Automated Buffer Sizing of Dataflow Applications in a High-Level Synthesis Workflow

Alexandre Honorat, Mickaël Dardaillon, Hugo Miomandre, Jean-François Nezan

▶ To cite this version:

Alexandre Honorat, Mickaël Dardaillon, Hugo Miomandre, Jean-François Nezan. Automated Buffer Sizing of Dataflow Applications in a High-Level Synthesis Workflow. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2024, 17 (1), pp.1-26. 10.1145/3626103 . hal-04237266

HAL Id: hal-04237266
https://hal.science/hal-04237266v1

Submitted on 9 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Automated Buffer Sizing of Dataflow Applications in a High-Level Synthesis Workflow

ALEXANDRE HONORAT, Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, France
MICKAËL DARDAILLON, HUGO MIOMANDRE, and JEAN-FRANÇOIS NEZAN, Univ Rennes, INSA Rennes, CNRS, IETR – UMR 6164, F-35000 Rennes, France

High-Level Synthesis (HLS) tools are mature enough to provide efficient code generation for computation kernels on FPGA hardware. For more complex applications, multiple kernels may be connected by a dataflow graph. Although some tools, such as Xilinx Vitis HLS, support dataflow directives, they lack efficient analysis methods to compute the buffer sizes between kernels in a dataflow graph. This paper proposes an original method to safely approximate such buffer sizes. The first contribution computes an initial overestimation of buffer sizes, wihout knowing the memory access patterns of kernels. The second contribution iteratively refines those buffer sizes thanks to cosimulation. Moreover, the paper introduces an open source framework using these methods to facilitate dataflow programming on FPGA using HLS. The proposed methods and framework have been tested on 7 dataflow applications, and outperform Vitis HLS cosimulation in 5 benchmarks, either in terms of BRAM and LUT usage, or in term of exploration time. In the 2 other benchmarks, our best method gets results similar to Vitis HLS. Last but not least, our method admits directed cycles in the application graphs.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**.

Additional Key Words and Phrases: FPGA, High-Level Synthesis, dataflow, buffer sizing

ACM Reference Format:
Alexandre Honorat, Mickaël Dardaillon, Hugo Miomandre, and Jean-François Nezan. 2023. Automated Buffer Sizing of Dataflow Applications in a High-Level Synthesis Workflow. *ACM Trans. Reconfig. Technol. Syst.* 00, 0, Article 0 (2023), 26 pages. https://doi.org/10.1145/3626103

1 INTRODUCTION

High-Level Synthesis (HLS) technology is now mature to address large applications both on Field Programmable Gate-Array (FPGA) and Application-Specific Integrated Circuit (ASIC), with industrial tools Xilinx Vitis HLS and Intel HLS for FPGA, Catapult from Siemens and Stratus HLS from Cadence for ASIC, and open-source tools LegUp and Bambu [12]. Those tools have been proved efficient to create statically scheduled kernels based on sequential algorithms described in C/C++. They are however challenged to describe or extract parallelism from larger multi function programs, which is essential to take advantage of the hardware inherent parallelism at a coarser level. Multiple abstractions have been proposed to overcome this problem, with a particular interest in Domain-Specific Language (DSL) to raise the level of abstraction. A prominent example is machine learning, with the use of ONNX to abstract DNN as an high-level, architecture agnostic model which can be optimized to various platforms such as Graphics Processing Unit (GPU) and FPGA.

Authors' addresses: Alexandre Honorat, Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, Grenoble, France, 38000, alexandre.honorat@inria.fr; Mickaël Dardaillon, mickael.dardaillon@insa-rennes.fr; Hugo Miomandre, hugo.miomandre@insa-rennes.fr; Jean-François Nezan, jean-francois.nezan@insa-rennes.fr, Univ Rennes, INSA Rennes, CNRS, IETR – UMR 6164, F-35000 Rennes, 20 Av. des Buttes de Coesmes, Rennes, France, 35700.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
1936-7406/2023/0-ART0 $15.00
https://doi.org/10.1145/3626103

Other applications domains have also developed their own DSL to raise the level of abstraction, with approaches such as Halide [32] in image processing. The main goal of such tools based on program abstractions is to propose an implementation matching both hardware and user constraints.

