop of A.

If these topological and arithmetic conditions are met, the two nodes can be clustered into a single node. The correctness of this transformation is ensured by the following property.

Property 2 (Rule (C_2)). *Let A and B be two nodes of a schedule graph.*

$$\forall S,\ \iota_B \leq 0 \ \wedge\ \pi_B \leq \pi_A - \iota_A \ \Leftrightarrow\ (A;B;S) \preceq_p (A;S;B)$$

The proof is similar to the proof of Prop. 1. As other proofs omitted from the main text, it can be found in the appendix.

The symmetry pointed out in the previous section is evidenced here: Rule (C_2) is the mirror image of Rule (C_1). Indeed, by reversing the edges of the graph of Rule (C_1) we get the graph of Rule (C_2) and its topological conditions (with A and B playing each other role). By swapping peaks and drops and replacing positive impact by negative impact we get the arithmetic conditions of Rule (C_2).

We show in Sec. 5.1 that clustering is an associative operation.

4.3 Node Sequentialization

Another useful transformation is the sequentialization of nodes that could be executed in any order in the original graph. The benefits of sequentialization are twofold: it eliminates useless schedules and it creates new clustering opportunities for Rules (C_1) and (C_2). Of course, this comes with conditions to ensure that sequentialization cannot suppress a schedule that minimizes the memory peak.

The sequentialization of two nodes A and B is performed by the two Rules (S_1) and (S_2) of Fig. 5.

• Rule (S_1). The topological condition is $\mathrm{Pred}(A) \subseteq \mathrm{Pred}^+(B)$. That is, the immediate predessors of A are all predecessors (not necessarily immediate) of B. This condition ensures that every schedule node that can be executed after B and before A can also be executed *after* A and B. The arithmetic conditions are $\iota_A \leq 0 \wedge \pi_A \leq \pi_B$, that is, the impact of A is negative and the peak of A is less than or equal to the peak of B.

If these topological and arithmetic conditions are met, the two nodes can be sequentialized by adding a new edge from A to B. The correctness of this transformation is ensured by the following property.

Property 3 (Rule (S_1)). *Let A and B be two nodes of a schedule graph.*

$$\forall S,\ \iota_A \leq 0 \ \wedge\ \pi_A \leq \pi_B \ \Leftrightarrow\ (A;B;S) \preceq_p (B;S;A)$$

This entails that all schedules of the form $\ldots;B;S;A;\ldots$ cannot lead to a strictly smaller peak than schedules of the form $\ldots;A;B;S;\ldots$, and therefore we can execute A before B. This is not a

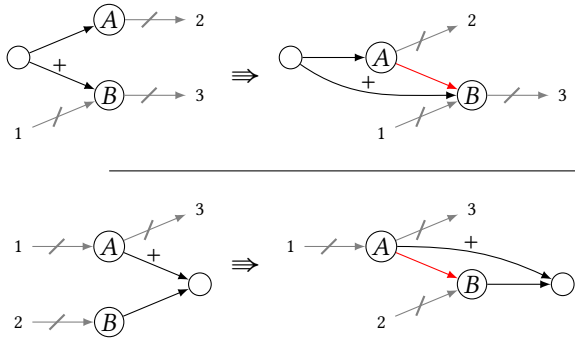


Fig. 5. Sequentialization rules for same predecessors (top) and same successors (bottom). The red arrow from A to B is added by the transformation. Striked through arrows $\not\rightarrow$ represent 0 or more incoming/outgoing edges.

sufficient condition to cluster A and B because interleaving some schedule between them might be beneficial w.r.t. the memory peak.

• Rule (S_2). The topological condition is $\text{Succ}(B) \subseteq \text{Succ}^+(A)$. That is, the immediate successors of B are all successors (not necessarily immediate) of A. This condition ensures that every schedule node that can be executed after B and before A can also be executed *before* A and B. The arithmetic conditions are $0 \leq \iota_B \wedge \pi_B - \iota_B \leq \pi_A - \iota_A$, that is, the impact of B is positive and the drop of B is less than or equal to the drop of A.

If these topological and arithmetic conditions are met, the two nodes can be sequentialized by adding a new edge from A to B. The correctness of this transformation is ensured by the following property.

Property 4 (Rule (S_2)). *Let A and B be two nodes of a schedule graph.*

$$\forall S,\ 0 \leq \iota_B \ \wedge\ \pi_B - \iota_B \leq \pi_A - \iota_A \ \Leftrightarrow\ (S;A;B) \preceq_p (B;S;A)$$

This entails that all schedules of the form $\ldots;B;S;A;\ldots$ cannot lead to a strictly smaller peak than schedules of the form $\ldots;S;A;B;\ldots$, and therefore we can execute A before B.

As for clustering, Rule (S_2) is the mirror image of Rule (S_1). Indeed, by reversing the edges of the graph in Rule (S_1) we get the graph of Rule (S_2) and its topological conditions (with A and B playing each other rule). By replacing in Rule (S_1) peaks by drops and negative impacts by positive impacts we get the arithmetic conditions of Rule (S_2).

4.4 The Compression Algorithm

Our compression algorithm, sketched in Alg. 1, makes use of clustering, sequentialization, and transitive reduction to compress schedule graphs as much as possible. It involves three nested loops:

1. clustering(G) traverses the whole graph G and tries to apply the two clustering Rules (C_1) and (C_2) (linear-time complexity). When a clustering creates a transitive edge, a local reduction is applied. It also computes the set of neighbours of the clustered nodes which will be candidates to further clustering. The subsequent steps of clustering are done on this set which may evolve. This phase ends when no clustering rule applies. As shown in Sec. 5.1, clustering is associative and the order in which rules are applied is insignificant.

Algorithm 1: Global compression algorithm (sketch)

```
/* Takes a schedule graph G and compresses it until none of the transformations apply */
changed := false;
repeat
    repeat
        repeat
            clustering(G); ▷ O(n)
        until ¬ changed;
        basic_sequentialization(G); ▷ O(n^2)
    until ¬ changed;
    complete_sequentialization(G); ▷ O(n^3)
    transitive_reduction(G); ▷ O(n^3)
until ¬ changed;
```

2. basic_sequentialization(G) is a simplified version of sequentialization where the topological conditions for the two rules respectively become $\text{Pred}(A) = \text{Pred}(B)$ and $\text{Succ}(B) = \text{Succ}(A)$. These are the most common cases and they are less costly to check than the general ones. If this step modifies the graph, a new round of clustering is performed.

3. complete_sequentialization(G) checks the most general conditions of Rules (S_1) and (S_2) and applies the corresponding transformations. It is followed by a step of transitive reduction, provided by a standard library. These two procedures have a cubic worst case time complexity, which may be problematic for large graphs. If these two steps have changed the graph, a new round of clustering plus basic_sequentialization is performed.

Before sequentializing two nodes A and B we always check if a path from A to B already exists. In that case, sequentialization is not performed as the added link between A and B is useless and could even make the algorithm loop between sequentialization and transitive reduction.

The algorithm stops when no further clustering, sequentialization, or reduction are possible. Its global worst case time complexity is quartic in the number of nodes.

5 PROPERTIES AND APPLICATIONS OF TRANSFORMATION RULES

The compression algorithm applies to any DAG and usually reduces significantly its size. As we will see in Sec. 9, it often compresses graphs into a single node representing an optimal schedule. In this section, we review additional properties and applications of our four transformation rules.

5.1 Associativity of Clustering

We write $R_i(A, B)$ if the two schedule nodes A and B satisfy the topological and arithmetic conditions of the R_i rule *i.e.*, (C_1), (C_2), (S_1), (S_2).

Sometimes, the same node can be part of two different possible clusterings. It is then natural to consider the order of application of these rules. We show that clustering enjoy associativity-like properties.

Property 5. *Let A, B and C three nodes of a schedule graph then*

- $C_1(A,B) \wedge C_1(B,C) \Rightarrow C_1([A;B],C) \wedge C_1(A,[B;C])$
- $C_2(A,B) \wedge C_2(B,C) \Rightarrow C_2([A;B],C) \wedge C_2(A,[B;C])$
- $C_1(A,B) \wedge C_2(B,C) \Rightarrow C_2([A;B],C) \wedge C_1(A,[B;C])$

If Rule (C_1) can be applied on the one hand to A and B, and on the other hand to B and C, then no matter the first two nodes clustered, we get a graph that can be clustered into the single $[A;B;C]$ schedule node. The same property holds when Rule (C_2) can be applied to A and B and to B and C, and when Rule (C_1) can be applied to A and B and Rule (C_2) to B and C. However, the case where $C_2(A,B)$ and $C_1(B,C)$ is more subtle: the topological and arithmetic conditions of Rules (C_1) and (C_2) on $([A;B];C)$ or $(A;[B;C])$ do not hold in general. However, this case enjoys some form of associativity as well.

Property 6. *Let A, B, and C be three nodes of a schedule graph, if $C_2(A,B)$ and $C_1(B,C)$, then the two possible clusterings, yield two isomorphic graphs*

$$X^{\binom{\pi_A}{\iota_A}}; Y^{\binom{\pi_C}{\iota_C}} \text{ with either } X = [A;B] \text{ and } Y = C \text{ or } X = A \text{ and } Y = [B;C]$$

Together, Props. 5 and 6 allow clustering rules to be applied in any order as long as possible. If sequentialization permits further clustering by putting new nodes in sequence, it also introduces new edges that may prevent some clustering that was possible before. Therefore, as done in Alg. 1, it is better to perform clustering as much as possible before sequentialization and to repeat this series of actions iteratively.

5.2 Compression of Linear Chains

We consider in this section *linear chains*, where each node has a single predecessor and successor, except the first (resp. last) node which has a single successor (resp. predecessor). In particular, a sequential schedule is naturally represented as a linear chain. When clustered, linear chains enjoy important properties used in Secs. 5.4 and 6 and in the corresponding proofs.

Property 7 (Sorted Chains). *By applying Rules (C_1) and (C_2) as long as possible, any linear chain is compressed into a chain of the form:*

$$N_1 \to \ldots \to N_n \to P_1 \to \ldots \to P_m \text{ with } 0 \leq n, m$$

where the N_i are nodes with a negative impact and the P_i nodes with a positive impact. These chains are referred to as maximally clustered chains or sorted chains.

Proof. In the rest of this article, we refer to nodes with a negative and positive impact as *negative* and *positive nodes*. Any dependency $P \to N$ with P a positive node and N a negative one, will be clustered into the single node $[P;N]$ because either the conditions of Rule (C_1) or Rule (C_2) are satisfied. Therefore, by applying repetitively the clustering rules, no positive node can be followed by a negative node. □

An important corollary is that dataflow applications represented by a linear chain can always be clustered into a single node. Indeed, in a dataflow application, a task can only consume tokens that have been produced before: the size of the memory during execution cannot be negative (all valleys are positive). Therefore, the impact of any prefix of a linear chain representing a complete dataflow application is positive. Such a chain always starts with a positive node (a producer) and if it is followed by a negative one, their combined impact remains positive.

Corollary 8. *Any linear chain whose prefixes have all a positive impact can be compressed using Rules (C_1) and (C_2) into a single node.*

Sorted chains share a very specific form as illustrated in Fig. 6. The negative (resp. positive) nodes have a *strictly* negative (resp. positive) impact. On the negative part, the relative peaks are strictly increasing whereas the valleys are strictly decreasing. On the positive part, the situation is symmetric. A node with a null impact can possibly occur but it is unique and located between the

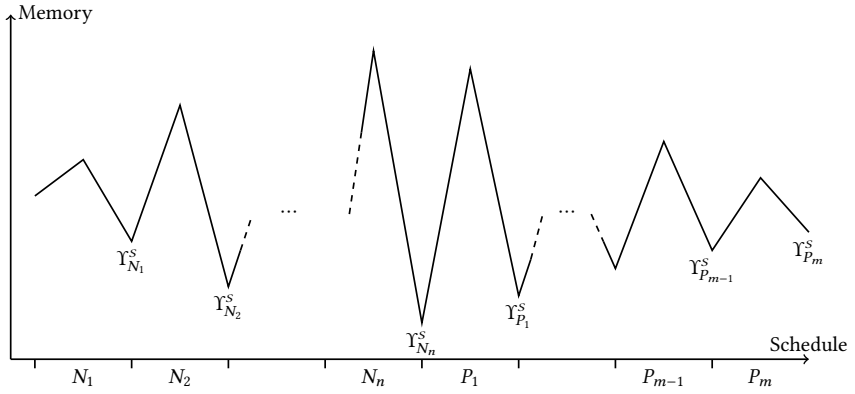


Fig. 6. A sorted chain.

negative and positive parts of the chain. The highest peak and lowest valley are also located in the middle of the chain.

