A Fast Heuristic to Pipeline SDF Graphs

Alexandre Honorat, Karol Desnos, Mickaël Dardaillon, Jean-François Nezan

To cite this version:

Alexandre Honorat, Karol Desnos, Mickaël Dardaillon, Jean-François Nezan. A Fast Heuristic to Pipeline SDF Graphs. Embedded Computer Systems: Architectures, Modeling, and Simulation, Jul 2020, Pythagorion, Samos Island, Greece. pp.139-151, 10.1007/978-3-030-60939-9_10 . hal-02993338

HAL Id: hal-02993338

https://hal.archives-ouvertes.fr/hal-02993338

Submitted on 6 Nov 2020

A Fast Heuristic to Pipeline SDF Graphs[*]

Alexandre Honorat[1], Karol Desnos[1], Mickaël Dardaillon[1], and Jean-François Nezan[1]

Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, F-35000 Rennes, France
firstname.lastname@insa-rennes.fr

Abstract. A common optimization of signal and image processing applications is the *pipelining* on multiple Processing Elements (PE) available on multicore or manycore architectures. Pipelining an application often increases the throughput at the price of a higher memory footprint. Evaluating different pipeline configurations to select the best one is time consuming: for some applications, there are billions of different possible pipelines. This paper proposes a fast heuristic to pipeline signal and image processing applications modelled with the Synchronous DataFlow (SDF) Model of Computation (MoC). The heuristic automatically adds pipeline stages in the SDF graph in the form of delays, given the Execution Time (ET) of the actors and the number of PEs. The heuristic decreases the time spent by the developer to pipeline its application from hours to seconds. The efficiency of the approach is illustrated with the optimisation of a set of signal and image processing applications executed on multiple PEs. On average, when adding one pipeline stage, our heuristic selects a stage resulting in a better throughput than 90% of all possible stage emplacements.

Keywords: SDF · pipeline · parallelism · throughput

1 Introduction

Synchronous DataFlow (SDF) [10] is widely used to model signal and image processing applications and to optimize them on multicore embedded systems. Pipelining is made possible in SDF by adding *delays*. Delays represent initial data in buffers, which break data dependencies. This method has already been proved to be efficient on SDF applications [9] but usually requires to add the delays manually in the SDF graph or to call heuristics [7]. Indeed, computing the optimal throughput of an application is a problem of high complexity that also requires computing the scheduling, the mapping and the pipelining.

In this paper we propose a fast heuristic to automatically pipeline an application modelled with an SDF graph. This heuristic is performed before mapping

[*] This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement №732105 (project CERBERO) and from the Région Bretagne (France) under grant ARED 2017 ADAMS. We would like to thank Michael Masin for his help to generate all admissible graph cuts.

and scheduling the application, and is thus suboptimal but fast and scalable. It automatically adds delays in the SDF graph, given the number of Processing Elements (PEs) and application profiling information. The heuristic is parametric: the user can choose the number of pipeline stages that he wants to add. Various experiments demonstrate that our heuristic increases the throughput of the majority of the tested applications. When adding one pipeline, the heuristic finds the solution ensuring optimal throughput for 19 applications out of 24 tested.

The paper is organized as follows. Section 2 presents the main properties of the SDF Model of Computation (MoC). Section 3 specifies the notion of pipeline for SDF graphs and outlines necessary conditions for their validity. The main contribution, automatic pipelining of SDF graphs, is developed in Section 4. Extensive experiments follow in Section 5 with both theoretical evaluation of the throughput gain and real measurements on hardware. The main drawback of pipelining, memory footprint increase, is also quantified. Related work is presented in Section 6. Finally, Section 7 concludes this paper.

2 Background

SDF [10] graphs are directed multi-graphs annotated with data communication *rates*. The vertices are called *actors*, and the edges are called *buffers*. Actors correspond to the processing of the application, while buffers contain the data sent from an actor to another. Data stored in a buffer are called *tokens*. Two rates are specified per buffer b: a production rate $\mathrm{prod}(b) \in \mathbb{N}^*$ on the sending side of b, and a consumption rate $\mathrm{cons}(b) \in \mathbb{N}^*$ on the receiving side. Hence, each time an actor is *fired*, which means its computations are executed, it consumes and produces a fixed number of tokens on each input and output buffers, respectively.

Only *consistent* SDF graphs are considered in this paper. An SDF graph is consistent when there exists a finite number of firings of each actor bringing back all buffers with the exact same number of tokens as initially. Such minimal number of firings is called a *repetition vector* and denoted $\vec{r}$. It is computed from the buffers production and consumption rates.

To avoid deadlocks, cycles of the graph need initial tokens in at least one of their buffers. These initial tokens are called *delays*. The size of a delay on a buffer b, that is the number of initial tokens that b contains, is denoted $d_0(b)$. In this paper, it is assumed that the user already set delays in cycles, ensuring no deadlock. Delays are not restricted to cycles though; when placed correctly and outside cycles, delays cut some data dependencies and create pipeline stages. This point will be discussed and illustrated in the next section.