While HLS tools, such as Vitis HLS, manage efficiently the hardware code generation of each kernel, tools based on program abstractions, such as FINN-R [6], take care of the code generation of data movements and standard kernels, and specialize both data types and kernel implementations. As Halide and to some extent FINN-R (via its support of ONNX), most of such program abstractions for HLS are data driven, with an explicit representation of data exchanged between computations and their interaction with these data. One problem is then to compute the sizes of buffers used for data exchanges. These program abstractions match dataflow models of computation such as Synchronous DataFlow (SDF) [25], which have a rich history of buffer analysis and optimization to propose parallel and memory efficient implementations, including FPGA ones [1, 23]. But as HLS allows for more and more complex kernels, applications in the SDF model are too coarse grain to analyze compared with the requirements and optimizations of FPGA implementations. Indeed, FPGA offer very limited memory on chip in the form of Block Random Access Memory (BRAM) and registers, but enable fine grain, custom synchronization at the pixel level instead of lines or an entire image. The challenge is then to analyze and propose optimized buffer sizes at this fine granularity, with a problem too large for classic approaches such as exact optimization based on exact memory *access patterns* [37].

In this work, we propose two complementary methods for the buffer sizing of dataflow graphs on FPGA; the first method gives theoretical worst-case bounds while the second method refines those bounds concretely based on iterative cosimulations. The first method models dataflow graphs using an affine representation amenable to Integer Linear Programming (ILP) analysis, without knowing the access patterns of kernels. This model is used to overestimate buffer sizes and provides a guarantee to reach optimal throughput of a dataflow graph of periodic actors. We extend this model to fine grain analysis for FPGA while keeping its fast and efficient analysis. The second method uses cosimulation as a basis to run Design Space Exploration (DSE) on buffer sizes. Multiple heuristics are proposed to reduce the number of iterations of the DSE by taking advantage of FPGA specificities and reach optimized buffer sizes. While the two buffer sizing methods are our main contributions, an additional contribution is their open-source implementation in the PREESM [31] fast prototyping tool now supporting code generation for dataflow graphs of Xilinx HLS kernels. This code generation enables us to evaluate our method on 7 open-source applications, 5 real ones plus 2 synthetic ones, and ensures reproducible results.

The rest of the paper is organized as follows. Section 2 sets the theoretical background on the dataflow model and its application on FPGA; Section 3 details our code generation flow and motivates our work to automatize buffer sizing. Section 4 introduces the Affine DataFlow Graph (ADFG) model and the ILP analysis proposed to bound buffer sizes; Section 5 describes the DSE refining the buffer size bounds with cosimulation. An evaluation of the contributions is reported in Section 6; finally, Section 7 reviews related work before concluding.

2 BACKGROUND

In this paper we consider buffer sizing of dataflow applications modeled with SDF graphs, and executed on FPGA with *free-running kernels*. This section presents the minimal knowledge needed about these two assumptions: first the SDF model and second the free-running mode.

2.1 SDF model

The SDF [25] model offers to design static applications as graphs of *actors* connected together by First-In First-Out (FIFO) *buffers*. Each actor can be *fired* multiple times and its data production (resp.

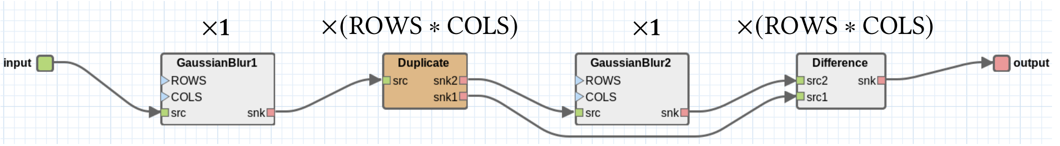


Fig. 1. Gaussian blur difference application as a multi-rate SDF graph. Above each actor is its number of firings.

consumption) rates on each output (resp. input) buffer are not required to equal the rates on the other end of the buffer. Thus the SDF model is said to be *multi-rate* whereas a mere task graph model with equal rates is said to be *single-rate*. The unit of data rates is called a *token* and can be associated to a different size in bits for each buffer. We denote as τ_p the production rate of a buffer, and τ_c the consumption rate.