Property 9. *A sorted chain S can be of two forms*

$$N_1 \to \ldots \to N_n \to P_1 \to \ldots \to P_m \text{ or } N_1 \to \ldots \to N_n \to Z \to P_1 \to \ldots \to P_m$$

$$\text{with} \quad \iota_Z = 0 \quad \begin{array}{lll} \iota_{N_i} < 0 & \wedge & \pi_{N_i} < \pi_{N_{i+1}} + \iota_{N_i} \quad i = 1,..,n-1 \\ 0 < \iota_{P_j} & \wedge & \pi_{P_{j+1}} + \iota_{P_j} < \pi_{P_j} \quad j = 1,..,m-1 \end{array}$$

This property derives from the following remarks:

- A null node occurring before a negative one or after a positive node can always be clustered since at least Rule (C_1) or Rule (C_2) applies. It follows that if a null node exists, then it is unique, located in the middle after the last negative node and before the first positive one. It also entails that all other negative and positive nodes have a non null impact.
- Two consecutive negative nodes N_i and N_{i+1} such that $\pi_{N_{i+1}} + \iota_{N_i} \leq \pi_{N_i}$ can be clustered using Rule (C_2). It follows that the relative peaks of $N_1 \to \ldots \to N_n$ are strictly increasing. Valleys on this part are decreasing since they are defined as the sum of the previous valley and a negative impact.
- Two consecutive positive nodes P_i and P_{i+1} such that $\pi_{P_i} \leq \pi_{P_{i+1}} + \iota_{P_i}$ can be clustered using Rule (C_1). It follows that the relative peaks of $P_1 \to \ldots \to P_m$ are strictly decreasing. Valleys on this part are increasing since they are defined as the sum of the previous valley and a positive impact.

These properties imply that the largest peak of the schedule $S^{\binom{\pi_s}{\iota_s}}$ is reached either by N_n or by P_1 (or both) and is defined as

$$\pi_s = \max(\pi_{N_n} + \Upsilon^S_{N_{n-1}}, \pi_{P_1} + \Upsilon^S_{N_n})$$

The lowest valley of S which is reached after N_n and before P_1 (and occurs twice if there is a null node) is equal to

$$\Upsilon^S_{N_n} = \iota_{N_1} + \ldots + \iota_{N_n}$$

Again such chains enjoy the same symmetry as before: the positive part is the mirror image of a negative chain. The mirror image of $P_1 \to \ldots \to P_m$ is a negative chain $P_m^{-1} \to \ldots \to P_1^{-1}$ such that if the peak and impact of P_k are π_k and ι_k then the peak and impact of P_k^{-1} are $\pi_k - \iota_k$ and $-\iota_k$.

5.3 Optimal Scheduling of Independent Tasks

Consider a set of tasks which may consume and produce tokens independently of each other. Sequentialization rules provide an easy characterization of an optimal sequential schedule for such sets. It is sufficient to execute the negative tasks (*i.e.*, with a negative impact) in an increasing peak order before executing the positive nodes in a decreasing drop order.

Property 10. *The schedule* $N_1;\ldots;N_n;P_1;\ldots;P_m$ *minimizes the memory peak of any set* $\mathcal{X} = \{N_1,\ldots,N_n,P_1,\ldots,P_m\}$ *of independent tasks with*

$$\iota_{N_i} \leq 0 \ \wedge\ \pi_{N_i} \leq \pi_{N_{i+1}} \text{ for } i = 1..n \quad \wedge \quad 0 \leq \iota_{P_i} \ \wedge\ \pi_{P_{i+1}} - \iota_{P_{i+1}} \leq \pi_{P_i} - \iota_{P_i} \text{ for } i = 1..m$$

We can show that this schedule minimize the memory peak using sequentialization. Consider the graph made of all the tasks of $\mathcal{X}$ in parallel with two dummy source and sink nodes S and T (see Fig. 7).

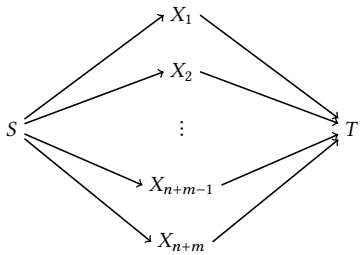


Fig. 7. A graph with $n + m$ independent tasks in parallel.

Applying the sequentialization Rules (S_1) and (S_2) transforms the graph into the above schedule (surrounded by S and T). For instance, sequentializing a negative node into an already sequentialized chain $N_1 \to \ldots \to N_j \to P_1 \to \ldots \to P_k$ inserts it before the first node N_i such that $\pi_X < \pi_{N_i}$. The property (increasing peaks then decreasing drops) is preserved. This is shown by induction on the successive sequentializations of the different tasks (see appendix C).

The properties of the sequentialization rules ensure that the schedule specified by Prop. 10 has the optimal memory peak.

5.4 Series-Parallel Graphs

SP-DAGs (see Fig. 8 for a simple example) are a well-known and important class of DAGs for which the compression algorithm is particularly effective. For this class, it can be shown that the four Rules (C_1), (C_2), (S_1), and (S_2) compress any SP-DAG to a single node. This single node represents (one of) its optimal schedule.

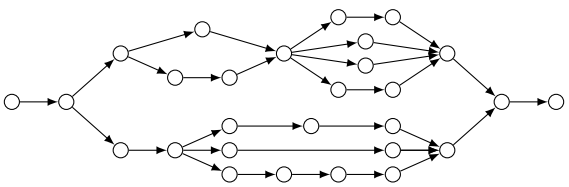


Fig. 8. An example of an SP-DAG.

Definition 11. *SP-DAGs are graphs with a single source and sink nodes which can be built using the following rules [23]. Let G_1 and G_2 be two SP-DAGs then*

- (base case) *connecting two nodes $A \to B$ creates an SP-DAG;*
- (sequential composition) *identifying the sink of G_1 with the source of G_2 makes a new SP-DAG;*
- (parallel composition) *identifying the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2 makes a new SP-DAG.*

The property on SP-DAG is expressed as follows.

Property 12. *Any SP-DAG can be compressed using Rules (C_1), (C_2), (S_1), and (S_2) into a sorted chain.*

Proof. By induction based on the Def. 11 of SP-DAGs .
∘ **Base case**. The basic dag $A \to B$ is sorted (possibly into a single node) using Rules (C_1) and (C_2) (Prop. 7).
∘ **Sequential case**. The sequential composition of two SP-DAGs G_1 and G_2 supposes that the sink of G_1, say A, is also the source of G_2. By induction hypothesis, G_1 and G_2 can be compressed into two sorted chains. If A has been clustered into G_1 or G_2 we extract it: a clustered node $(X;A)$ or $(A;X)$ can be unclustered into the chains $X \to A$ and $A \to X$. The chains are joined in A in a single chain which can be sorted. ∘ **Parallel case**. The parallel composition of two SP-DAGs G_1

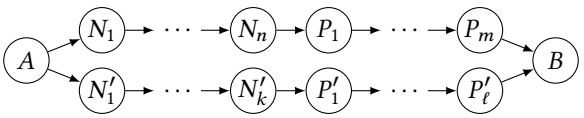


Fig. 9. Parallel composition of two sorted chains.

and G_2 supposes that the source and sink of G_1, say A and B, are also the source and sink of G_2. By induction hypothesis, G_1 and G_2 can be compressed into two sorted chains. If A or B have been clustered into G_1 or G_2 we can extract them as before. The two chains are joined in A and B to get a graph of the form depicted in Fig. 9. If A is connected to two negative nodes N_1 and N_1' then Rule (S_1) always applies. We can sequentialize N_1 or N_1' depending on which has the greater peak and suppress the edge of $A \to N_1'$ or $A \to N_1$ by (local) transitive reduction. Similarly, if the two predecessors of B are positive nodes then Rule (S_2) always applies. In both cases, one of the parallel branches gets shorter: new nodes are serialized before A and after B to eventually form a linear chain. The problematic case is when A has a positive successor and a negative one and B has a positive predecessor and a negative one. In this case, the conditions to apply Rules (S_1) and (S_2) may not be satisfied. However, since the two chains are sorted, this case can occur only when one chain is made of only positive nodes $P_1, \dots P_n$ while the other is composed of only negative ones $N_1, \dots N_m$. The two chains being maximally clustered, we know that:

- the peak condition of Rule (C_1) is false on all the P_i, which entails that $\pi_{P_2} + \iota_{P_1} < \pi_{P_1}, \pi_{P_3} + \iota_{P_2} < \pi_{P_2}, \dots$ and since all impacts are positive: $\pi_{P_m} < \pi_{P_1}$;
- the peak condition of Rule (C_2) is false on all the N_i, which entails that $\pi_{N_1} < \pi_{N_2} + \iota_{N_1}, \pi_{N_2} < \pi_{N_3} + \iota_{N_2}, \dots$ and since all impacts are negative: $\pi_{N_1} < \pi_{N_m}$.

Now, assume that N_1 and P_1 cannot be sequentialized because the condition of Rule (S_1) is not satisfied *i.e.*, because $\pi_{P_1} < \pi_{N_1}$. Then, we just pointed out that $\pi_{P_m} < \pi_{P_1} < \pi_{N_1} < \pi_{N_m}$ and since ι_{P_m} is positive and ι_{N_m} is negative we have $\pi_{N_m} + \iota_{P_m} > \pi_{P_m} + \iota_{N_m}$ which is the condition of Rule (S_2) to sequentialize P_m. Therefore, in any case, there always exists node that can be sequentialized. The

parallel composition is eventually transformed into a linear chain which can be sorted by Rules (C_1) and (C_2) if needed. □

According to Prop. 12 any SP-DAG representing a dataflow application can be compressed into a linear chain representing a schedule. Being a dataflow application, all prefixes of this chain have a positive impact and Corollary 8 applies.

Theorem 13. *Any SP-DAG representing a dataflow application can be compressed using Rules (C_1), (C_2), (S_1), and (S_2) into a single node.*

Even if Alg. 1 is effective on SP-DAGs, a simpler recursive algorithm expressed along Def. 11 decomposition would be more time-efficient. Global nested iterations and transitive reduction become useless and sequentialization can be specialized to deal only with the case of Fig. 9.

6 COMPOSITIONAL ANALYSIS

When dealing with big graphs a natural idea is to try to decompose the search for an optimal schedule by analysing its subgraphs. Being local, our transformation rules implement some kind of compositional analysis. In particular, when a subgraph is compressed to a single node or a chain we can say that its optimal schedule has been found. However, there may be cases where subgraphs cannot be compressed at this level and a compositional analysis would help. A natural candidate for such modular analysis are single-source single-sink subgraphs. Such graphs, called S-T subgraphs, are connected to the surrounding graph only by incoming edges to its source node S and outgoing edges from its terminal node T (see Fig. 10). The key property of such graphs is that they can be replaced by their schedule (*i.e.*, represented as a chain of nodes) without violating any dependencies. Otherwise, internal incoming edges from and outgoing edges to the enclosing graph might hide dependencies that should be considered when analyzing the subgraph.

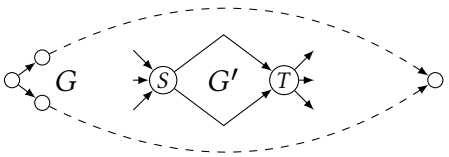


Fig. 10. A S-T subgraph G' inside graph G.

Unfortunately, replacing an S-T subgraph by one of its optimal schedule might be suboptimal w.r.t. the global memory peak. For instance, consider two schedules of a subgraph having the same global peak but different minimal valleys. The schedule with a lower minimal valley might be a better choice in order to interleave a high peak node of the enclosing graph. Without further information, we are left to generate all schedules of the S-T subgraph and proceed to analyze the the global graph by replacing the subgraph with all its possible schedules in turn. Fortunately, we can characterize optimal schedules for composition.

We first define a preorder relation $\preceq_s$ between schedules and show that replacing a S-T subgraph by a lower schedule will always lead to a better global result than replacing it by a greater one. The intuition is that interleaving any element in a lower schedule The intuition is that interleaving any element in a lower schedule should yield a better global memory peak than interleaving the same element in a greater one.

Sequentialization defines the optimal interleaving of an element within a chain (see Sec. 5.3). The preorder relation is defined so that the sequentialization of any element into two ordered chains preserves that relation.

Consider the insertion by sequentialization of a new negative node in the negative part of a sorted schedule (*i.e.*, with increasing relative peaks and a global peak located in its middle, see Sec. 5.2). The new global peak either (i) remains unchanged, or (ii) is replaced by the peak at the left of the insertion (*e.g.*, if the new node has a sufficient negative impact) or (iii) becomes the peak of the new node (*e.g.*, which has a sufficient negative impact and a peak larger than its left node).

We first define two functions returning the new relative peak at the insertion point for negative and positive nodes with peak x.

Definition 14 (Comparison functions). *Let S be a sorted chain of the form $S = N_1 \to \cdots \to N_n \to P_1 \to \cdots \to P_m$, with N_i and P_j denoting respectively negative and positive nodes. We define two functions $\hat{f}_S^- : \mathbb{N} \to \mathbb{N}$ and $\hat{f}_S^+ : \mathbb{N} \to \mathbb{N}$ as*

$$\hat{f}_S^-(x) = \max(\pi_{N_i} + \Upsilon^S_{N_{i-1}}, x + \Upsilon^S_{N_i}) \quad \text{if } x \in [\pi_{N_i}, \pi_{N_{i+1}})$$
$$\hat{f}_S^+(x) = \max(\pi_{P_{j-1}} + \Upsilon^S_{P_j}, x + \Upsilon^S_{P_j}) \quad \text{if } x \in [\pi_{P_{j+1}} - \iota_{P_{j+1}}, \pi_{P_j} - \iota_{P_{j+1}})$$
$$\text{by convention } \pi_{N_0} = \pi_{P_{m+1}} = 0 \text{ and } \pi_{N_{n+1}} = \pi_{P_0} = +\infty$$

With the convention that $\pi_{N_0} = \pi_{P_{m+1}} = 0$ and $\pi_{N_{n+1}} = \pi_{P_0} = +\infty$, the set of strictly increasing (Prop 9) intervals $\{[\pi_{N_i}, \pi_{N_{i+1}})\}_{0..n}$ is a partition of $\mathbb{N}$. It follows that $\hat{f}_S^-$ is defined for all $x \in \mathbb{N}$. A similar remark can be made for $\hat{f}_S^+$.