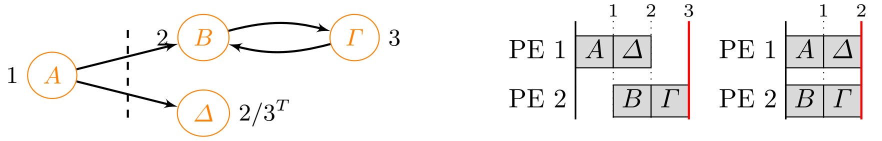
The contribution of this paper is a heuristic for automating the placement of pipeline delays in any SDF graph. Following the semantics introduced in [2], we assume all delays being permanent, which means that their token values are transmitted from one graph iteration to the next.

3 Admissible graph cuts for pipelining

Admissible graph cuts for pipelining correspond to *feed-forward* graph cuts as defined for the design of integrated circuits [14]. A graph cut is a set of edges which, if removed, disconnects the graph in two or more components. In a feed-forward graph cut, all edges of the cut are going in the same direction. Hence, such cut cannot contain an edge from a cycle. The direction of an edge is deduced from the topological ranks of the actors. An example of actor ranks of As Soon As Possible (ASAP) and As Late As Possible (ALAP) topological orderings is presented in Figure 1a. Lowest actor rank 1 correspond to actors without input buffers, and the highest actor rank correspond to actors without output buffers.

A pipeline is created by adding delays on all buffers of a feed-forward graph cut, in order to break the data dependencies. For example, the feed-forward graph cut (dashed line) between topological ranks 1 and 2 of the graph in Figure 1a breaks the data dependencies between actors A and B, and A and $\varDelta$. The pipeline increases the throughput of the graph, as depicted in Figure 1b. In Figure 1a as in all SDF graphs presented in this section, rates are all equal to 1 for simplification. The throughput is defined by the inverse of the *Initiation Interval (II)* duration, that is the duration to periodically execute a graph *iteration*. A graph iteration contains as many firings as in the repetition vector $\vec{r}$. We assume that there is a synchronization barrier at the end of each graph *iteration*. On the left part of Figure 1b, without pipeline, the II duration is 3, whereas on the right part, the II duration is only 2 with the pipeline. Note that, depending on the topological ordering, the graph cuts may not be identical.

To create a pipeline on an SDF graph, the size of a delay on a buffer b must be equal to $d_0(b) = \text{prod}(b) \times \vec{r}[\text{src}(b)] = \text{cons}(b) \times \vec{r}[\text{dst}(b)]$. Thus, dependencies between all firings of producer actor src(b) and receiver actor dst(b) are broken. If multiple feed-forward graph cuts contains the same buffer, the delay sizes are summed. In this paper, the number of pipelines n correspond to n different feed-forward graph cuts, dividing the execution of the application in $n+1$ *stages*.



(a) Graph example annotated with ASAP and ALAP topological orderings. ALAP ordering is specified with ^{T}, only if different from ASAP.

(b) Two schedule examples of graph 1a on two PEs: on the left without pipeline, on the right with one pipeline between ranks 1 and 2.

Fig. 1: Topological ordering and schedule example without and with pipeline.

Unfortunately, the number of admissible graph cuts may be large. An example is given with a commonly used split-join graph topology [17], which is a subcategory of SDF graph. Although the graph represented in Figure 2 only contains 4 parallel paths with 3 buffers each, $3^4 = 81$ cuts are admissible. Indeed, if l paths connect a split actor to a join actor, each path having m buffers, the total number of feed-forward graph cuts is equal to m^l. Because the number of admissible graph cuts may grow exponentially with the number of edges of the graph, exploring them all is not feasible. For this reason, our heuristic algorithm will only explore a subset of the admissible cuts. For example, our heuristic considers at most 3 admissible cuts for the graph in Figure 2. Those admissible cuts are detailed in the next section.

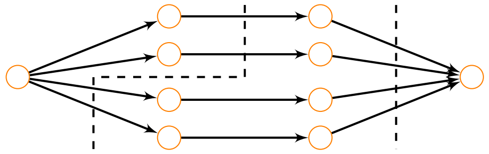


Fig. 2: Split-join graph with four parallel branches. 2 admissible cuts are represented with a dashed line, among 81 possible.

4 Automatic pipelining of SDF graphs

The automatic pipelining heuristic has two main steps: (1) generation of topological graph cuts, (2) selection of topological graph cuts. The first step, described in Section 4.1, computes a subset of admissible cuts. The second step, detailed in Section 4.2, selects a few cuts among the cuts computed in step (1).