One important feature of the SDF model is to assert buffers having bounded sizes. The SDF theory ensures this when a *repetition vector* of the given graph exists. Otherwise, the input graph is said to be *inconsistent*; in this work we consider only consistent SDF graphs. The repetition vector $\vec{r}$ stores the minimal number of actor *firings* so that all buffers go back to their initial state. SDF graphs can contain directed cycles, where at least one buffer must contain initial tokens called *delays* to avoid data starvation. The main problem tackled in this paper is to compute all buffer sizes δ which minimize memory requirements while avoiding deadlocks for a given throughput.

An example of a Gaussian difference application modeled as an SDF graph is given in Figure 1. In this application, there are four actors: two Gaussian blurs each processing a whole image of ROWS ∗ COLS pixels, and duplication and difference actors both fired at the pixel level. Hence, actors GaussianBlur1 and GaussianBlur2 are fired only once each, while Duplicate and Difference are fired ROWS ∗ COLS times each. The Gaussian difference application contains one *undirected graph cycle*[1] since there are two different paths between Duplicate and Difference actors. One path involves the extra actor GaussianBlur2 and thus implies a *cycle delay*: pixels at same location in the image will not arrive at the same time on both inputs of Difference actor.

The SDF model allows the processing resource to fire an actor only when at least $\tau_c(e)$ tokens are present on each input buffer e. In the case of a cycle delay, buffers of the two different cycle paths may need to have different sizes, depending on the schedule and the processing time of actors on each path. This makes the buffer sizing problem a difficult one. In this paper we assume that all actors are self-timed and data-driven, and run concurrently with no shared resource; this corresponds to an As Soon As Possible (ASAP) concurrent schedule [22].

In an FPGA, all input and output data are usually not consumed in a single time unit (called a clock cycle). Cyclo-Static DataFlow (CSDF) [5] is an extension of the SDF model which makes it possible to refine the token rates as a sequence of integers, eventually being only 0 and 1. Note that a coarse SDF view of a CSDF graph can be obtained by simply summing each CSDF token rate sequence. In this paper, we consider an SDF application graph as the main input, but we internally refine it as an CSDF abstracted one, as it is seen in Sections 4.1 and 4.2.

[1]An undirected cycle in a directed graph G is defined as a cycle in the corresponding undirected version of G.

2.2 HLS free-running kernels

The free-running mode[2] provided by the Xilinx Vitis tool enables FPGA designers to allow multiple kernel firings by a single C call in the HLS code. The kernels are then fired indefinitely as soon as their input data are available: the free-running mode corresponds to self-timed data-driven kernels. Besides, Xilinx Vitis provides a dataflow pragma supporting kernels linked together in the HLS code by *streams* having fixed *depths*. Such stream depths correspond to the buffer sizes that we compute. The HLS pragma is compatible with the free-running mode and we use both in this paper. Yet, most Xilinx dataflow examples are single-rate ones not benefiting from the free-running mode, such as Xilinx version of Gaussian difference[3].

In this paper, we consider only HLS kernels producing/consuming a constant number of tokens on each buffer. Moreover, all inter-kernel communications are done through FIFO buffers using `hls::stream` to enable fine grain buffer sizing. Those two assumptions about kernels and buffers match the SDF model, and in the rest of the paper, we identify the considered HLS *kernels* with SDF *actors*. In our internal analysis, we eventually refine kernels as CSDF abstracted actors, but without knowing precisely the token rates, a.k.a *access patterns*.