Def. 14 illustrates again the symmetry of sorted chains. Instead of defining the function $\hat{f}_S^+$ we could have used $\hat{f}_S^-$ on the mirror image (see Sec. 5.2) of $P_1 \to \cdots \to P_m$. Equipped with these functions, we can define the appropriate preorder relation $\preceq_s$.

Definition 15 (Preorder $\preceq_s$). *Let S and S' be two sorted chains.*

$$S \preceq_s S' \Leftrightarrow \begin{cases} S \preceq_p S' & \wedge \\ \forall x \in \mathbb{N},\ \hat{f}_S^-(x) \le \hat{f}_{S'}^-(x) & \wedge\ \hat{f}_S^+(x) \le \hat{f}_{S'}^+(x) \end{cases}$$

The relation $S \preceq_s S'$ ensures that the global peak of S is less than or equal to the global peak of S' ($S \preceq_p S'$) and remains so whatever node is sequentialized into S and S'.

Property 16. *Let S and S' be two sorted chains, let X be an arbitrary node and let $S \| X$ (resp. $S' \| X$) denote the sequentialization of X into S (resp. S') according to Rules (S_1) and (S_2), then*

$$S \preceq_s S' \Rightarrow S \| X \preceq_p S' \| X$$

This property generalizes to the sequentialization of sorted chains $S \| T$ and $S' \| T$ when $S \preceq_s S'$ and S, S' and T being sorted chains. Each sequentialization step inserts a node T_i and produces new schedules of the form $N; (S_r \| T_r); P$ and $N'; (S'_r \| T_r); P'$ increasing the sequential chains (N, P, N' and P') and decreasing the chains in parallel (S_r, S'_r and T_r). The remaining chains S_r and S'_r being subsequences of S and S', the relation $S_r \preceq_s S'_r$ still hold.

Corollary 17. *Let S, S', and T be three sorted chains and let $S \| T$ (resp. $S' \| T$) denote the sequentialization of T into S (resp. S'). Then we have*

$$S \preceq_s S' \Rightarrow S \| T \preceq_p S' \| T$$

Assume that we replace a S-T subgraph by one of its schedules $S = S_1; \ldots, S_n$ and compute the optimal global schedule. That global schedule is of the form $G_s = A; S_1; X_1; S_2; \ldots; X_{n-1}; S_n; B$ with S_1 and S_n being the source and sink nodes of the S-T subgraph, the X_i being interleaved sub-schedules of the enclosing graph. If there is another S-T subgraph schedule R whose sorted version is lower ($\preceq_s$) than the sorted version of S, then Corollary 17 ensures that the new schedule

$G_R = A;(R\|(X_1;\ldots;X_{n-1}));B$ has a better global peak *i.e.*, $G_R \preceq_p G_s$. This means that when considering the schedules of an S-T subgraph we can prune all greater schedules w.r.t. $\preceq_s$.

Our implementation and experiments showed that this pruning always keeps a single schedule. In order to prove that this is always the case, we rely on a recent article [12] that defines a similar preorder relation between schedules, called *dominance* and written $\trianglelefteq$. They prove that a task graph always has a schedule that dominates all others.

We use their result to show that there is a schedule which is lower than all others with our relation as well. We describe the dominance relation in the appendix along with proofs of the two next properties.

The first issue is that the dominance relation is defined on any schedule whereas ours is defined only on their sorted form. Actually, clustering preserves the dominance relation. That is to say, a schedule S and its maximally clustered version $\overline{S}$ dominates each other.

Property 18. *Let S be a schedule and $\overline{S}$ its sorted version. Then $S \trianglelefteq \overline{S} \trianglelefteq S$.*

An easy proof (see appendix D) shows that a chain and the same chain with two nodes clustered by Rule (C_1) or Rule (C_2) dominates each other. Prop. 18 follows by induction on the number of clustering steps from S to $\overline{S}$.

Next it can be shown that the dominance relation implies our $\preceq_s$ relation.

Property 19. *Let S and S' be two sorted schedules of a task graph G, then $S \trianglelefteq S' \Rightarrow S \preceq_s S'$.*

According to [12], any task graph has a schedule S^d that dominates all others. Furthermore, Prop. 18 ensures that its sorted form $\overline{S}^d$ also dominates all schedules (and their sorted forms), and Prop. 19 ensures that it is a least element for our relation as well. Therefore, this smallest schedule can be used to replace the associated S-T subgraph without loosing optimality of the global analysis. We explain in Sec. 7.3 how such a schedule is computed.

7 BRANCH AND BOUND ALGORITHM

On some DAGs, our global compression algorithm is not able to compress the graph into a single node. Instead, it produces a smaller schedule graph whose optimal memory peak has yet to be found. As already mentioned, this can be done by computing all the linear extensions of the DAG and their memory peak. This naive approach is costly and, in our experience, can deal only with very small graphs (smaller than 15 nodes).

We propose an optimized B&B algorithm that builds the tree of all schedules in a depth-first manner. The depth of this tree is equal to the number of tasks and each branch corresponds to a decision to schedule one node of the DAG, chosen in the so-called ready list that contains all the nodes having all their predecessors already scheduled. Initially, the ready list contains the source nodes of the DAG.

Even though the theoretical time complexity of B&B remains factorial in the size of the DAG, as we compute the *minimum* memory peak we expect that cutting branches in the exploration tree will result in a better practical time complexity.

Our B&B algorithm is equipped with several heuristics and optimizations. We first present the heuristic used to sort the ready list and guide the exploration and several heuristics to get a good initial upper bound. We describe two optimizations allowing to prune many schedules and finally sketch how B&B can be used to find the smallest schedule described in the previous section.

7.1 Heuristics for Branch and Bound

Our B&B algorithm can be tuned with many heuristics. Suboptimal ones could consider a partial exploration of the tree of all schedules, for example,through the use of a timeout. Here, we use

heuristics to enhance the B&B algorithm while preserving its optimality. We use an heuristic to sort ready lists in order to explore the most promising nodes first and some others to find a good initial upper bound for B&B.

7.1.1 Heuristic for Sorting the Ready List. In our implementation, the ready list is sorted in the same order than the optimal scheduling of independent tasks (see Sec. 5.3). Negative nodes come first in an increasing peak order and positive nodes come after in a decreasing drop order. This order serves as the strategy for the exploration of the tree of schedules.

7.1.2 Heuristic for the Initial Upper Bound. The quality of a good initial upper bound is important because the smaller it is, the more branches will be cut. It is therefore worth spending a little more computation time on this heuristic in order to save more computation time on the B&B algorithm.

Two heuristics are considered to find an initial upper bound. In both cases, we use the value returned by the first traversal of the B&B algorithm (starting with an initial memory peak value equal to $+\infty$) but with different sorting of the ready lists:

- the ready lists are sorted with negative nodes first, ordered by increasing impact. Nodes with equal impacts are ordered such that those having a direct successor with a smaller impact among have priority. Node peaks are not considered to favor negative nodes triggering another negative node;
- the ready lists are sorted in the same order as the optimal scheduling of independent tasks as described above in Sec. 7.1.1.

The cost of such heuristics is linear in the number of edges. As they have a low computational cost, we call them on different versions of the graph: the schedule graph before compression, after compression and their mirror images obtained by symmetry (as described in Sec. 3). They all share the same minimal memory peak[1]. Depending on the graph, each heuristic can give a better result on a version of the graph than on another version. Combining the two heuristics with the four possible graphs yields eight possible initial values and we can start the B&B algorithm with their minimum.

7.2 Optimizations for Branch and Bound

The optimizations of the B&B algorithm are: (1) a longer backtrack when the current schedule is completed, and (2) a smaller ready list when it contains negative nodes. These optimizations enable us to optimally solve graphs up to 50 nodes in a second.

7.2.1 Fast Backtracking. Our first optimization takes advantage of the minimization nature of our problem. When building a sequential schedule, we retain the node X that caused the first occurrence of the memory peak value. When we reach the end of the scheduling tree with a full schedule, we backtrack directly to the immediate predecessor of X in the tree. Indeed, since the memory peak of this schedule was already reached when scheduling X, the entire sub-tree starting at X cannot result in a strictly smaller memory peak.

This direct and long backtracking significantly reduces the computation time by pruning the tree of schedules. If we assume a constant branching factor b for the scheduling tree and a number of k remaining nodes to be scheduled from the backtracking point, then the number of pruned schedules is b^k.

7.2.2 Direct Scheduling of Negative Nodes. Our second optimization takes advantage of the impact of nodes to prune the ready list after each newly scheduled node. Assume that we are at step n with the ready list Rdy_n, the current memory used $MemCurr_n$, and the current memory peak

[1]For reversed versions, if the sum of all impacts is not null, the peak of a schedule has to be offset by this sum.

$MemPeak_n$. Pruning occurs when Rdy_n contains at least one negative node: we select the negative node A ($\iota_A \leq 0$) with the smallest peak π_A among the other negative nodes. Two cases are possible:

- If A does not increase the memory peak ($MemCurr_n + \pi_A \leq MemPeak_n$), we replace the ready list by just $\{A\}$. Since A leaves the global peak unchanged and reduces the memory level, nothing is lost by scheduling A right away.
- Otherwise, we remove from the ready list all nodes B such that $\pi_A \leq \pi_B$. Indeed, since A does not increase the current memory, no gain can be expected by scheduling at this step a node having a larger peak. However, nodes with a positive impact and a smaller peak can eventually lead to a better peak and must remain in Rdy_n. This pruning can be seen as an on-line application of Rule (S_1): when $\iota_A \leq 0 \wedge \pi_A \leq \pi_B$, then A can be executed before B.

In both cases, the nodes removed from the ready list at this step remain to be scheduled and will be in the ready lists of subsequent steps. Pruning nodes from Rdy_n yields a computational benefit when backtracking to this step n. If we assume a constant branching factor b and a number of k remaining tasks to be scheduled after step n, then removing a single node from Rdy_n prunes b^{k-1} schedules.

Our B&B algorithm finds the optimal memory peak of applications up to 50 nodes in a second, but its computational time generally explodes past 100 nodes. Combined with our compression algorithm, many task graph applications can be analyzed optimally. At worst, a timeout always permits to retrieve an overestimation of the peak, namely the best schedule found so far.

7.3 Finding the Smallest Schedule

The B&B algorithm needs some precautions to be used to find the smallest schedule needed for our compositional analysis (see Sec. 6). The standard version of our B&B algorithm cuts any branch resulting in a schedule with a memory peak greater than or equal to the best schedule produced so far. To find the smallest schedule it must only cut any branches resulting in a schedule with a memory peak *strictly* greater than the memory peak of the best schedule produced so far. For the same reason, optimizations must be deactivated. Indeed, both cutting when equal and optimizations avoid the exploration of many schedules reaching the same memory peak whereas the smallest schedule could be one of them. In case of peak equality (which can only be assessed when the schedule is entirely built), we keep only the best schedule w.r.t. $\preceq_s$, as introduced in Def. 15. Upon completion, that is, when all the schedules with a minimal peak have been explored, the returned schedule is the smallest one.

Such method is somewhat slow and we have also implemented a version of algorithms 1 and 2 given in [12] to find a dominant schedule. Unlike their description, we use a minimal cut algorithm that works on the original graph. It is in polynomial time and ensures that the cut contains only edges in the flow direction. With this approach, a smallest schedule can be found using at most n calls to our optimized B&B algorithm. An example of use is described in Sec. 9.4.

8 APPLICATION TO SDF GRAPHS

As described in Sec. 2, SDF graphs are a strict extension of tasks graphs. Our technique can be applied to this model by first converting an SDF graph into an SRSDF graph [4], which is precisely a task graph. This standard transformation can be applied to *any* (even cyclic) consistent and live SDF graph. It produces a task graph encoding a minimal iteration of the initial SDF graph. Each actor in the SDF graph may induce multiple nodes in the task graph, corresponding to actor firings in the minimal iteration. In the worst case, the number of firings is exponential in function of the number of actors.

Despite the potential exponential blow up of this conversion, this approach remains effective to analyze the memory peak of many SDF graphs. There are nonetheless some SDF applications for which the resulting task graphs are too large to be optimally scheduled or to possess a schedule with a reasonable size.

The size of the schedule can always be reduced *a posteriori* thanks to classic string compression techniques (see, *e.g.,* LZ78 [26]), where repeated parts of the schedule are stored in the same procedure. Such schedule size optimization corresponds to the Smallest Grammar Problem [7], which is NP-complete. An advantage of this approach is that the schedule is smaller while remaining optimal. However it does not offer any control on the resulting size nor does it permit large compression ratios. Obviously, there are size constraints that require to consider suboptimal solutions.

We focus in this section on techniques that can produce schedules meeting specific limits. We partially expand the SDF graph into a Partial Expansion Graph (PEG) [24]. In this reduced SDF graph, multiple firings of the same actor are *grouped* together so that they are all fired together.

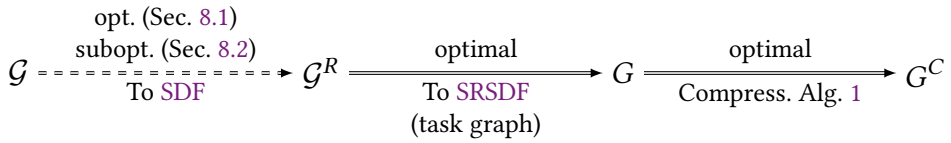


Fig. 11. Graph transformation chain for an SDF graph $\mathcal{G}$ to a compressed schedule graph G^C.