4.1 Computing topological graph cuts

The heuristic selects a subset of admissible graph cuts: *topological* graph cuts according to the ASAP and ALAP topological orderings. A topological graph cut of rank c contains all buffers coming from an actor of rank lower than c and going to an actor of rank higher than or equal to c. Such topological cut is admissible if none of its buffers is part of a directed cycle of the graph.

The number of admissible topological graph cuts is upper bounded by the diameter of the graph, that is the number of buffers on the longest path. For example, the graph depicted in Figure 2 admits only 3 topological cuts according to ASAP graph ordering, whereas this graph admits 81 admissible cuts in total. Moreover, in the case of Figure 2, ASAP and ALAP graph orderings are identical so the same 3 graph cuts are considered for both topological orderings.

In order to build the ASAP and ALAP topological orderings, a cycle analysis of the graph is run first: the Johnson's algorithm [6] computes all simple cycles of a directed graph. Johnson's algorithm upper bounds the complexity of the whole heuristic. The buffers being part of cycles are recorded to later filter the admissible cuts. For example, the topological cut of rank 3 in the SDF graph depicted in Figure 1a is invalid since there is a cycle between actors B and Γ.

Note that it is assumed that the user sets enough delays on at least one buffer of any cycle, so that this buffer breaks the data dependency of the cycle. Thanks to this assumption, ASAP and ALAP orderings are computed by a mere breadth first search on the graph, not visiting the buffer breaking each cycle.

The number of admissible topological graph cuts is small and upper bounded by the graph diameter, enabling our heuristic to be fast. The admissible topological graph cuts naturally include all cuts located at sequential bottlenecks of the application, so they are the best candidates to increase the application performances by pipelining. Formally, sequential bottlenecks are located on single paths of the graph: when two successive actors of ranks $c - 1$ and c are the only actors having these ranks. Selecting such cuts particularly benefits the applications having single paths and their repetition vector equal to $\vec{1}$.

4.2 Selecting best topological graph cuts

To select the best topological graph cuts, the presented heuristic relies on a map linking topological ranks to an estimate of the Execution Time (ET) of all their actors. By definition, all actors having the same topological rank can be executed in parallel. We introduce a few notations to formalize the computation of this map. $T(a)$ denotes the Execution Time (ET) of an actor a. The number of firings of a is $\vec{r}[a]$. The rank of a is $\text{rank}(a)$. The number of PEs is #PE. The ET estimate of rank c, denoted rankLoad(c), is computed as follows in Equation (1).

$$\text{rankLoad(c)} = \frac{\sum_{\text{rank}(a)=c} \left\lceil \frac{\vec{r}[a]}{\#\text{PE}} \right\rceil \times T(a)}{\#\{a|\, \text{rank}(a) = c\}} \tag{1}$$

The main purpose of Equation (1) is to provide a metric indicating if cutting before actors of rank c improves the throughput, that is to balance the computation before and after the cut. To do so, we actually compare the estimated ET of all ranks before the cut of rank c, $\sum_{1\leq i<c} \text{rankLoad(i)}$, with the estimated ET of all ranks after the cut of rank c, $\sum_{c\leq i} \text{rankLoad(i)}$. However, it is needed to weight the ranks according to the amount of parallelism that they contain, so that the graph is cut where it matters most: on single paths for example. Thus, Equation (1) contains two divisions in order to reduce the weight of already parallel ranks: the repetition factor is divided by #PE, and the whole sum is divided by the number or actors in the considered rank. The rankLoad is averaged for both ASAP and ALAP topological orderings.

The selection of cuts is parameterized by two integers: the number of cuts wanted by the user x, selected among the number of balanced cuts to consider y (denoted D x, y in Section 5). We always have x lower than or equal to y, and y lower than the highest actor rank. y helps to define a first set of equally distributed topological graph cuts. To do so, the sum of all rankLoad(i) is divided by y, giving an average stage load avgStageLoad. Then we enumerate ASAP cuts by increasing order of their rank, and select the closest ones to a multiple of avgStageLoad. The same operation is performed on ALAP cuts sorted by decreasing order of their rank. At most, $2\times y$ balanced cuts are selected, since cuts selected with ASAP and ALAP orderings may not be identical. Both orderings are considered in order to avoid selecting 0 cuts because of unbalanced ETs.

The last step of the heuristic is to select x cuts among the $2 \times y$ balanced cuts. This is done by two means: removing cuts that are too close from each other, and then selecting the one using less delays. Two topological cuts of rank c_1 and c_2 are considered too close from each other if the sum of their intermediate estimated ET is lower than avgStageLoad, as formalized in Equation (2).

$$\text{avgStageLoad} > \sum_{c_1 \leq i < c_2} \text{rankLoad(i)} \qquad (2)$$

5 Evaluation

The presented heuristic is evaluated on various applications coming from the StreamIt [18] benchmark, the examples provided with the SDF3 [16] tool, and the applications provided with the Preesm [15] tool. These applications represent a panel of basic state of the art signal and image processing algorithms, as well as more complex telecommunications, video coding and computer vision applications. The heuristic results are compared by gain in throughput, relative to the sequential non-pipelined throughput on a single Processing Element (PE).