3 MOTIVATION: HLS GLUE CODE FOR DATAFLOW APPLICATIONS

One advantage of programming abstraction tools such as Halide [32] or FINN-R [6] is to let the user focus on the application functionnality by managing automatically glue code generation and kernel specialization. Such tools rely on existing HLS code compilers such as Xilinx Vitis HLS. In this paper, we focus on the SDF programming abstraction and use Xilinx Vitis HLS as compiler. Latest developments in Xilinx Vitis HLS made glue code generation for SDF application possible and we briefly present our contribution about automated code generation in Section 3.1. However, as Xilinx Vitis HLS requires bounded buffer sizes for hardware implementation, a first challenge is to compute initial buffer sizes for the glue code generation. A second challenge is to refine the initial buffer sizes thanks to cosimulation. Section 3.2 presents an overview of this buffer sizing approach, and both challenges are adressed further in Sections 4 and 5, respectively. Contrary to existing exact dataflow buffer sizing approaches [37], our approach scales to more and more complex kernels. Indeed, an extra constraint is to compute buffer sizes in a reasonable time.

3.1 Glue code generation

Our code generation is implemented in the open-source[4] dataflow framework PREESM [31],which we extended with FPGA support as presented in Figure 2. In this framework, the user (i.e. designer) on the left specifies their application as a dataflow graph of kernels in the SDF model using a graphical representation as in Figure 1. Kernels are implemented by the user for a specific target. Generic kernels, such as broadcasts, are provided by PREESM with optimized implementation. For example, the Duplicate kernel of Gaussian Difference application (depicted in Figure 1) is a generic broadcast kernel provided by PREESM. A generic kernel to initialize delays is also provided. Such *delay kernel* is automatically inserted in the graph, replacing a delayed buffer by two buffers around the delay kernel. It acts as a mere pass-through after the initialization.

Our implementation generates glue code for the FPGA, for testing before deployment, and for control by a host processor. For the device, a C++ dataflow region is generated with all free-running kernels as defined in Section 2.2 and buffers connecting those kernels. Additionally, data transfer kernels are generated for communication between the device and the host. Those kernels are

[2]https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Free-Running-Kernels
[3]https://github.com/Xilinx/Vitis_Libraries/tree/main/vision/L1/examples/gaussiandifference
[4]https://preesm.github.io

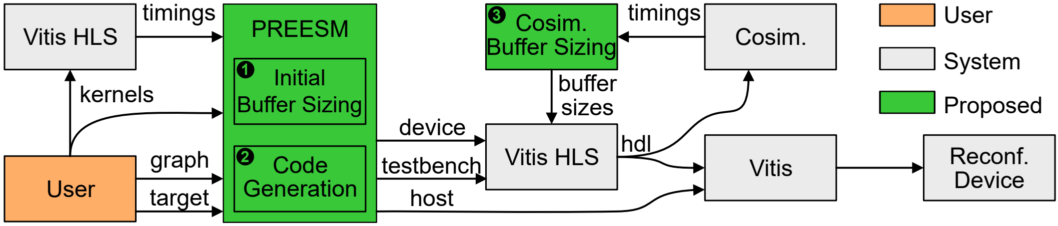


Fig. 2. PREESM compilation flow targeting FPGA.

functional but not optimized (e.g. buffer sizing, burst transfer), and could be improved as future work on heterogeneous optimization [35, 40].

A code excerpt generated for the Gaussian Difference application is given in Listing 1. The main kernel representing the whole dataflow graph, prefixed by `top_graph`, has as many arguments as the number of input and output streams connected to data transfer kernels and declared as `axi` streams in the first two `pragma`. The last two `pragma` disable classic kernel control in order to run subsequent kernel calls in free-running mode. Following lines (not shown) declare streams between kernels and specify their depths. Finally, each kernel is called once to be implemented for hardware synthesis.

For functional evaluation by the user using simulation or cosimulation, the generated kernels are sorted by their topological order in the dataflow graph in order to avoid starvation[5]. In this case, kernels are executed as many times as specified in the repetition vector $\vec{r}$ multiplied by the number of iterations as discussed later in Section 5.1. To further facilitate functional evaluation, testbench code is generated and interfaced with the main kernel, and is used for both simulation and cosimulation in Xilinx Vitis HLS.