Fig. 11 summarizes the different transformation steps from the input SDF graph $\mathcal{G}$ to a reduced SDF graph $\mathcal{G}^{\mathcal{R}}$, which is converted to an SRSDF graph G, then compressed by our transformation rules into G^C and finally analyzed by the B&B algorithm. The SDF graphs $\mathcal{G}$ and $\mathcal{G}^{\mathcal{R}}$ have the same number of actors, but the total number of firings expressed in $\mathcal{G}^{\mathcal{R}}$ is lower than in $\mathcal{G}$. In the following, we present techniques to reduce the number of firings of each SDF actor, from $\mathcal{G}$ to $\mathcal{G}^R$. Some preserve the minimal memory peak (Sec. 8.1) others are suboptimal but can arbitrarily limit the expansion ratio (Sec. 8.2).

8.1 Peak-Preserving Schedule Compression

When grouping the firings of an actor A, its input (resp. output) rates s (resp. r) must be updated accordingly, as depicted in Fig. 12. This modification is *local* to the considered actor A and does *not* modify the balance equations of the graph. However, this modification may increase the node peak and impact of the resulting schedule graph, and the optimal memory peak is preserved only under some conditions presented hereafter.

Fig. 12. Firing reduction of actor A by divisor d.

Consider an actor A with its immediate predecessors and successors. To consume all tokens produced by *one* firing of each predecessor, or to produce all tokens consumed by *one* firing of each successors, multiple firings of A may be necessary. When both cases occur for the same actor,

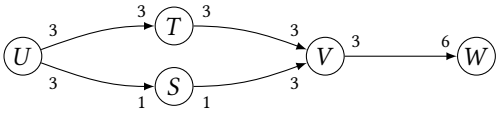


Fig. 13. SDF example: a simple graph $\mathcal{G}$ with minimal iteration $\{\#U = 2, \#T = 2, \#S = 6, \#V = 2, \#W = 1\}$.

this allows us to group several of its firings without modifying the memory peak of the sequential schedule. Indeed, such situation leads to multiple firings in parallel that all have a common set of predecessors and a common set of successors, which allow the application of Rule (S_1) or Rule (S_2). Once sequentialized, since all firings have the same peak and impact, our clustering Rule (C_1) or Rule (C_2) can merge them to a single node proving that the grouping of these firings preserves optimality.

For example, this case occurs for node S in Fig. 13: a single firing of its predecessor U enables 3 firings of S, and its successor V requires 3 firings of S, these firings can then be regrouped.

Formally, for a given actor A, the number of firings to regroup, denoted $\text{firingDiv}(A)$, is defined in Eq. (4) where $\{d_1 = 1, \ldots, d_z = \#A\}$ is the ordered set of divisors of $\#A$, r° or s° denote the rate values in the original SDF graph and r or s denote the current rate values:

$$\text{firingDiv}(A) = \max_{k\in[1,z]} \left\{ d_k \,|\, d_k \leq \min \left\{ \min_{A \xrightarrow{r} \xrightarrow{s} X} \left\{ \left\lfloor \frac{s}{r^\circ} \right\rfloor \right\}, \min_{X \xrightarrow{r} \xrightarrow{s} A} \left\{ \left\lfloor \frac{r}{s^\circ} \right\rfloor \right\} \right\} \right\} \tag{4}$$

It is the minimum number of firings enabled by one firing of each predecessor of A and required by one firing of each successor of A. When $\text{firingDiv}(A)$ is defined, that is when at least one divisor meets the condition in the max operator, A is updated according to the rule in Fig. 12 with $d = \text{firingDiv}(A)$. Otherwise A is not updated.

These grouping must be remembered in order to produce a faithful schedule graph. Consider, for instance, the grouping of three firings of S in Fig. 13 whose input and output rates are now 3. By considering the regrouped S as a standard task, the corresponding node in a schedule graph will have as peak and impact $\pi_S = 0$ and $\iota_S = 0$ in the CBP model and $\pi_S = 3$ and $\iota_S = 0$ in the PBC model. However, this last peak is wrong since it does not take into account the fact that a token is consumed after each firing. The correct peak should be $\pi_S = 1$.

For the generic grouping of Fig. 12, the correct values can be established by considering A^d as a sequence of tasks $A_1; \ldots; A_d$ and computing the global peak and impact according to the model considered. Let $\iota_A = r^\circ - s^\circ$, applying the rules of Sec. 3, we get

$$\begin{aligned} &\pi_{A^d} = \max(0, d \cdot \iota_A) && \text{and} && \iota_{A^d} = d \cdot \iota_A && \text{in the CBP model} \\ &\pi_{A^d} = \max(0, (d-1) \cdot \iota_A) + r && \text{and} && \iota_{A^d} = d \cdot \iota_A && \text{in the PBC model} \end{aligned} \tag{5}$$

8.2 Non Peak-Preserving Schedule Compression

We first present a coarse but well-known conversion that produces a task graph G with the same number of nodes as actors into the SDF graph $\mathcal{G}$, *i.e.*, $|\mathcal{G}| = |\mathcal{G}^R| = |G|$. Such coarse conversion prevents any expansion and allows the analysis of any actual SDF graph, but the resulting schedules have a much larger memory peak than the optimal one. We then present a more refined technique inspired from the work on PEGs [24], which can incrementally reach an objective expressed in terms of a maximum number of firings, and can be combined with the optimal transformation presented in Sec. 8.1.

8.2.1 Flat Single Appearance Schedules. To reduce the number of nodes in the expanded graph, a drastic approach is to ensure that each actor X executes its $\#X$ firings in a row. To enforce this

policy, each edge rate is set to the total production or consumption of an iteration as in Eq. (1). Each node X therefore executes once in the schedule and the graphs $\mathcal{G}$, $\mathcal{G}^R$, and G have the same number of nodes. The schedules of such a graph correspond to what is called flat Single Appearance Schedules (SASs) in [4].

For instance, regrouping all the firings of each actor in Fig. 13 is enforced by changing the production and consumption rates of all edges to 6. Denoting X^k the actor that executes X k times in a row, all actors in the resulting graph $\mathcal{G}^R$ are of the form $X^{\#X}$. For Fig. 13, the resulting expanded graph $\mathcal{G}^R$ has the same topology but with the actors U^2, T^2, S^6, V^2, and W^1, and with all the input and output rates equal to 6. It follows that the corresponding SRSDF graph G is identical to $\mathcal{G}^R$, and the resulting schedule is $U^2;T^2;S^6;V^2;W$, which is a flat SAS.

A flat SAS schedule usually has a much higher memory peak than the optimal one obtained for the SRSDF expansion of the same SDF graph. The flat SAS schedule prevents different firings of a given node from being interleaved, in particular firings of negative nodes between firings of positive nodes. For example, in the PBC model with Eq. (5), the previous flat SAS schedule for Fig. 13 yields a peak of 15 tokens, against only 12 tokens for the optimal peak.

8.2.2 Partial Expansion Graph Transformation. Instead of grouping all the firings of an actor, a more refined technique is to consider any possible grouping of its firings according to its integer divisors. For example, the actor S in Fig. 13 could be transformed either into the actor S^2, S^3, or S^6, respectively appearing thrice, twice, or once in the SRSDF graph and in the schedule.

Our PEG transformation is based on this idea: we modify the graph progressively by setting at each step a greater firing divisor to one of the actors. Each modification leads to a suboptimal schedule, and iterating this process to limit the expansion ratio as requested.

Since a PEG transformation may increase the memory peak, we define a score criterion to always start with the actor with the smallest memory requirement. Let $\{d_1 = 1, \ldots, d_z = \#A\}$ be the ordered set of divisors of $\#A$. Each divisor d_k yields the corresponding input (resp. output) rates $s_k = d_k \cdot s$ (resp. $r_k = d_k \cdot r$). The score of A currently at divisor k is defined by Eq. (6):

$$\text{score}(A,k) = \sum_{A,k \xrightarrow{r_k\ s_\ell} X,\ell} \left\lceil \frac{s_\ell}{r_{k+1}} \right\rceil r_{k+1} - \left\lceil \frac{s_\ell}{r_k} \right\rceil r_k + \sum_{X,\ell \xrightarrow{r_\ell\ s_k} A,k} \left\lceil \frac{s_{k+1}}{r_\ell} \right\rceil r_\ell - \left\lceil \frac{s_k}{r_\ell} \right\rceil r_\ell \tag{6}$$

The memory requirement of A is the sum of the tokens that A must produce to enable a single firing of each of its successors, and the tokens that must be produced by its predecessors for a single firing of A. The criterion $\text{score}(A,k)$ is defined as the difference between the memory requirements using the divisor d_{k+1} (after the modification) and d_k (before).

Consider actor S in Fig. 13. For its first divisor 2, we have $\text{score}(S,1) = 1$: indeed, U must be fired once (as before) to enable S^2, but S^2 must be fired twice to produce 4 tokens to enable V to fire, whereas S needed to produce only 3 tokens to enable V. For its second divisor 3, the memory requirement does not change and $\text{score}(S,2) = 0$: indeed, only 3 tokens must be produced by A and then by V as before.

Our PEG algorithm takes an objective expressed as a maximum of firings for the whole graph. Until the objective is reached, it repeats the local transformation of Fig. 12 by selecting at each step the pair (A,k) (k being the current divisor of A) that minimizes the score (Eq. (6)). Then, the score is updated only for node A and its immediate neighbors. Optionally, the optimal grouping of firings presented in Sec. 8.1 may be executed between each step. If the objective in terms of number of firings is equal or less than the number of actors, the result will be the same as in Sec. 8.2.1.

9 EXPERIMENTS

We have conducted a series of experiments to evaluate several of our contributions: compression of task graphs (sec. 9.1), partial expansion of SDF graphs (sec. 9.2), heuristics for B&B initial values (sec. 9.3), and compositionality (sec. 9.4). When possible, we compare the memory peaks obtained with previously known results. Our experimental setup is a regular laptop (Intel® Core™ i5-8265U @1.60GHz processor, 16 GB of RAM), with Linux Ubuntu 22.04. We use Python 3.10 with the graph library Networkx 2.8.8. Our implementation is available online.[2] All the results presented here consider the PBC model.

9.1 Optimal Memory Peak Analysis by Compression

Concerning task graphs, most existing work focus on tree-shaped graphs or SP-DAGs but do not provide benchmarks. For such graphs, the optimal memory peak can be found very quickly, even for very large ones. General SDF graphs are more challenging especially after their conversion onto SRSDF graphs. In all examples, even huge ones with thousands of tasks, we find the optimal memory peak using only compression.

Table 1. Memory peaks for the satellite application.

satellite	$\|G\|$	[19]	[18]	[13]	[ours]	sec.
flat SAS	22	1,920	—	**1,680**	**1,680**	0.002
SDF	4,515	—	991	**960**	**960**	7.7

Table 1 presents the results for satellite, one of the first SDF applications that was used to analyze the memory peak. It can be handled directly as a task graph with 22 nodes (by analyzing only its flat SAS schedules), and as a 4,515 node task graph after conversion into its SRSDF version. Ritz *et al.* [19] consider only flat SAS schedules and expressed the problem as an Integer Linear Programming (ILP) problem. In 1995, it took four days to provide an overestimation. Murthy *et al.* [18] consider the SDF application and use heuristics to produce memory efficient schedules whose peak is evaluated afterwards. The technique presented in [13] finds optimal memory peaks for SP-DAGs and overestimated peaks for general task graphs. Even if the two versions of satellite (flat SAS and SDF) are not SP-DAGs, [13] finds the optimal memory peaks for both. Our graph transformations compress both versions into a single node, providing the optimal peaks in a short runtime (last two columns in Table 1).

Table 2 presents results obtained on Quadrature Mirror Filterbanks (QMFs), a well-known tree-structured class of signal processing applications often used in benchmarks for SDF. Different versions of filterbank exist: the SDF graph topology remains the same but the number of nodes (nesting depth) and the rates vary. We evaluate the memory peak on the nine filterbank versions used in [18], where qmf[ij]_[l]d (resp. qmf[ijk]_[l]d) refers to a version with rates i and j (resp. i, j, and k) and a depth ℓ. The $|G|$ column shows the total number of nodes of the SRSDF graph obtained after conversion. The next column gives the suboptimal results obtained by [18] based on the same heuristics as for satellite. The SRSDF conversions of filterbank are not SP-DAGs and the technique of [13] only finds overestimations. The results of our algorithm (Alg. 1) are given in the last two columns of Table 2 where we indicate the memory peak found and the total runtime. On all filterbank versions, our graph transformation rules always compress the corresponding SRSDF graph to a single node with the optimal memory peak.

[2] https://gitlab.inria.fr/spades-pub/mastag

Table 2. Memory peaks for different qmf filterbank benchmark applications in [18].

filterbank	$\|G\|$	[18]	[13]	[ours]	sec.
qmf23_2d	78	22	18	**13**	0.007
qmf23_3d	324	63	53	**31**	0.06
qmf23_5d	4,536	492	405	**247**	6.7
qmf12_2d	40	9	10	**7**	0.003
qmf12_3d	112	16	20	**11**	0.009
qmf12_5d	704	58	79	**35**	0.1
qmf235_2d	190	55	45	**22**	0.03
qmf235_3d	1,300	240	133	**47**	0.7
qmf235_5d	50,000	5,690	1,190	**272**	802.5

9.2 Partial Expansion of SDF Graphs

The PEG reduction is useful when compression cannot be completed in a reasonable time and/or to reduce the size of schedules. In Table 3, we evaluate the PEG algorithm presented in Sec. 8 to get shorter schedules for qmf23_5d and qmf235_5d (for which we found optimal schedules but with lengths of 4,536 and 50,000 tasks respectively).