Three different evaluations are performed. In Section 5.1, the theoretical throughput gain is computed based on the schedule length obtained after adding the pipelines selected by the heuristic. A comparison is made with the optimal throughput gain among all admissible cuts, for applications amenable to an exhaustive exploration, while large applications are detailed in Section 5.2. Finally in Section 5.3, the throughput and memory increases are measured on real executions of applications running on hardware.

All experiments have been run with the open-source Preesm tool (`https://preesm.github.io/`), on a laptop with an Intel processor i7-7820HQ. For all selected applications, the execution time of the proposed heuristic is between 1 and 18 ms (maximum reached for SIFT). Note that the StreamIt/SDF3 applications are all stateless, except h263decoder (noAC) having self-loops. Self-loops disable auto-concurrency of an actor: multiple firings are serialized.

Main characteristics of the applications are presented in the results tables. In the second column, MAP is the Maximum number of Actors in Parallel in the SDF graph; MAP equals the maximum number of parallel paths in the graph. When known, the total number of admissible graph cuts is specified in the column labeled #Cuts. Note that multiple versions of SIFT and sobel-morpho applications are considered: their graph is identical but they do not have the same amount of firings. Some of their actors are fired a number of times equal to a multiple of a parameter p. Only SIFT and stereo contain directed cycles in their SDF graph. In all results tables, the columns labeled by D x, y contain the throughput gain obtained by the heuristic selecting x pipelines among y balanced pipelines. Columns labeled by C x contain the optimal throughput gain, over all admissible cuts, for x pipelines. Lines of results tables without any value printed in bold means that the throughput gain is similar for all setups; otherwise, the value in bold corresponds to the best gain among the line.

5.1 Theoretical throughput gain: regular applications

Theoretical throughput gain obtained with the heuristic is presented in Table 1, for three setups: no pipeline, one pipeline among one, three pipelines among three. Most applications have a repetition vector $\vec{r}$ equal to $\vec{1}$, except Chain4.2noAC (which contains self-loops), cd2dat, h263decoder, modem, mp3decoder, samplerate and satellite. Chain4.2noAC and Chain4.1 are toy examples made to fit the best cases of the heuristic; they correspond to the graph depicted in Figure 2, with only one path instead of four.

In Table 1, the best throughput gain is obtained by the heuristic with 3 cuts (D 3,3) for 11 of the 17 applications. More importantly, the heuristic finds a close to the optimal throughput with 1 cut for all applications except mp3decoder. The number of admissible cuts generating a throughput gain lower than or equal to D 1,1 is reported as a percentage of the total number of admissible cuts, in column %. In average, D 1,1 reaches a better throughput gain than 91% of the admissible cuts. Note that two applications are not compared with the optimal gain, FMRadio and Vocoder, because they admit too many cuts. These applications, and three others, are discussed in section 5.2.

On DCT and h263decoder, the throughput gain is less than 2.0, even with 3 pipelines: this comes from too few actors in the original graphs (respectively 8 and 4), having unbalanced ETs. This configuration leads the heuristic to find only 2 graph cuts for DCT and h263decoder, even if 3 pipelines were asked by the user. The number of effectively selected cuts is specified as an exponent. The same behavior happens for modem and mp3decoder applications: only 2 cuts are selected whereas 3 pipelines were asked. To avoid this problem, only 2 pipelines among 3 are requested for the Preesm applications, see Table 2 Indeed, in these applications the ETs are greatly unbalanced and the ET of the longest actor represents up to 47% of the sequential ET of sobel-morpho (p1).

For the Preesm applications evaluated in table 2, the heuristic reaches the best throughput in 7 cases out of 9. SIFT application is a difficult case: its SDF graph is widely parallel (up to 30 parallel paths) and contains multiple cycles. Moreover, its parallel paths have unbalanced ET. In this situation, selecting topological cuts is not the best option and 1 optimal cut (C 1) even reaches a better throughput than 2 cuts from the heuristic (D 2,3): for SIFT (p1) and SIFT (p2). However, when more balanced parallelism is expressed, for SIFT (p4), the heuristic configuration D 2,3 once again is better than the other setups.