For host interfacing, code is generated in multiple formats with OpenCL for Vitis, Python for PYNQ and bare metal for low level implementation on Zynq. In any case, all kernels of the considered application are mapped on the same unique FPGA target.

```
extern "C" {
void top_graph_GaussianDifference(
  hls::stream<ap_uint<8>> &input_stream,
  hls::stream<ap_uint<8>> &output_stream){
#pragma HLS INTERFACE axis port=input_stream name=input_stream
#pragma HLS INTERFACE axis port=output_stream name=output_stream
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS dataflow disable_start_propagation
    // follows declaration of streams with their depth
    ...
    // follows one call to each dataflow kernel with related streams
    ...
```

Listing 1. Xilinx Vitis HLS pragma used for the main kernel dataflow region, here generated for Gaussian Difference application.

3.2 Overview of our buffer sizing approach

The afformentioned glue code generation must declare streams with bounded buffer sizes for hardware implementation. Computing minimal buffer sizes with the knowledge of the exact memory access patterns of kernels does not scale and infering those access patterns from HLS kernel code is difficult as well. In this paper we propose an original approach to avoid those two

[5]The topological order also supports directed cycles thanks to the delays; delays indicate where to break the cycles so that the cycles' kernels can be ordered.

difficulties. First, we compute safe but overestimated buffer sizes in order to perform glue code generation for cosimulation. Second, we refine those buffer sizes thanks to iterated cosimulations. Figure 2 depicts the whole flow of our automated buffer sizing approach. The initial buffer sizing is performed internally by PREESM, right before glue code generation (center left, see steps ❶ and ❷). A loop can be observed on the top right part, in which the cosimulations are launched iteratively until buffer sizes cannot be refined anymore (see step ❸). The refinement stops when decreasing any of the buffer sizes slows down or deadlocks the application. Alternatively, it could be stopped as soon as the buffer sizes are small enough to be implemented on the target.

Our initial buffer sizing is fast because it does not rely on exact code analysis: cycle accurate behavior of actors including access patterns are not needed. Instead, a software synthesis is applied independently to each kernel in order to derive their timing metrics, from which their worst-case access patterns can be inferred. Those kernel timing metrics are the Initiation Interval (II) and the Execution Time (ET); they are produced by Xilinx Vitis HLS in our case (top left of Figure 2). Following the dataflow terminology, kernel ET denoted ET corresponds to each SDF actor total execution time, otherwise known as latency or Worst Case Execution Time (WCET). Kernel II denoted II corresponds to the period of a possibly pipelined actor, with $II \leq ET$ since the kernel may need initial computations to reach its steady pipelined state. In its CSDF refinement, a kernel II equals the lengths of all its related buffer input/output rate sequences. Based on those metrics, PREESM also computes the whole graph II, highlighting potential bottlenecks in the application.

Regarding the graph II timing metric, it is used for the buffer sizing refinement by iterated cosimulations: it must remain identical to assert that the refined buffer sizes do not decrease the application throughput. Between each cosimulation, the glue code is not regenerated and only buffer sizes are updated. Experiments show that the iterative refinement by cosimulation is faster, for two large applications, than the Xilinx Vitis HLS buffer sizing approach. Moreover, our approach works for application graphs having directed and undirected cycles, whereas the Xilinx Vitis HLS approach does not always support directed cycles.

4 INITIAL BUFFER SIZING

Initial buffer sizing is built on prior work on synchronous languages. In this section we present the conclusions of these works without providing proofs since they are either a simple adaptation to the constraints of HLS kernels, or taken as is from existing work. In particular, Equations (1) to (5) are our own, while Equations (6) to (10) come from multiple works on the ADFG tool [8, 18]. All of these equations rely on an abstraction of kernel firing clocks, whose ticks are kernel cycles in our case (Section 4.1). Such clocks make it possible to get an affine overestimation of token production and consumption (Section 4.2). The affine overestimation is the main part of this contribution: it enables us to not require exact memory access patterns of the kernels. Finally, an ILP formulation solves buffer sizing considering the affine overestimation (Section 4.3). While affine overestimation has been used for the buffer sizing of CSDF graphs [39] even before the ADFG tool, our new Equations (1) to (5) introduce linear constraints specific to FPGA and to the ILP formulation of the ADFG theory.