Table 3. Memory peaks for the PEG reductions of the largest task graphs of Table 2.

filterbank	method	PEG with increasing objectives starting from flat SAS				
		188	400	800	1,600	3,200
qmf23_5d	Eq. (6)	729	351	283	267^{*}	$253^{\dagger}$
	Eqs. (4) to (6)	487	298	268	259^{*}	248
qmf235_5d	Eq. (6)	9,375	4,750	$3,625^{*}$	$3,625^{*}$	$2,556^{\dagger}$
	Eqs. (4) to (6)	6,252	4,127	$3,397^{*}$	$3,377^{*}$	$2,379^{\dagger}$

In Table 3, we present the memory peaks found for qmf23_5d and qmf235_5d with different PEG objectives expressed as number of firings/tasks: 188 (which is the flat SAS of both qmf), 400, 800, 1,600, and 3,200. Each benchmark is evaluated twice: once with the suboptimal firing reduction technique based on Eq. (6), and once with the addition of optimal steps computed by Eq. (4) and the peak refinement of Eq. (5).

In almost half of the cases (marked by $*$ and †), the corresponding task graph is not reduced to a single node and the B&B algorithm is applied. In three of those cases (marked by †), the compressed schedule graph is too large for B&B (123 for qmf23_5d, 174 and 225 nodes for qmf235_5d) which stopped at its 600 sec. timeout and returned an overestimated peak. These results show the trade-off between the schedules size and the memory peak. For qmf235_5d, dividing the number of tasks by ~ 266 (from 50,000 to 188, which is the flat SAS size) multiplies the memory peak by ~ 23 (from 272, the optimal peak, to 6,252 with Eqs. (4) and (5)). Subsequent increases of the number of firings improve the overestimation of the memory peak.

9.3 Heuristics for B&B Initial Values

The memory peak of qmf235_5d with the objective of 3,200 firings is now better (2,556) than the one we previously found (3,275) in [10] with the same settings. This is due to the use of new heuristics to find good initial values for the B&B algorithm (see Sec. 7.1.2).

Table 4 evaluates these heuristics on qmf235_5d with two different PEG objectives and on its SRSDF version. The heuristics H1 and H2 denote the two different sortings of the ready list listed in Sec. 7.1.2. They are either applied on the original graph (H1 and H2) or its reversed version ($H1^R$ and $H2^R$). Furthermore, these four heuristics are applied on the graph before and after compression. They are not applicable to the compressed SRSDF version which is a single node. The last column recalls the best memory peak found for these graphs (see Tables 2 and 3).

Initial values are often close to the best peak found afterwards by the B&B (and sometimes directly finds it). These examples show that there is not a single best heuristic or graph version. For this reason, we always call H1 and H2 on the different versions of the graph (before and after compression, in the original or reversed order) and initiate the B&B algorithm with the minimal identified value.

Table 4. Initial upper bounds found by the heuristics for various qmf235_5d graphs.

qmf235_5d version	Before compression				After compression				Best peak
	H1	$H1^R$	H2	$H2^R$	H1	$H1^R$	H2	$H2^R$	
PEG 1,600	4,726	4,598	7,469	7,469	3,397	3,422	**3,377**	3,397	3,377
PEG 3,200	3,653	3,571	7,457	7,458	3,292	**2,397**	3,283	2,470	2,379
SRSDF	**462**	**462**	5,208	5,208	N/A	N/A	N/A	N/A	272

9.4 Compositionality

Our benchmarks do not exhibit large S-T subgraphs where a compositional analysis (Sec. 6) could be beneficial. Hierarchical task graphs are interesting candidates but they are used in a parallel context [9]. We have not found any hierarchical task graph that could be used for benchmarking. Thus, we have built a synthetic graph made of four incompressible S-T subgraphs. Each S-T subgraph contains 32 nodes and the four of them are connected in parallel in such a way that no transformation rule applies on the complete graph (further details can be found in the implementation). Solving it directly with the B&B algorithm times out after 600 sec. and returns an overestimated memory peak at 499. The compositional analysis first finds the smallest schedule (w.r.t. $\preceq_s$) of each of the four S-T subgraphs using the B&B algorithm. After replacing each instance by these schedules, the complete graph is compressed and analyzed using B&B. The optimal peak, with a value of 394, is found within a total time of five seconds. Even if S-T subgraphs do not appeared in our benchmarks, our current implementation automatically detects them. They are solved compositionally when they contain a maximum of 50 nodes which is the limit for which our B&B algorithm always finds a solution in a timely manner.

10 RELATED WORK

Much work aiming at minimizing memory requirements for SDF graphs assume a *non-shared model* where buffers are allocated independently in memory [4, 11, 21]. The goal of analyses is to find the minimal size of each buffer to ensure a live execution.

The more memory efficient *shared buffer model* assumes that buffers are allocated in the same global shared memory. The analyses focus on the gobal memory needed *i.e.*, minimizing the memory peak. Work on sequential scheduling of dataflow graphs for memory peak minimization can be classified into two categories: optimal solutions for a subclass of task graphs and suboptimal solutions for SDF graphs.

Regarding optimal solutions, previous work presented algorithms to find the optimal schedules in $\mathcal{O}(n^2)$ for tree-shaped task graphs [15], and in $\mathcal{O}(n^3)$ for SP-DAGs [13]. Our graph transformations presented in Sec. 4 solve optimally these classes of graphs with the same time complexity. The notions of *hills* and *valleys* were present in [15] and correspond to our peaks and valleys. The notion of impact corresponds to *vertex weights* in [13]. However, our graph transformations are more general since they apply to any dag and, along with an optimized B&B algorithm, they are able to find optimal schedules for a much larger class of task graphs.

Recently, a preprint article [12] studied the memory peak analysis of task graph and introduced the notion of dominant schedule of a subgraph. An S-T subgraph can be replaced by its dominant schedule without loosing optimality for peak analysis. The principal outcome of their work is a proof that any S-T subgraph has a dominant schedule. We reuse this result to show that there always exists a lowest schedule for our preorder that can be used by our compositional analysis (see Sec. 6). In order to compute the dominant schedule they need an oracle returning for each graph a schedule minimizing the memory peak but do not propose any implementation. Our work is complementary since it provides an efficient technique to find such schedules.

Regarding overestimated solutions, previous work focused on SDF graphs and reduced the problem complexity by considering only SAS. While [19] considered only flat SAS, [18] relaxed this restriction to non flat SAS. In both cases, we outperform their benchmarks and find the optimal solution, as shown in Sec. 9. A relaxed class of SAS was also studied in [22] but we have not been able to compare with their results due to the unavailability of their code and benchmarks. Shared memory allocation might also be performed after scheduling [8], as a secondary minimization objective; this is useful for parallel scheduling, but is clearly suboptimal in the sequential case.

Finally, scheduling is not the only way to minimize memory needs. *Rematerialization* [5, 20] is another technique that releases data and recomputes it later on when needed instead of keeping it in memory during that interval. This technique is widely used in large neural networks [25], but at the cost of higher execution times.

11 CONCLUSION

In this article, we have tackled the problem of finding sequential schedules that minimize memory requirements of dataflow applications. We have proposed simple task graph transformations that compress task graphs and reduce the problem size. These transformations are local, they preserve the optimal memory peak, and can be applied on any task graph. The transformations alone permit to solve optimally the problem for a larger class of task graphs than before. This class is not formally characterized, but we proved that it includes SP-DAGs. When transformations do not compress a task graph into a single node, an optimized B&B algorithm explores the (reduced) task graph to find its optimal memory peak with the corresponding schedule.

Our graph transformations allowed a simple characterization of optimal schedules for sets of independent tasks. We showed that they always compress the important class of SP-DAGs into a single node representing their optimal schedule. We also designed an optimal compositional analysis dealing with single-source single-sink subgraphs (S-T subgraphs). These S-T subgraphs can be analyzed separately and replaced by their best schedule. We have not found real benchmarks to evaluate this technique, but its application to hierarchical task graphs is a promising avenue for further investigation.

Our approach also applies to SDF graphs after conversion to SRSDF graphs. All the standard SDF benchmarks we found were solved optimally. Still, the conversion may produce very large task graphs, and therefore very large schedules which can be a problem within an embedded context. We therefore designed a dedicated PEG transformation to reduce the problem size and therefore the schedule size. Together, the algorithms we have proposed significantly outperform the state of the art and reveal new optimal or better suboptimal bounds on benchmarks.

A natural extension of this work is to consider parallel schedules with shared memory. The optimal memory peak found by our approach for a sequential execution already provides a lower bound for all its parallel versions.

It would be also interesting to study the class of task graphs that are reduced to a single node. For instance, why are some PEG reduced versions of qmf23_5d compressed into a single node while others are not?

Finally, a more speculative avenue for further research is to investigate the application of our transformation-based approach to the analysis of other task graph properties.

ACKNOWLEDGMENTS

The authors would like to thank Basile Pesin for helpful discussions.

REFERENCES

[1] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. 1972. The Transitive Reduction of a Directed Graph. *SIAM J. Comput.* 1, 2 (1972), 131–137.

[2] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. 2021. Efficient Combination of Rematerialization and Offloading for Training DNNs. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., Virtual, 23844–23857. https://proceedings.neurips.cc/paper/2021/file/c8461bf13fca8a2b9912ab2eb1668e4b-Paper.pdf

[3] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. 2021. Pipelined Model Parallelism: Complexity Results and Memory Considerations. In *27th International European Conference on Parallel and Distributed Computing*. Springer, Lisbon, Portugal, 183–198. https://hal.inria.fr/hal-02968802

[4] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Pub., Hingham, MA.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1992. Rematerialization. *SIGPLAN Not.* 27, 7 (jul 1992), 311–321. https://doi.org/10.1145/143103.143143

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., Virtual, 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[7] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Transactions on Information Theory* 51, 7 (2005), 2554–2576. https://doi.org/10.1109/TIT.2005.850116

[8] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. 2015. Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs. *Journal of Signal Processing Systems* 80, 1 (July 2015), 19–37. https://doi.org/10.1007/s11265-014-0952-6

[9] Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Raymond Namyst, Samuel Thibault, and Pierre-André Wacrenier. 2023. Programming Heterogeneous Architectures Using Hierarchical Tasks. In *Euro-Par 2022: Parallel Processing Workshops*. Springer Nature Switzerland, Cham, 97–108.

[10] Pascal Fradet, Alain Girault, and Alexandre Honorat. 2023. Sequential Scheduling of Dataflow Graphs for Memory Peak Minimization. In *LCTES 2023 - 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, Orlando (FL), United States, 76–86. https://doi.org/10.1145/3589610.3596280

[11] Marc Geilen, Twan Basten, and Sander Stuijk. 2005. Minimising Buffer Requirements of Synchronous Dataflow Graphs with Model Checking. In *Proceedings of the 42nd Design Automation Conference, DAC'05*, William H. Joyner Jr., Grant

Martin, and Andrew B. Kahng (Eds.). ACM, San Diego, CA, USA, 819–824.
[12] Ce Jin, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang. 2023. New Tools for Peak Memory Scheduling. arXiv 2312.13526. arXiv:2312.13526 [cs.DS]
[13] Enver Kayaaslan, Thomas Lambert, Loris Marchal, and Bora Uçar. 2018. Scheduling series-parallel task graphs to minimize peak memory. *Theoretical Computer Science* 707 (2018), 1–23. https://doi.org/10.1016/j.tcs.2017.09.037
[14] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous Data Flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876
[15] Joseph W. H. Liu. 1987. An Application of Generalized Tree Pebbling to Sparse Matrix Factorization. *SIAM Journal on Algebraic Discrete Methods* 8, 3 (1987), 375–395. https://doi.org/10.1137/0608031 arXiv:https://doi.org/10.1137/0608031
[16] Svetlana Minakova, Erqian Tang, and Todor Stefanov. 2020. Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer International Publishing, Cham, 18–35.
[17] Rolf H. Möhring. 1989. *Computationally Tractable Classes of Ordered Sets*. NATO ASI Series, Vol. 255. Springer, Dordrecht, Netherlands, 105–193. https://doi.org/10.1007/978-94-009-2639-4_4
[18] Praveen K. Murthy and Shuvra S. Bhattacharyya. 2001. Shared Buffer Implementations of Signal Processing Systems Using Lifetime Analysis Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 2 (2001), 177–198. https://doi.org/10.1109/43.908427
[19] Sebastian Ritz, Markus Willems, and Heinrich Meyr. 1995. Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis. In *International Conference on Acoustics, Speech, and Signal Processing, ICASSP'95*, Vol. 4. IEEE, Detroit, MI, USA, 2651–2654. https://doi.org/10.1109/ICASSP.1995.480106
[20] Ravi Sethi. 1973. Complete Register Allocation Problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing* (Austin, Texas, USA) *(STOC '73)*. Association for Computing Machinery, New York, NY, USA, 182–195. https://doi.org/10.1145/800125.804049
[21] Tae-ho Shin, Hyunok Oh, and Soonhoi Ha. 2011. Minimizing Buffer Requirements forg Throughput Constrained Parallel Execution of Synchronous Dataflow Graph. In *Proceedings of the 16th Asia South Pacific Design Automation Conference, ASP-DAC'11*. IEEE, Yokohama, Japan, 165–170.
[22] Wonyong Sung, Junedong Kim, and Soonhoi Ha. 1998. Memory Efficient Software Synthesis Form Dataflow Graph. In *Proceedings of the 11th International Symposium on System Synthesis* (Hsinchu, Taiwan, China) *(ISSS '98)*. IEEE Computer Society, USA, 137—-142.
[23] Krishnaiyan Thulasiraman and M.N. Shanmukha Swamy. 1992. *Graphs - Theory and Algorithms*. Wiley, Hoboken, NJ, USA.
[24] George F. Zaki, William Plishker, Shuvra S. Bhattacharyya, and Frank Fruth. 2017. Implementation, Scheduling, and Adaptation of Partial Expansion Graphs on Multicore Platforms. *Journal of Signal Processing Systems* 87, 1 (Apr 2017), 107–125. https://doi.org/10.1007/s11265-016-1107-8
[25] Xunyi Zhao, Théotime Le Hellard, Lionel Eyraud-Dubois, Julia Gusak, and Olivier Beaumont. 2023. Rockmate: an Efficient, Fast, Automatic and Generic Tool for Re-materialization in PyTorch. In *International Conference on Machine Learning, ICML'23 (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, Honolulu, HI, USA, 42018–42045. https://proceedings.mlr.press/v202/zhao23b.html
[26] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934