5.2 Theoretical throughput gain: widely parallel applications

This subsection evaluates the applications revealing the main advantage of the presented heuristic: no explosion of the number of cuts to test when the SDF graph is already parallel. Indeed, all evaluated applications in Table 3 admit between 10^5 and 10^{10} cuts, which makes it impossible to evaluate the throughput of

Name	MAP	#Actors	#Cuts	D 0	D 1,1	C 1	%	D 3,3
Chain4.1	1	4	3	1.0	2.0	2.0	100	**3.9**
Chain4.2noAC	1	4	3	1.4	2.3	2.3	100	**3.4**
BitonicSort	4	40	141	1.6	2.9	2.9	100	**3.6**
cd2dat	1	6	5	4.0	4.0	4.0	80	4.0
DCT	1	8	7	1.0	1.8	1.8	100	1.8[2]
DES	3	53	128	1.2	2.2	2.2	96	**2.4**
FFT	1	17	16	1.0	2.0	2.0	100	**3.7**
FMRadio	12	43	—	3.1	**3.3**	—	—	**3.3**
h263decoder (noAC)	1	4	3	1.8	1.8	1.9	100	**2.0**[2]
modem	1	6	5	2.0	**3.3**	3.3	100	3.3[2]
mp3decoder	2	14	33	3.7	3.7	**3.8**	66	3.7[2]
MPEG2noparser	3	23	140	1.1	2.2	2.2	100	**2.7**
samplerate	1	6	5	4.0	4.0	4.0	60	4.0
SAR	2	44	63	1.0	1.8	1.8	100	**2.3**
satellite	3	22	90	4.0	4.0	4.0	68	4.0
TDE	1	29	28	1.0	1.9	1.9	100	**3.4**
Vocoder	17	114	—	1.2	2.1	—	—	**2.6**

Table 1: Characteristics and throughput gain with delays (D) of SDF benchmark applications, on four PEs. D 0 corresponds to no pipeline. D 1,1 corresponds to one pipeline selected among one. C 1 corresponds to the optimal single stage pipeline. % is the percentage of cuts worst than or equal to the heuristic. D 3,3 corresponds to three pipelines selected among three.

Name	MAP	#Actors	#Cuts	D 0	D 1,1	C 1	%	D 2,3
SIFT (p1)	30	54	868	1.2	1.6	**2.2**	92	1.6
SIFT (p2)	30	54	868	2.3	2.8	**3.7**	91	3.0
SIFT (p4)	30	54	868	3.5	3.5	3.6	80	**3.7**
sobel-morpho (p1)	1	6	5	1.0	**2.0**	2.0	100	2.0
sobel-morpho (p2) *	1	6	5	1.7	2.4	2.4	100	**2.6**
sobel-morpho (p3) *	1	6	5	2.3	**3.5**	3.5	100	3.4
sobel-morpho (p4)	1	6	5	2.3	2.8	**3.3**	40	3.3
stereo	3	28	3631	3.3	3.9	**3.9**	99	3.9
lane-detection *	3	11	24	1.0	1.7	1.7	100	**2.5**

Table 2: Throughput gain with delays (D) of SDF benchmark applications, on four PEs. D 0 corresponds to no pipeline. D 1,1 corresponds to one pipeline selected among one. C 1 corresponds to the optimal single stage pipeline. % is the percentage of cuts worst than or equal to the heuristic. D 2,3 corresponds to two pipelines selected among three possibilities.

each cut by performing scheduling and mapping. Moreover, the number of possibilities also explodes with the number of pipelines asked: it is equal to the number of cut combinations without repetition (binomial coefficient): $\binom{\#Cuts}{\#Stages-1}$.

Table 3 presents results for the applications already having parallelism expressed in their graph: MAP is between 12 and 17 for all of them. In this experiment, the throughput is evaluated on 64 PEs for the heuristic setup D 3,3 selecting 3 pipelines. Having 64 PEs ensures to observe the effect of the pipelines instead of the inherent task parallelism. Indeed, the maximum number of actors in parallel MAP (17) is almost 4 times smaller than the number of PEs. The maximum theoretical throughput gain with unlimited PEs, Max Θ, is given as a reference. All applications in Table 3 are acyclic, so Max Θ is computed by dividing the sequential ET of the application by the ET of its longest actor, as if each buffer had a pipeline delay. Adding 3 pipelines increases the throughput gain from a factor 2 (for FMRadio) to 3 (for ChannelVocoder).

Name	MAP	#Actors	#Cuts	D 0	D 3,3	Max Θ
Beamformer	12	57	1.7×10^{7}	8.9	19.0	25.6
ChannelVocoder	17	55	1.3×10^{10}	11.1	33.2	33.4
Filterbank	16	85	4.3×10^{8}	10.5	30.5	32.2
FMRadio	12	43	2.6×10^{5}	6.0	12.7	13.1
Vocoder	17	114	3.0×10^{10}	1.2	2.7	2.8

Table 3: Throughput gain with delays (D) of SDF benchmark applications, on sixty-four PEs. D 0 corresponds to no pipeline. D 3,3 corresponds to three pipelines selected among three possibilities. Max Θ corresponds to the maximum possible throughput gain, with unlimited PEs.