4.1 Abstraction of clock cycles

The concrete placement of token production and consumption during the execution of an application on FPGA is difficult to observe and even more difficult to predict. Yet some bounds can be quickly computed thanks to an abstraction of clock cycles: in this paper we consider a periodic abstraction of events occurring per clock cycle. Such events are: token production or consumption, pure processing and finally stall (i.e. absence of processing).

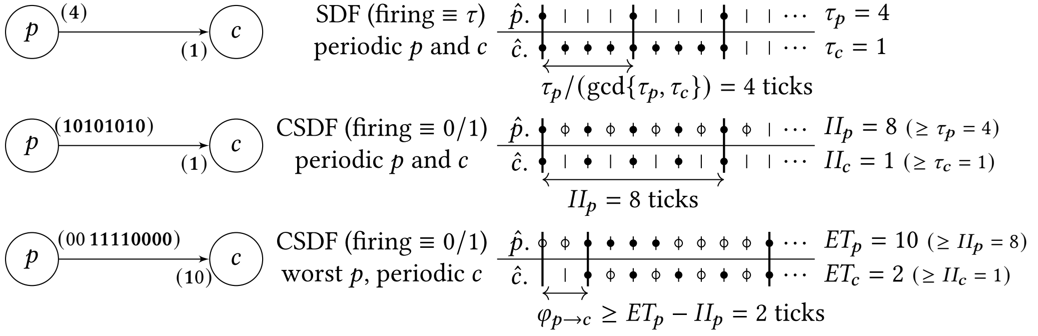


Fig. 3. Activation clocks of: SDF periodic kernel firings (top), CSDF periodic token production and consumption (center) and CSDF unbalanced production and periodic consumption with phase (bottom). Solid circles ● represent firings implying token production/consumption, and empty circles ○ represent the kernel processing not producing nor consuming any token. Vertical ticks | without any circle correspond to kernel stall.

At the SDF coarse level, token production and consumption as well as pure processing time of a single kernel firing are all considered at once. An example is depicted at the top of Figure 3, between a producer kernel p and a consumer c connected by a single buffer. Considering $\tau_p = 4$ and $\tau_c = 1$, the SDF repetition vector gives 4 firings of c for each of p in order to ensure bounded buffers. The periodic placement of firings over clock cycles minimizes locally the buffer size between p to c. Globally, p or c can be connected to other kernels affecting the buffer size: for example, if c waits for another input whose processing time is longer, more tokens of p will accumulate on the buffer before being consumed.

However, executing a kernel on FPGA constrains the token production and consumption a little bit more, which fits in the CSDF model. Indeed, we assume that only one token is produced or consumed per clock cycle, based on the buffer implementation constraints. This assumption can be relaxed by using multiple buffers or grouping tokens if they are produced by batch. As a consequence, II must be greater or equal than all connected buffer rates: $II_a \geq \max_{e_{*\to a} \cup e_{a \to *}} \tau_a(e)$. This is required to enable the production of all tokens during II cycles. Moreover we assume II and ET independent of kernel input/output values. Hence with the information of the II of each kernel, it is possible to deduce a best (periodic) case of token placement over II clock cycles, as at the center of Figure 3, or to deduce a worst case as at the bottom of Figure 3. In the best case, all token production and consumption occur on the same clock cycle; as kernels are executed in free-running mode, c stalls on half of the ticks while p is always present to process or to produce data. However the reality of implementation more commonly leads to unbalanced token production or consumption and unbalanced execution time. More generally, note that a clock cycle common to both p and c does not always imply a firing of p or c; it is especially the case when τ_p and τ_c are relative prime numbers in the SDF case, or II_p and II_c in the CSDF refinement we are now considering.