Appendix

Many proofs below involve reasoning with max functions and inequalities. We first recall some basic properties. The function max is associative and commutative. We write $\max(x_1, \ldots, x_n)$ for any expression $E ::= x_i \mid \max(E_1, E_2)$ where each x_i with $(1 \leq i \leq n)$ occurs exactly once. The following rules are often used to simplify inequalities.

$$\begin{array}{rlcll} & \max(a,b) \leq \max(a,c) & \Leftrightarrow & b \leq \max(a,c) & \Leftrightarrow \quad \max(a,b) \leq c \\ \text{If } a \leq c \text{ then} & \max(a,b) \leq \max(c,d) & \Leftrightarrow & b \leq \max(c,d) & \\ \text{If } c \leq a \text{ then} & \max(a,b) \leq \max(c,d) & \Leftrightarrow & \max(a,b) \leq d & \end{array}$$

A CORRECTNESS PROOFS OF TRANSFORMATION RULES

Property 2 (Rule (C_2)). $\forall S,\ \iota_B \leq 0 \ \wedge\ \pi_B \leq \pi_A - \iota_A \ \Leftrightarrow\ (A;B;S) \preceq_p (A;S;B)$

Proof.

($\Rightarrow$) $\iota_B \leq 0 \ \wedge\ \pi_B \leq \pi_A - \iota_A$ is a sufficient condition. Indeed,

$$\begin{cases} \iota_B \leq 0 & \Rightarrow \quad \pi_S + \iota_A + \iota_B \leq \pi_S + \iota_A \\ \pi_B \leq \pi_A - \iota_A & \Rightarrow \quad \pi_B + \iota_A \leq \pi_A \end{cases}$$

therefore, the inequation

$$\max\{\pi_A, \pi_B + \iota_A, \pi_S + \iota_A + \iota_B\} \leq \max\{\pi_A, \pi_S + \iota_A, \pi_B + \iota_A + \iota_S\}$$

boils down to $\pi_A \leq \max\{\pi_A, \pi_S + \iota_A, \pi_B + \iota_A + \iota_S\}$ which is obviously true and implies $(A;B;S) \preceq_p (A;S;B)$.

($\Leftarrow$) We show that if the condition does not hold, *i.e.*, $0 < \iota_B \ \vee\ \pi_A - \iota_A < \pi_B$, then there exists a schedule node S that must be executed between A and B to minimize the peak, *i.e.*,

$$\exists S, (A;S;B) \prec_p (A;B;S) \ \ \textit{i.e.},\ \max\{\pi_A, \pi_S + \iota_A, \pi_B + \iota_A + \iota_S\} < \max\{\pi_A, \pi_B + \iota_A, \pi_S + \iota_A + \iota_B\}$$

Let S be a node of the schedule graph s.t. $(\pi_S = \pi_A - \iota_A) \ \wedge\ (\iota_S = \pi_S - \pi_B)$. Since the impact of a node is always less than or equal to its positive peak, π_S is a valid and possible peak (*i.e.*, $0 \leq \pi_S$). It follows that $\max\{\pi_A, \pi_S + \iota_A, \pi_B + \iota_A + \iota_S\} = \pi_A$. Now, either $0 < \iota_B$ and we have $\pi_A < \pi_A + \iota_B = \pi_S + \iota_A + \iota_B$, or $\pi_A < \pi_B + \iota_A$. In both cases, it implies $\pi_A < \max\{\pi_A, \pi_B + \iota_A, \pi_S + \iota_A + \iota_B\}$, hence $\max\{\pi_A, \pi_S + \iota_A, \pi_B + \iota_A + \iota_S\} < \max\{\pi_A, \pi_B + \iota_A, \pi_S + \iota_A + \iota_B\}$, *i.e.*, $(A;S;B) \prec_p (A;B;S)$. □

Property 3 (Rule (S_1)). $\forall S,\ \iota_A \leq 0 \ \wedge\ \pi_A \leq \pi_B \ \Leftrightarrow\ (A;B;S) \preceq_p (B;S;A)$

Proof.

($\Rightarrow$) $\iota_A \leq 0 \ \wedge\ \pi_A \leq \pi_B$ is a sufficient condition. Indeed, with $\iota_A \leq 0 \Leftrightarrow \pi_B + \iota_A \leq \pi_B$ and $\pi_A \leq \pi_B$, the inequation

$$\max\{\pi_A, \pi_B + \iota_A, \pi_S + \iota_A + \iota_B\} \leq \max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_B + \iota_S\}$$

boils down to $\pi_S + \iota_A + \iota_B \leq \max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_B + \iota_S\}$. Moreover, since $\iota_A \leq 0$, we have $\pi_S + \iota_A + \iota_B \leq \pi_S + \iota_B$, which implies $(A;B;S) \preceq_p (B;S;A)$.

($\Leftarrow$) We show that, if the condition does not hold, *i.e.*, $0 < \iota_A \vee \pi_B < \pi_A$, then there exists a schedule node S that must be executed between A and B to minimize the peak, *i.e.*,

$$\exists S, (B;S;A) \prec_p (A;B;S) \ \ \textit{i.e.},\ \max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_B + \iota_S\} < \max\{\pi_A, \pi_B + \iota_A, \pi_S + \iota_A + \iota_B\}$$

Let S be a node of the schedule graph s.t. $(\pi_S = \pi_B - \iota_B) \ \wedge\ (\iota_S = \pi_S - \pi_A)$. It follows that $\max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_B + \iota_S\} = \pi_B$. Moreover, either $0 < \iota_A$ and we have $\pi_B < \pi_B + \iota_A$, or $\pi_B < \pi_A$. In both cases, it implies $\pi_B < \max\{\pi_A, \pi_B + \iota_A, \pi_S + \iota_A + \iota_B\}$, hence $\max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_B + \iota_S\} < \max\{\pi_A, \pi_B + \iota_A, \pi_S + \iota_A + \iota_B\}$ *i.e.*, $(B;S;A) \prec_p (A;B;S)$. □

Property 4 (Rule (S_2)). $\forall S,\ 0 \le \iota_B \ \wedge\ \pi_B - \iota_B \le \pi_A - \iota_A \ \Leftrightarrow\ (S;A;B) \preceq_p (B;S;A)$

Proof.

($\Rightarrow$) $0 \le \iota_B \ \wedge\ \pi_B - \iota_B \le \pi_A - \iota_A$ is a sufficient condition. Indeed, with $0 \le \iota_B \Rightarrow \pi_S \le \pi_S + \iota_B$ and $\pi_B - \iota_B \le \pi_A - \iota_A \Rightarrow \pi_B + \iota_S + \iota_A \le \pi_A + \iota_S + \iota_B$ the inequation

$$\max\{\pi_S, \pi_A + \iota_S, \pi_B + \iota_S + \iota_A\} \le \max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_B + \iota_S\}$$

boils down to $\pi_A + \iota_S \le \max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_S + \iota_B\}$. Since $0 \le \iota_B$, we have $\pi_A + \iota_S \le \pi_A + \iota_B + \iota_S$, which implies $(S;A;B) \preceq_p (B;S;A)$.

($\Leftarrow$) We show that if the condition does not hold, *i.e.*, $\iota_B < 0 \ \vee\ \pi_A + \iota_B < \pi_B + \iota_A$, then there exists a schedule node S which must be executed between A and B to minimize the peak, *i.e.*,

$$\exists S, (B;S;A) \prec_p (S;A;B) \ \ \textit{i.e.},\ \max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_B + \iota_S\} < \max\{\pi_S, \pi_A + \iota_S, \pi_B + \iota_S + \iota_A\}$$

Let S be a node of the schedule graph s.t. $(\pi_S = \pi_B - \iota_B) \ \wedge\ (\iota_S = \pi_S - \pi_A)$. Then, $\max\{\pi_B, \pi_S + \iota_B, \pi_A + \iota_B + \iota_S\} = \pi_B$. Moreover, either $\iota_B < 0$ and $\pi_B < \pi_S = \pi_B - \iota_B$, or $\pi_A + \iota_B < \pi_B + \iota_A$ and $\pi_B < \pi_B + \iota_A + \pi_B - \iota_B - \pi_A = \pi_B + \iota_S + \iota_A$. In both cases, it implies $\pi_B < \max\{\pi_S, \pi_A + \iota_S, \pi_B + \iota_S + \iota_A\}$, hence $(B;S;A) \prec_p (S;A;B)$. □

B PROOFS OF ASSOCIATIVITY OF CLUSTERING

Recall that the conditions $C_1(A,B)$ and $C_2(A,B)$ means that the two nodes A and B satisfy the topological and arithmetic conditions of the (C_1) and (C_2) rules respectively. Recall also that $[A;B]$ denotes the node made of the clustering of nodes A and B.

Property 5 (a). $C_1(A,B) \wedge C_1(B,C) \Rightarrow C_1([A;B],C) \wedge C_1(A,[B;C])$

Proof. First note that since $\mathrm{Succ}(A) = \{B\} \wedge \mathrm{Succ}(B) = \{C\}$ the topological conditions of (C_1) after the two clusterings hold *i.e.*, $\mathrm{Succ}([A;B]) = \{C\} \wedge \mathrm{Succ}(A) = \{[B;C]\}$. From the initial arithmetic conditions

$$C_1(A,B) \Rightarrow (0 \le \iota_A) \wedge (\pi_A \le \pi_B + \iota_A) \quad \wedge \quad C_1(B,C) \Rightarrow (0 \le \iota_B) \wedge (\pi_B \le \pi_C + \iota_B)$$

$$\begin{array}{lllllllll} \text{we have} & \pi_{[A;B]} & = & \max(\pi_A, \pi_B + \iota_A) & = & \pi_B + \iota_A & \wedge & \iota_{[A;B]} & = & \iota_A + \iota_B \\ \text{and} & \pi_{[B;C]} & = & \max(\pi_B, \pi_C + \iota_B) & = & \pi_C + \iota_B & \wedge & \iota_{[B;C]} & = & \iota_B + \iota_C \end{array}$$

It is easy to check that $0 \le \iota_{A;B}$ and $\pi_{[A;B]} = \pi_B + \iota_A \le \pi_C + \iota_{[A;B]}$, which implies $C_1([A;B],C)$. Similarly, we have $0 \le \iota_{[B;C]}$ and $\pi_A \le \pi_{[B;C]} + \iota_A = \pi_C + \iota_B + \iota_A$, which implies $C_1(A,[B;C])$. □

Property 5 (b). $C_2(A,B) \wedge C_2(B,C) \Rightarrow C_2([A;B],C) \wedge C_2(A,[B;C])$

Proof. First note that since $\mathrm{Pred}(B) = \{A\} \wedge \mathrm{Pred}(C) = \{B\}$ the topological conditions of (C_2) after the two clusterings hold *i.e.*, $\mathrm{Pred}(C) = \{[A;B]\} \wedge \mathrm{Pred}([B;C]) = \{A\}$. From the initial arithmetic conditions

$$C_2(A,B) \Rightarrow (\iota_B \le 0) \wedge (\pi_B + \iota_A \le \pi_A) \quad \wedge \quad C_2(B,C) \Rightarrow (\iota_C \le 0) \wedge (\pi_C + \iota_B \le \pi_B)$$

$$\begin{array}{lllllllll} \text{we have} & \pi_{[A;B]} & = & \max(\pi_A, \pi_B + \iota_A) & = & \pi_A & \wedge & \iota_{[A;B]} & = & \iota_A + \iota_B \\ \text{and} & \pi_{[B;C]} & = & \max(\pi_B, \pi_C + \iota_B) & = & \pi_B & \wedge & \iota_{[B;C]} & = & \iota_B + \iota_C \end{array}$$

It is easy to check that $\iota_C \le 0$ and $\pi_C \le \pi_B - \iota_B \le \pi_A - \iota_A - \iota_B = \pi_{[A;B]} - \iota_{[A;B]}$, which implies $C_2([A;B],C)$. Similarly, we have $\iota_{[B;C]} \le 0$ and $\pi_{[B;C]} + \iota_A = \pi_B + \iota_A \le \pi_A$, which implies $C_1(A,[B;C])$. □

Property 5 (c). $C_1(A,B) \wedge C_2(B,C) \Rightarrow C_2([A;B],C) \wedge C_1(A,[B;C])$