5.3 Practical experimentation

In this subsection, the throughput and memory measurements come from real executions on hardware, on the same laptop used for all experiments, having 4 PEs. Note that the scheduler used in this practical experimentation differs from the one used in the theoretical experimentation. Both schedulers are a variant of list scheduling [8]. Memory is allocated after the scheduling process, with buffer merging [4] optimizations activated. The memory needed is computed by Preesm, and compared with the sequential version on 1 PE for reference.

Results are provided in Table 4, for an average of 100 executions for SIFT and stereo, and 10000 executions for sobel-morpho and lane-detection. The heuristic especially improves the throughput of SIFT and sobel-morpho with $p = 1$ and $p = 2$, that is, when the application is not parallel enough. Yet, for lane-detection which has $\vec{r} = \vec{1}$, the heuristic only slightly increases the throughput, while increasing the memory by a factor 1.9. The theoretical throughput gain of lane-detection is 2.5, that is two times higher than reality. We explain this gap by the variability of the ET of the display actor, representing 28% of the application sequential execution time. Also, synchronization points added by Preesm may be non-negligible. None of the applications reaches the throughput expected in the theoretical evaluation.

An interesting point is that selecting 1 cut among 2 (D 1,2) gives better results than 1 among 1 for half of the cases. Such heuristic setups may compensate the case of unbalanced ETs or cycles, especially for SIFT (p2). Moreover for SIFT (p2) the D 1,2 setup greatly reduces the memory footprint compared to D 1,1: from a factor 3.0 to 1.1. Finally, the heuristic offers a trade-off between memory footprint and throughput. This trade-off is especially needed for memory bounded application as SIFT requiring 197 MBytes (reference). At worst, for sobel-morpho (p4), adding one pipeline decreases the throughput while greatly increasing the memory (3.3 times). The memory increase is due to the graph cut location: between buffers transmitting numerous data, and thus it causes additional time for memory copies and synchronizations.

Name	D 0		D 1,1		D 1,2		D 2,3	
	Sp.	Mem.	Sp.	Mem.	Sp.	Mem.	Sp.	Mem.
SIFT (p1)	1.2	1.1	**1.6**	2.1	1.4	1.3	1.3	1.8
SIFT (p2)	1.8	1.1	1.9	3.0	**2.4**	1.2	2.2	2.3
SIFT (p4)	**2.5**	1.2	2.2	1.1	2.2	1.1	2.5	1.8
sobel-morpho (p1)	0.9	1.0	1.3	2.2	1.3	2.6	**1.6**	3.8
sobel-morpho (p2)	1.7	1.6	2.3	2.1	**2.5**	2.4	2.1	3.4
sobel-morpho (p3) *	2.3	2.1	2.4	2.8	**2.5**	2.6	2.5	3.2
sobel-morpho (p4)	**2.5**	2.0	1.9	2.6	2.2	2.3	2.4	3.3
stereo	2.2	1.1	2.3	1.1	2.3	1.1	**2.4**	1.1
lane-detection *	1.0	1.0	1.1	1.8	1.1	1.7	**1.2**	1.9

Table 4: Throughput and memory increases with delays (D), on four PEs, for different parallelism parameters (p). Specific mapping constraints are enforced for applications marked with *: read and display actors are alone on their core if there is a pipeline.

6 Related work

Pipelining and more generally retiming has been extensively studied in the context of VLSI circuit design [11,14]. Pipelining legality was formally defined by Parhi [14] for a subset of SDF graphs: Homogeneous Synchronous DataFlow (HSDF) graphs, which always have their repetition vector equal to $\vec{1}$. It was also studied for software pipelining [1], with retiming methods used in this context [3]. Our work focus on pipelining SDF graphs, avoiding the costly conversion to HSDF and thus reducing the analysis complexity.

Pipelining of SDF graphs was originally proposed by Lee [9] as an optimization. Gordon et al. [5] proposed an heuristic to pipeline a partially unfolded SDF graph, as well as Kudlur et al. [7]. The heuristic presented by Kudlur et al. requires the Initiation Interval (II) length as an input of their algorithm; on the contrary, our heuristic requires a maximum number of pipelines as an input, and tries to minimize the II accordingly. Also, the heuristic presented by Gordon et al. relies on a first transformation of the original actors, in order to balance the ETs and to adapt the amount of parallelism.

Multiple works [21,12] addressed the optimal finding of a retiming to reduce the makespan of a graph. Additionally, [21] accepts a constraint on the maximum number of processors, at the cost of non-optimality. Both use symbolic execution of a partially unfolded SDF graph to find a retiming. In this paper we focus on the pipelining of an SDF graph in its reduced original form to provide a fast heuristic. We do not perform any execution, symbolic or not.

Scheduling has been largely explored in optimal and heuristic forms [8,13]. A few works look at combining pipelining with scheduling, restricted to HSDF graphs [19] or acyclic SDF graphs [20]. Our work separates pipelining from scheduling. Scheduling is computed afterwards on the pipelined graph, taking advantage of original data and task parallelism, as well as temporal parallelism.