In the CSDF model considered in Figure 3, producing a token on a buffer (a *push*) and consuming another one from it (a *pop*) can occur during the same clock cycle. This is safe only if the buffer is neither full nor empty before such cycle, and the worst case is considered in our equations: a push, respectively a pop, can only occur if the buffer is not full, respectively empty, at the beginning of a cycle. An example execution is depicted in Figure 4, with a single producer actor connected to a single consumer actor (graph on top). On the producer side, II_p is always executed at the end, while on the consumer side, II_c is always executed at the beginning; furthermore no token production

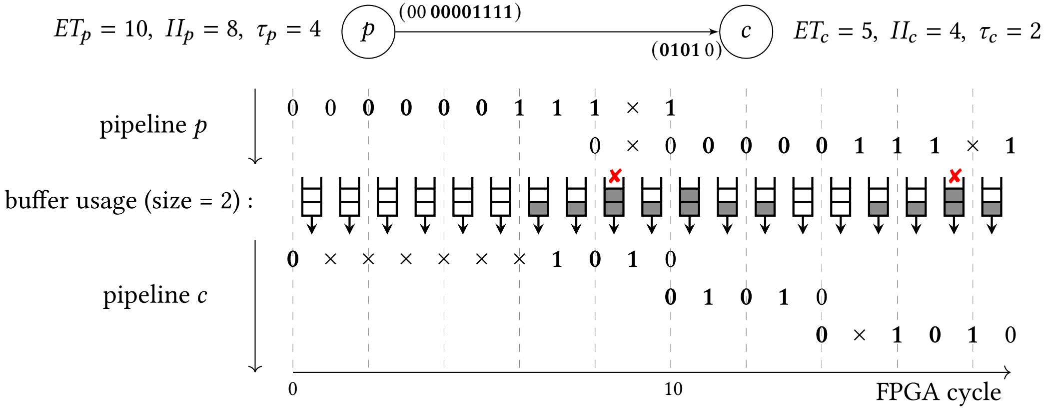


Fig. 4. Concrete execution of a producer and a consumer actor with a buffer of size 2. Corresponding CSDF graph and attributes are given on top. Small black crosses × represent actor stalls. Stylized red crosses ✘ precede producer stalls caused by the buffer capacity.

or consumption marked as 1 can appear outside the II rate sequence. This hypothesis ensures non-overlapping pushes or pops in the pipeline, which would come from the same actor accessing the same buffer. In both Figures 3 and 4, the II rate sequences are written in bold, and only this rate is actually considered in the CSDF refinement. Indeed, once the graph reaches *steady state* the same operations are periodically repeated, and only the II rate sequences remain useful to the analysis since the rates outside II cannot carry any token. The steady state is reached at cycle 8 in Figure 4. In the same example, the buffer of size 2 becomes full and causes a stall at cycle 8 for the producer actor p. This extra stall occurs periodically (e.g. at cycle 17) and the maximal throughput is not reached: a new complete execution of p occurs every 9 cycles whereas $II_p = 8$. In our equations, we compute buffer sizes to guarantee maximum throughput.

Finally, the clock abstraction is characterized by three variables: n, φ, d defining an *affine relation* between the producer and the consumer kernel of each buffer. Affine relations come from the theory of synchronous languages, as used for SIGNAL [24, 34][6]. Variables n and d characterize the linear part, whose fraction is irreducible and defined in Equation (1). They give the relative number of firings of p for each firing of c and vice versa. At the CSDF level, a firing is simply a busy clock cycle of token production or consumption, or a busy clock cycle of pure processing.

$$\frac{n}{d} = \frac{\tau_p}{II_p} \times \frac{II_c}{\tau_c} \tag{1}$$

The affine part is set by the phase φ, bounded as in Equation (2) to ensure that a token cannot be consumed before the producer kernel enters in its II phase. In this way, the potential latency of a pipelined kernel represented by its ET is taken into account, as in Figure 4.

$$\varphi_{p \to c} \geq ET_p - II_p \geq 0 \tag{2}$$

Note that this equation is an overestimation if initial tokens, a.k.a. delays, are present in the buffer. Moreover, in the case of directed graph cycles, φ can be negative since at least one kernel has to break the cycle. Then, if there is a delay $\theta_{e_{p \to c}}$ (always positive), it must compensate for the