Proof. First note that since $\text{Succ}(A) = \{B\} \wedge \text{Pred}(C) = \{B\}$ the topological conditions of (C_1) and (C_2) after the two clusterings hold *i.e.*, $\text{Succ}(A) = \{[B;C]\} \wedge \text{Pred}(C) = \{[A;B]\}$. From the initial arithmetic conditions

$$C_1(A,B) \Rightarrow (0 \le \iota_A) \wedge (\pi_A - \iota_A \le \pi_B) \quad \wedge \quad C_2(B,C) \Rightarrow (\iota_C \le 0) \wedge (\pi_C \le \pi_B - \iota_B)$$

$$\begin{array}{lllllllll} \text{we have} & \pi_{[A;B]} & = & \max(\pi_A, \pi_B + \iota_A) & = & \pi_B + \iota_A & \wedge & \iota_{[A;B]} & = & \iota_A + \iota_B \\ \text{and} & \pi_{[B;C]} & = & \max(\pi_B, \pi_C + \iota_B) & = & \pi_B & \wedge & \iota_{[B;C]} & = & \iota_B + \iota_C \end{array}$$

It is easy to check that $\iota_C \le 0$ and $\pi_C \le \pi_B - \iota_B = \pi_{[A;B]} - \iota_{[A;B]}$ which implies $C_2([A;B],C)$. Similarly, $0 \le \iota_A$ and $\pi_A \le \pi_{[B;C]} + \iota_A = \pi_B + \iota_A$ implies $C_1(A,[B;C])$. □

Property. 6. Let A, B and C three nodes of a schedule graph then if $C_2(A,B)$ and $C_1(B,C)$ the two possible clusterings both yield the graph

$$X^{\binom{\pi_A}{\iota_A}}; Y^{\binom{\pi_C}{\iota_C}} \text{ with either } X = [A;B] \text{ and } Y = C \text{ or } X = A \text{ and } Y = [B;C]$$

Proof. The two graphs are isomorphic as it is easy to check that in both cases

$$\begin{array}{lll} & \text{Pred}(X) = \text{Pred}(A) & \wedge \quad \text{Succ}(X) = \text{Succ}(A) \cup \{Y\} \\ \wedge & \text{Pred}(Y) = \text{Pred}(C) \cup \{X\} & \wedge \quad \text{Succ}(Y) = \text{Succ}(C) \end{array}$$

From the initial arithmetic conditions

$$C_2(A,B) \Rightarrow (\iota_B \le 0) \wedge (\pi_B + \iota_A \le \pi_A) \quad \wedge \quad C_1(B,C) \Rightarrow (0 \le \iota_B) \wedge (\pi_B \le \pi_C + \iota_B)$$

$$\begin{array}{lllllllllll} \text{we have} & \pi_{[A;B]} & = & \max(\pi_A, \pi_B + \iota_A) & = & \pi_A & \wedge & \iota_{[A;B]} & = & \iota_A & \wedge \quad \iota_B = 0 \\ \text{and} & \pi_{[B;C]} & = & \max(\pi_B, \pi_C + \iota_B) & = & \pi_C & \wedge & \iota_{[B;C]} & = & \iota_C & \end{array}$$

Therefore, $\pi_X = \pi_{[A;B]} = \pi_A \wedge \iota_X = \iota_{[A;B]} = \iota_A \wedge \pi_Y = \pi_{[B;C]} = \pi_C \wedge \iota_Y = \iota_{[B;C]} = \iota_C$ □

C PROOF OF OPTIMALITY OF THE INDEPENDENT TASKS SCHEDULE

Property. 10 The schedule $N_1;\ldots;N_n;P_1;\ldots;P_m$ minimizes the memory peak of any set $\mathcal{X} = \{N_1,\ldots,N_n,P_1,\ldots,P_m\}$ of independent tasks with

$$\iota_{N_i} \le 0 \ \wedge \ \pi_{N_i} \le \pi_{N_{i+1}} \text{ for } i = 1..n \quad \wedge \quad 0 \le \iota_{P_i} \ \wedge \ \pi_{P_{i+1}} - \iota_{P_{i+1}} \le \pi_{P_i} - \iota_{P_i} \text{ for } i = 1..m$$

Proof. The graph depicted in Fig. 7 represents $n+m$ independent tasks in parallel. It can be shown by induction that the sequentialization Rules (S_1) and (S_2) transform it into the above schedule (surrounded by S and T).

∘ Base case. Two nodes are sequentialized into a 2-node chain having the above properties:

- two negatives nodes in parallel are sequentialized using Rule (S_2) into $N \to N'$ with $\pi_N \le \pi_{N'}$;
- two positive nodes in parallel are sequentialized using Rule (S_1) into $P \to P'$ with $\pi_{P'} - \iota_{P'} \le \pi_P - \iota_P$;
- a negative node N and a positive node P in parallel are sequentialized into $N \to P$ by Rule (S_2) if $\pi_N \le \pi_P$ or by Rule (S_1) if $\pi_P < \pi_N$. Indeed, ι_N being negative and ι_P positive, we have $\pi_P - \iota_P < \pi_N - \iota_N$ which is the condition to apply Rule (S_1).

∘ Induction.

Consider a chain of the form $N_1 \to \ldots \to N_j \to P_1 \to \ldots \to P_k$
with $\iota_{N_i} \le 0 \wedge \pi_{N_i} \le \pi_{N_{i+1}} \wedge 0 \le \iota_{P_i} \wedge \pi_{P_{i+1}} - \iota_{P_{i+1}} \le \pi_{P_i} - \iota_{P_i}$
in parallel (via the source and sink nodes S and T) with a task X.

Let X be a negative node ($ip_X \le 0$), then by applying sequentialization rules iteratively

- either X is inserted before the first node N_i such that $\pi_X < \pi_{N_i}$ or after after the last negative node N_j if $\pi_{N_j} \le \pi_X$;

- or X is in parallel with a positive chain $P_1 \to \ldots \to P_k$. If $\pi_X \leq \pi_{P_1} X$ is sequentialized before P_1 by Rule (S_2). Otherwise, $\pi_{P_1} < \pi_X$ and by induction hypothesis $\pi_{P_k} - \iota_{P_k} \leq \ldots \leq \pi_{P_1} - \iota_{P_1}$ and ι_X being negative and ι_{P_1} positive, we have $\pi_{P_1} - \iota_{P_1} < \pi_X - \iota_X$. These inequalities met the conditions to apply Rule (S_1) k times which will insert X before P_1.

The resulting chain has the desired properties (increasing peaks followed by decreasing drops). The same reasoning applies when inserting a positive node X. □

D PROOFS RELATED TO THE COMPOSITIONAL ANALYSIS

Property. 16 Let S and S' be two sorted chains, let X be an arbitrary node and let $S\|X$ (resp. $S'\|X$) denote the sequentialization of X into S (resp. S') according to Rules (S_1) and (S_2), then

$$S \preceq_s S' \Rightarrow S\|X \preceq_p S'\|X$$

Proof. Let $S = N_1 \to \cdots \to N_n \to P_1 \to \cdots \to P_m$ and $S' = N'_1 \to \cdots \to N'_p \to P'_1 \to \cdots \to P'_q$.

∘ Case X is a negative node ($\iota_X \leq 0$). As shown in the proof of Prop. 10, sequentialization will insert

- X in S between the nodes N_i and N_{i+1} such that $\pi_X \in [\pi_{N_i}, \pi_{N_{i+1}})$, and
- X in S' between the nodes N'_j and N'_{j+1} such that $\pi_X \in [\pi_{N'_j}, \pi_{N'_{j+1}})$

with, by convention, $\pi_{N_0} = \pi_{N'_0} = 0$ and $\pi_{N_{n+1}} = \pi_{N'_{p+1}} = +\infty$.

Let Π_S and $\Pi_{S'}$ denote respectively the global peaks of the two sorted chains S and S'. Sorted chains have strictly increasing relative peaks (see Sec. 5.2) and therefore the new global peak of $S\|X$ is located either at N_i (the left of X), at X, or where it was in S (*i.e.*, at N_n or P_1). In mathematical terms:

$$\Pi_{S\|X} = \max(\pi_{N_i} + \Upsilon^S_{N_{i-1}}, \pi_X + \Upsilon^S_{N_i}, \Pi_S + \iota_X)$$

and similarly for $S'\|X$:

$$\Pi_{S'\|X} = \max(\pi_{N'_i} + \Upsilon^{S'}_{N'_{j-1}}, \pi_X + \Upsilon^{S'}_{N'_j}, \Pi_{S'} + \iota_X)$$

Since $S \preceq_s S'$ implies that $\Pi_S \leq \Pi_{S'}$ and $\hat{f}^-_S(\pi_X) \leq \hat{f}^-_S(\pi_X)$, that is,

$$\max(\pi_{N_i} + \Upsilon^S_{N_{i-1}}, \pi_X + \Upsilon^S_{N_i}) \leq \max(\pi_{N'_i} + \Upsilon^{S'}_{N'_{j-1}}, \pi_X + \Upsilon^{S'}_{N'_j})$$

it follows that $\Pi_{S\|X} \leq \Pi_{S'\|X}$.

∘ Case X is a positive node ($0 \leq \iota_X$). The positive chain and functions $\hat{f}^+_S$ are symmetric of the previous case and the same reasoning applies. □

We now make use of a *dominance relation* between sequences of numbers, written $\trianglelefteq$ and introduced in [12]. The dominance relation is reflexive and transitive. It admits *two equivalent definitions*, the first one expressed in terms of pointer advancements over the two sequences to be compared.

Definition 20 (Dominance (Pointer Advancement) [12]). *Let A and B denote two arbitrary sequences of numbers. Sequence A dominates sequence B (denoted $A \trianglelefteq B$) if and only if there exists a sequence of pairs of indexes $(a_1, b_1), \ldots, (a_{|A|+|B|-1}, b_{|A|+|B|-1})$ such that:*

- $(a_1, b_1) = (1, 1)$;
- $(a_{|A|+|B|-1}, b_{|A|+|B|-1}) = (|A|, |B|)$;
- *For each $i \geq 2$, either (i) $a_i = a_{i-1} + 1$ and $b_i = b_{i-1}$, or (ii) $a_i = a_{i-1}$ and $b_i = b_{i-1} + 1$;*
- *For each $i \geq 1$, $A[a_i] \leq B[b_i]$.*

The second equivalent definition, referred to as algebraic, compares the two sequences of numbers in terms of left and right maximum and minimum.

Definition 21 (Dominance (Algebraic) [12]). *Let A and B denote two arbitrary sequences of numbers. Sequence A dominates sequence B (denoted $A \trianglelefteq B$) if and only if for all $i \in [1, |A|]$, there exists $j \in [1, |B|]$ such that:*

$$\begin{array}{ll} \text{(D1)} & \max_{i' \le i} A[i'] \le \max_{j' \le j} B[j'] \\ \text{(D2)} & \min_{i' \le i} A[i'] \le \min_{j' \le j} B[j'] \\ \text{(D3)} & \max_{i' \ge i} A[i'] \le \max_{j' \ge j} B[j'] \\ \text{(D4)} & \min_{i' \ge i} A[i'] \le \min_{j' \ge j} B[j'] \end{array}$$

The dominance relation $\trianglelefteq$ is used to compare *memory profiles*, that is, sequences of integers representing the successive amounts of live memory during a schedule. The memory profile of a schedule $S = X_1; \ldots; X_n$, denoted P_S, is the sequence of the relative peaks and valleys it produces, starting from 0:

$$P_S = \left[0, \pi_{X_1}, \Upsilon^S_{X_1}, \pi_{X_2} + \Upsilon^S_{X_1}, \Upsilon^S_{X_2} \ldots, \pi_{X_n} + \Upsilon^S_{X_{n-1}}, \Upsilon^S_{X_n}\right] \tag{7}$$

If S has length n, then P_S has length $2n + 1$. Because each node is made of a positive peak and a lower impact, the memory profile alternates between greater and lower elements:

$$P_S[2i] = \pi_{X_i} + \Upsilon^S_{X_{i-1}} \ge P_S[2i+1] = \Upsilon^S_{X_i} \le P_S[2(i+2)] = \pi_{X_{i+1}} + \Upsilon^S_{X_i}$$

The $\trianglelefteq$ relation is overloaded to directly compare schedules. Concretely, this means that we may write $S \trianglelefteq S'$ instead of $P_S \trianglelefteq P_{S'}$. We now prove two key properties on schedules involving the dominance relation.

Property 18 Let S be a schedule and $\overline{S}$ its sorted version. Then $S \trianglelefteq \overline{S} \trianglelefteq S$.

Proof. The proof is by induction on the number of clustering steps from S to $\overline{S}$. We start by proving the double inequality after one clustering step. Let S a schedule and S' the schedule after *one* clustering:

$$S = X_1; \ldots, X_i; A; B; Y_1; \ldots, Y_j \quad \text{and} \quad S' = X_1; \ldots, X_i; [A; B]; Y_1; \ldots, Y_j$$

with $|S| = i + j + 2$ and $|S| = i + j + 1$.

We show that $S \trianglelefteq S' \trianglelefteq S$ using the pointer advancement version of the dominance relation (Def. 20).