7 Conclusion

A fast heuristic to automatically pipeline SDF applications at coarse grain has been presented and actually improves the throughput of the evaluated applications. The heuristic is able to quickly pipeline applications containing up to billions of admissible cuts. Our algorithm limits its exploration to a few cuts to reduce analysis time, and experiments show this method is close to the optimal solution. The last experiment have shown a gap between the theoretical throughput gain and the practical gain, always lower than expected. This gap is observed for both our heuristic and the theoretical optimal solution.

The presented heuristic is especially useful when considering a large amount of PEs. However, our method can still be improved for complex applications, especially if containing cycles, for examples by adding smaller delays to break the dependencies between only a certain amount of firings instead of all.

Finally, this heuristic is only one optimization method among various others, as the most related to this work: retiming. Combination of our pipelining heuristic and classic retiming techniques is kept for future work.

References

1. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Comput. Surv. **27**(3), 367–432 (1995)
2. Arrestier, F., Desnos, K., Pelcat, M., Heulot, J., Juarez, E., Menard, D.: Delays and States in Dataflow Models of Computation. In: SAMOS XVIII (2018)

3. Calland, P.Y., Darte, A., Robert, Y.: Circuit retiming applied to decomposed software pipelining. IEEE Trans. Parallel and Distributed Systems **9**(1), 24–35 (1998)
4. Desnos, K., Pelcat, M., Nezan, J.F., Aridhi, S.: On Memory Reuse Between Inputs and Outputs of Dataflow Actors. ACM Trans. Embedded Computing Systems (TECS) **15**(2), 30 (2016)
5. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. SIGPLAN Not. **41**(11), 151–162 (2006)
6. Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM Journal on Computing **4**(1), 77–84 (1975)
7. Kudlur, M., Mahlke, S.: Orchestrating the execution of stream programs on multicore platforms. ACM SIGPLAN Notices **43**(6), 114–124 (2008)
8. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. **31**(4), 406–471 (1999)
9. Lee, E., Messerschmitt, D.: Pipeline interleaved programmable dsp's: Synchronous data flow programming. IEEE Trans. Acoustics, Speech, and Signal Processing **35**(9), 1334–1345 (1987)
10. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
11. Leiserson, C.E., Saxe, J.B.: Retiming synchronous circuitry. Algorithmica **6**(1), 5–35 (1991)
12. Liveris, N., Lin, C., Wang, J., Zhou, H., Banerjee, P.: Retiming for synchronous data flow graphs. In: 2007 Asia and South Pacific Design Automation Conference. pp. 480–485 (2007)
13. Malik, A., Gregg, D.: Orchestrating Stream Graphs Using Model Checking. ACM Trans. Archit. Code Optim. **10**(3), 19:1–19:25 (2013)
14. Parhi, K.K.: VLSI digital signal processing systems : design and implementation (2007)
15. Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.F., Aridhi, S.: PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming. In: EDERC. p. 36 (2014)
16. Stuijk, S., Geilen, M., Basten, T.: SDF^3: SDF For Free. In: Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings. pp. 276–278 (2006)
17. Tendulkar, P., Poplavko, P., Maler, O.: Symmetry breaking for multi-criteria mapping and scheduling on multicores. In: Braberman, V., Fribourg, L. (eds.) Formal Modeling and Analysis of Timed Systems. pp. 228–242 (2013)
18. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: Proceedings of the 11th International Conference on Compiler Construction. pp. 179–196. CC '02 (2002)
19. Yang, H., Ha, S.: Pipelined data parallel task mapping/scheduling technique for mpsoc. In: 2009 Design, Automation, Test in Europe Conference Exhibition. pp. 69–74 (2009)
20. Yuankai Chen, Hai Zhou: Buffer minimization in pipelined sdf scheduling on multicore platforms. In: 17th Asia and South Pacific Design Automation Conference. pp. 127–132 (2012)
21. Zhu, X., Geilen, M., Basten, T., Stuijk, S.: Multiconstraint static scheduling of synchronous dataflow graphs via retiming and unfolding. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems **35**(6), 905–918 (2016)

Addendum

This addendum is not part of the original accepted paper: it has been added by A. Honorat after publication and has not been reviewed. It focuses on:

- linear equation to find all admissible graph cuts;
- details on ALAP ordering;
- example for selection of balanced cuts.