◦ When $C_1(A, B)$ the two following sequences of pairs of indexes show that $S \trianglelefteq S'$ and $S' \trianglelefteq S$.

$$\begin{aligned}[(1,1), \ldots, (2i+1, 2i+1), (2i+1, 2i+2), (2i+2, 2i+2), (2i+3, 2i+2), (2i+4, 2i+2),\\ (2i+5, 2i+2), (2i+5, 2i+3), \ldots, (2i+2j+5, 2i+2j+3)]\end{aligned} \tag{8}$$

and

$$\begin{aligned}[(1,1), \ldots, (2i+1, 2i+1), (2i+1, 2i+2), (2i+1, 2i+3), (2i+1, 2i+4), (2i+2, 2i+4),\\ (2i+3, 2i+4), (2i+3, 2i+5), \ldots, (2i+2j+3, 2i+2j+5)]\end{aligned} \tag{9}$$

The prefixes (from $(1,1)$ to $(2i+1, 2i+1)$) and suffixes (from $(2i+5, 2i+3)$ or $(2i+3, 2i+5)$ to the end) of the two sequences correspond to identical subschedules and are easily built since $\trianglelefteq$ is reflexive.

The interesting part is the comparison of $A; B$ and $[A; B]$, starting at $(2i+1, 2i+1)$ and ending at $(2i+5, 2i+3)$ for the first sequence (Eq. (8)), and at $(2i+3, 2i+5)$ for the second sequence (Eq. (9)).

If we denote by m the memory level reached after X_i, then we have:

$$\begin{aligned} m &= P_S[2i+1] & P_{S'}[2i+1] &= m \\ \pi_A + m &= P_S[2i+2] & P_{S'}[2i+1] &= m + \pi_{[A;B]} \\ \iota_A + m &= P_S[2i+3] & P_{S'}[2i+3] &= m + \iota_A + \iota_B \\ \pi_B + \iota_A + m &= P_S[2i+4] & & \\ \iota_B + \iota_A + m &= P_S[2i+5] & & \end{aligned}$$

Using the facts that $C_1(A,B)$ entails $0 \le \iota_A$, $\pi_A \le \pi_B + \iota_A$ and $\pi_{[A;B]} = \pi_B + \iota_A$ and of course $\iota_X \le \pi_X$ for all X, the two above sequences satisfy the criteria of Def. 20. The sequence Eq. (8) proves $P_S \trianglelefteq P_{S'}$ and Eq. (9) proves $P_{S'} \trianglelefteq P_S$.

∘ When $C_2(A,B)$ the two sequences proving that $S \trianglelefteq S'$ and $S' \trianglelefteq S$ are:

$$[(1,1),\dots,(2i+1,2i+1),(2i+1,2i+2),(2i+2,2i+2),(2i+3,2i+2),(2i+4,2i+2),\\ (2i+5,2i+2),(2i+5,2i+3),\dots,(2i+2j+5,2i+2j+3)] \quad (10)$$

and

$$[(1,1),\dots,(2i+1,2i+1),(2i+1,2i+2),(2i+2,2i+2),(2i+3,2i+2),(2i+3,2i+3),\\ (2i+3,2i+4),(2i+3,2i+5),\dots,(2i+2j+3,2i+2j+5)] \quad (11)$$

The proof is similar to the $C_1(A,B)$ case, but this time we rely on the fact that $C_2(A,B)$ entails $\iota_B \le 0$, $\pi_B + \iota_A \le \pi_A$ and $\pi_{[A;B]} = \pi_A$ to prove that the two sequences in Eqs. (10) and (11) imply respectively $S \trianglelefteq S'$ and $S' \trianglelefteq S$.

Showing that $S \trianglelefteq \overline{S} \trianglelefteq S$ ($\overline{S}$ being the sorted version of S) is an easy induction on the number of clustering steps needed to reach $\overline{S}$. We rely on the property just shown for one clustering step and on the reflexivity and transitivity of $\trianglelefteq$. □

Property 19 Let G be a task graph and let S and S' be two sorted schedules of G, then $S \trianglelefteq S' \Rightarrow S \preceq_s S'$.

Proof. Let S and S' be the two following sorted chains:

$$S = N_1 \to \cdots \to N_n \to P_1 \to \cdots \to P_m \quad \text{and} \quad S' = N'_1 \to \cdots \to N'_p \to P'_1 \to \cdots \to P'_q$$

Based on Def. 15, we must prove that $S \trianglelefteq S' \Rightarrow S \preceq_p S'$, and that $S \trianglelefteq S' \Rightarrow \forall x \in \mathbb{N}, \hat{f}^-_S(x) \le \hat{f}^-_{S'}(x) \wedge \hat{f}^+_S(x) \le \hat{f}^+_{S'}(x)$. To do this, we rely on the algebraic version of the dominance relation (Def. 21).

Let i be the index on the memory profile P_S corresponding to its highest peak: it is reached either by N_n or by P_1 in S. Since $S \trianglelefteq S'$, we know that there exists an index j for S' such that the four inequalities of Def. 21 hold. Then, $\max_{i' \le i} P_S[i'] \le \max_{j' \le j} P_{S'}[j']$ and $\max_{i' \ge i} P_S[i'] \le \max_{j' \ge j} P_{S'}[j']$ enforce that the maximal peak of S is less than or equal to the maximal peak of S'. Therefore $S \trianglelefteq S' \Rightarrow S \preceq_p S'$.

Considering the same index i as above, we have $\min_{i' \le i} P_S[i'] \le \min_{j' \le j} P_{S'}[j']$ and $\min_{i' \ge i} P_S[i'] \le \min_{j' \ge j} P_{S'}[j']$, which enforce that the lowest valley of S is less than or equal to the lowest valley of S'.

We now prove $S \trianglelefteq S' \Rightarrow \forall x \in \mathbb{N}, \hat{f}^-_S(x) \le \hat{f}^-_{S'}(x)$.

Let x be any integer, let k be such that $\pi_{N_k} \le x < \pi_{N_{k+1}}$, and let ℓ be such that $\pi_{N'_\ell} \le x < \pi_{N'_{\ell+1}}$. With the conventions of Def. 14, k and ℓ exist and are unique. We must prove

$$\hat{f}^-_S(x) = \max(\pi_{N_k} + \Upsilon^S_{N_{k-1}}, x + \Upsilon^S_{N_k}) \le \max(\pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}}, x + \Upsilon^{S'}_{N'_\ell}) = \hat{f}^-_{S'}(x)$$

Intuitively, this case represents the sequentialization of a negative node X with peak x within S and S': X is inserted just after N_k in S whereas it is inserted just after N'_ℓ in S'.

The node N_k is represented on the memory profile of S by $P_S[2k] = \pi_{N_k} + \Upsilon^S_{N_{k-1}}$ and $P_S[2k+1] = \Upsilon^S_{N_k}$. As shown in Sec. 5.2, the properties of sorted chain ensure that $\max_{i' \le 2k+1} P_S[i'] = \pi_{N_k} + \Upsilon^S_{N_{k-1}}$ (the highest relative peak on the left is the next peak at the left of N_k) and that $\min_{i' \le 2k+1} P_S[i'] = \Upsilon^S_{N_k}$ (the lowest valley on the left is the current valley at N_k). In contrast, the highest peak and lowest

valley on the right (*i.e.*, $\max_{i' \geq 2k+1} P_S[i']$ and $\min_{i' \leq 2k+1} P_S[i']$) are the global peak and valley of S, and are the same for all negative nodes.

For each index i on the memory profile of S, there exists an index j on the memory profile of S' ensuring the four min-max inequalities of Def. 21. There are three cases depending on this associated index j.

∘ Case 1: $(i = 2k+1, j = 2\ell+1)$ satisfies Def. 21.

The insertion point after N'_ℓ in S' corresponds to the index $(2\ell+1)$ of the memory profile of S' associated with the index $2k+1$ of S. Therefore $(i = 2k+1, j = 2\ell+1)$ satisfies the four inequalities of Def. 21, in particular (D1) and (D2):

$$\begin{array}{rrcccccl} & \pi_{N_k} + \Upsilon^S_{N_{k-1}} & = & \max_{i' \leq 2k+1} P_S[i'] & \leq & \max_{j' \leq 2\ell+1} P_{S'}[j'] & = & \pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}} \\ \text{and} & \Upsilon^S_{N_k} & = & \min_{i' \leq 2k+1} P_S[i'] & \leq & \min_{j' \leq 2\ell+1} P_{S'}[j'] & = & \Upsilon^{S'}_{N_k} \end{array}$$

As a result, $\max(\pi_{N_k} + \Upsilon^S_{N_{k-1}}, x + \Upsilon^S_{N_k}) \leq \max(\pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}}, x + \Upsilon^{S'}_{N'_\ell})$, hence $\hat{f}^-_S(x) \leq \hat{f}^-_{S'}(x)$.

∘ Case 2: $(i = 2k+1, j > 2\ell+1)$ satisfies Def. 21 and $(i = 2k+1, j = 2\ell+1)$ does not.

Either j lies in the negative part of S' and since, in this part, valleys are decreasing, all valleys from j are lower than the valley of S' at $2\ell+1$. Otherwise, j lies in the positive part of S' and its left lowest valley is the global valley of S'. Therefore, (D2) holds for both pairs (i, j) and $(i, 2\ell+1)$:

$$\Upsilon^S_{N_k} = \min_{i' \leq 2k+1} P_S[i'] \leq \min_{j' \leq j} P_{S'}[j'] \leq \min_{j' \leq 2\ell+1} P_{S'}[j'] = \Upsilon^{S'}_{N'_\ell}$$

Since $2\ell+1$ is on the negative part of S', we also have (D3) $\max_{i' \geq 2k+1} P_S[i'] \leq \max_{j' \geq 2\ell+1} P_{S'}[j']$ (which represent the global peak of S and S' respectively) and (D4) $\min_{i' \geq 2k+1} P_S[i'] \leq \min_{j' \geq 2\ell+1} P_{S'}[j']$ (which represent the global valley of S and S' respectively). But since the pair $(i, 2\ell+1)$ does not satisfy the conditions of Def. 21, it must be the case that (D1) does not hold for $(i, 2\ell+1)$, that is:

$$\pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}} = \max_{j' \leq 2\ell+1} P_{S'}[j'] < \max_{i' \leq 2k+1} P_S[i'] = \pi_{N_k} + \Upsilon^S_{N_{k-1}}$$

Along with $\Upsilon^S_{N_k} \leq \Upsilon^{S'}_{N'_\ell}$, this makes

$$\hat{f}^-_S(x) = \max(\pi_{N_k} + \Upsilon^S_{N_{k-1}}, x + \Upsilon^S_{N_k}) \leq \max(\pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}}, x + \Upsilon^{S'}_{N'_\ell}) = \hat{f}^-_{S'}(x)$$

boil down to

$$\pi_{N_k} + \Upsilon^S_{N_{k-1}} \leq x + \Upsilon^{S'}_{N'_\ell} \tag{12}$$

Consider $i = 2k$: the associated j must be greater than $2\ell+1$ to satisfy (D1) (which does not hold for $2\ell+1$). Moreover, (D2) states that

$$\Upsilon^S_{N_{k-1}} = \min_{i' \leq 2k} P_S[i'] \leq \min_{j' \leq j} P_{S'}[j']$$

and, since valleys are decreasing, we have $\Upsilon^S_{N_{k-1}} \leq \min_{j' \leq j} P_{S'}[j'] \leq \Upsilon^{S'}_{N'_\ell}$.

Along with $\pi_{N_k} \leq x$ this makes inequality (12) hold, which concludes this case.

∘ Case 3: $(i = 2k+1, j < 2\ell+1)$ satisfies Def. 21 and $(i = 2k+1, 2\ell+1)$ does not.

By definition of ℓ, j lies in the negative part of S' and since, in this part, relative peaks are increasing, the memory (being a valley or a peak) at j is smaller than the peak of S' at $2\ell+1$. Therefore, (D1) holds for both pairs (i, j) and $(i, 2\ell+1)$:

$$\pi_{N_k} + \Upsilon^S_{N_{k-1}} = \max_{i' \leq 2k+1} P_S[i'] \leq \max_{j' \leq j} P_{S'}[j'] \leq \max_{j' \leq 2\ell+1} P_{S'}[j'] = \pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}}$$

(D3) and (D4) also hold for the pair $(i, 2\ell+1)$. But since this pair does not satisfy the conditions of Def. 21, it must be the case that (D2) does not hold for $(i, 2\ell+1)$, that is:

$$\Upsilon^{S'}_{N'_\ell} = \min_{j' \le 2\ell+1} P_{S'}[j'] < \min_{i' \le 2k+1} P_S[i'] = \Upsilon^{S}_{N_k}$$

Therefore, the property

$$\hat{f}^{-}_{S}(x) = \max(\pi_{N_k} + \Upsilon^{S}_{N_{k-1}}, x + \Upsilon^{S}_{N_k}) \le \max(\pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}}, x + \Upsilon^{S'}_{N'_\ell}) = \hat{f}^{-}_{S'}(x)$$

boils down to

$$x + \Upsilon^{S}_{N_k} \le \pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}} \tag{13}$$

Consider $i = 2k+2$: the associated j must be smaller than or equal to 2ℓ to satisfy (D2) (which does not hold for $2\ell+1$). Moreover, (D1) states that

$$\pi_{N_{k+1}} + \Upsilon^{S}_{N_k} = \max_{i' \le 2k+2} P_S[i'] \le \max_{j' \le j} P_{S'}[j']$$

and since relative peaks are increasing, we have $\pi_{N_{k+1}} + \Upsilon^{S}_{N_k} \le \max_{j' \le j} P_{S'}[j'] \le \pi_{N'_\ell} + \Upsilon^{S'}_{N'_{\ell-1}}$.

Along with $x < \pi_{N_{k+1}}$, this makes Inequality (13) hold, which concludes this case.

By symmetry, the same reasoning applies on the positive part of the schedule to show that $\forall x, \hat{f}^{+}_{S}(x) \le \hat{f}^{+}_{S'}(x)$.

□