Equation to find all admissible graph cuts

Thanks to Michael Masin and his team, we were able to formulate the admissible feed-forward cut constraint as the following recursive Equation (3). The equality must be respected for every actor $\alpha \in V$, the set of actors of the SDF graph. It introduces the notion of *actor delay*, denoted $\daleth$. An actor delay corresponds to a shift of data on all its input. The unit of the actor delay function $\daleth$ is the number of pipeline stages: if $\daleth(\alpha) = 2$, there are two pipeline stages until actor α. In the equation, E is the set of edges of the SDF graph.

$$\forall e \in \{b \in E|\, \mathrm{dst}(b) = \alpha\}, \daleth(\alpha) = \daleth(\mathrm{src}(e)) + \frac{d_0(e)}{\mathrm{cons}(e) \times \vec{r}\,[\alpha]} \tag{3}$$

The recursion stops on actors having no incoming buffer, where the actor delay is set to 1 by default: such actors are executed during the first pipeline stage. It is assumed that the user sets enough delays on at least one buffer of any cycle; these buffers are ignored for the computation of $\daleth$, otherwise the formula Equation (3) would imply an indefinite recursion.

As the placement validity is defined recursively from the actors having no outgoing buffers, delays may be distributed over the whole paths going to an actor, and not only on its direct incoming buffers. Figure 3 illustrates this possibility. Removing one of its two delays makes the cut not admissible anymore.

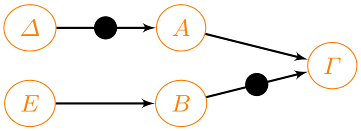
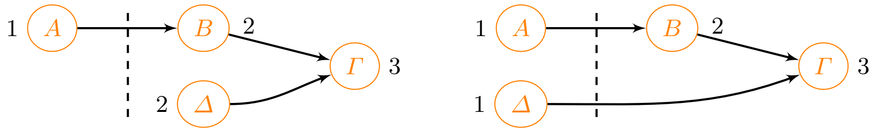


Fig. 3: Graph with valid delay placement distributed on the paths. There are two pipeline stages: $\daleth(\varDelta) = \daleth(E) = \daleth(B) = 1$ and $\daleth(A) = \daleth(\varGamma) = 2$.

Details on ALAP ordering

It is important to note that we actually use a modified version of ALAP topological ordering, otherwise Equation (3) might not always be respected. The modified ALAP version enforces that all actors having no incoming buffers (or only incoming buffers breaking cycles) have the lowest topological rank. A counter-example and a valid example are given in Figure 4.



(a) ✘ Regular ALAP topological ordering creating invalid graph cut.

(b) ✔ Modified ALAP topological ordering to have valid (i.e. feed-forward) graph cuts.

Fig. 4: Graph cut examples for regular and modified ALAP topological ordering.

Example for selection of balanced cuts

An example of preselection and final selection of cuts is depicted in Figure 5. 4 cuts are preselected by the heuristic with configuration D $2, 3$ on the given input graph having 9 actors in line. Each actor is executed only once and its ET is equal to 10. The two cuts with a dashed line correspond to the cuts found during the first enumeration of ASAP cuts. The two cuts with a dotted line correspond to the cuts found during the second enumeration of ALAP cuts. Note that there are less than 3 cuts preselected by each traversal because an extra condition stops the traversal when the sum of remaining rankLoad is higher than avgStageLoad $= 22$. The current value of the sum of rankLoad and the closest multiple of avgStageLoad when a cut is preselected is recalled below the cut in Figure 5. The ranks of the preselected cuts are: $4, 6, 7, 5$, in order of appearance. Except between cuts 4 and 7, none of the other pair of ranks respects Equation (2). The removal procedure first sorts the cuts by the size of their pipeline delays, and then starts in the reverse order of appearance to remove the largest cuts in delay size. In this case, all cuts imply the same delay size, and the first two cuts to compare are the cuts 5 and 7. As cuts 5 and 7 are too close to each other and imply the same delay size, the highest rank is removed by default: 7. Then only three preselected cuts remain: cuts $4, 6, 7$ and the removal procedure stops since three is the number of preselected cuts asked by the configuration D $2, 3$. Finally, the heuristic selects the first two of the remaining cuts: 4 and 6.

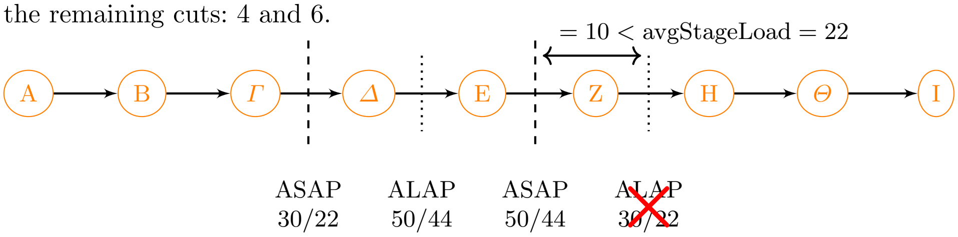


Fig. 5: Preselected and final cuts computed by the delay placement heuristic with configuration D $2, 3$ on a sample chain graph. Dotted cuts correspond to the preselected cuts while the dashed cuts correspond to the 2 final cuts. Each actor is fired once and has an ET equal to 10.