[6] http://polychrony.inria.fr/publications.php

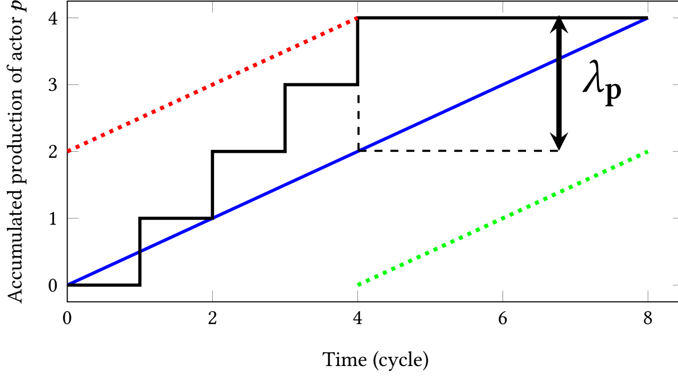


Fig. 5. Affine token production bounds over time (if no stall). Token production placement giving the worst λ_p value is plotted with the black stepped line (corresponds to Figure 3, bottom plot). Upper bound is plotted in dotted red (upper left), symmetrical lower bound in dotted green (lower right), average token production per clock cycle in solid blue (center).

extra phase introduced by the ET difference. Equation (3) formalizes this reasoning and generalizes Equation (2) (see Section 4.2 for a_p notation).

$$\theta_{e_{p\to c}} \geq a_p \times \frac{(ET_p - II_p) - \varphi_{p\to c}}{n} \tag{3}$$

The linear coefficients n and d are directly available and the periodic assumption also gives naturally the bottleneck of the graph as well as its own II, in Equation (4).

$$II_{\text{graph}} = \max_{\forall a \in \text{kernels}} \vec{r}[a] \times II_a \tag{4}$$

The bottleneck kernel is simply given considering argmax in the previous equation. However φ is bounded but remains to be computed, as well as the buffer sizes.

4.2 Affine overestimation

In order to bound the buffer sizes, we then need to introduce another variable depending on the token placement over the II clock cycles of a kernel. This variable λ represents the distance between the reality of token production or consumption and the ideal periodic case. In the ideal periodic case, $a_p = \frac{\tau_p}{II_p}$ tokens are produced per firing of p. In the worst case, all tokens are produced at the beginning, as in the bottom part of Figure 3, or symmetrically at the end of the kernel II. Figure 5 depicts the same worst case, this time with the total production in function of clock cycles. The ideal case corresponds to the blue solid line of slope a_p, in the center. λ_p is the maximal vertical distance between the a_p and the real token placement plotted with the black stepped line. Yet, it is possible to overestimate λ_p without knowing the real token placement, i.e. memory access pattern.

The worst possible λ can be computed as in Equation (5).

$$\lambda = \tau(1 - \frac{\tau}{II}) \tag{5}$$

Indeed, the fastest possible token production is achieved by the identity function $\tau = II$ since we assume $\tau \leq II$. So λ is the distance between maximum production τ and the total production at the clock cycle τ, that is: $\lambda_p = \tau - (a_p \times \tau)$. The unit of λ is a token and $\lambda \in [0, \tau[$.

The theory of ADFG states that the accumulated token production $\mathbb{p}_{\text{tot}}(\mathbb{c})$ at clock cycle $\mathbb{c}$ for a given buffer can then be bounded by affine equations as in Equation (6). In the general form,

λ is different for the lower bound and the upper one. Yet, without any more information about token production or consumption placement, we consider the worst case of symmetrical lambda overestimation: $\lambda^l = \lambda^u = \lambda$. Equation (6) is similar for the total token consumption case.

$$a_p \times \mathbb{c} - \lambda_p^l \quad \leq \ \mathbb{p}_{\text{tot}}(\mathbb{c}) \ \leq \quad a_p \times \mathbb{c} + \lambda_p^u \tag{6}$$

4.3 General ILP

Main equations from ADFG work [8, 9] enable to derive an ILP formulation to compute buffer sizes based on the aforementioned affine relations and lambda overestimation. As in FPGA, buffers are considered independent between them: especially, they do not share memory. Bounded buffer sizes within the ADFG periodic abstraction naturally avoids back-pressure: a firing occurs only if enough inputs are ready (avoids *underflow*) and if outputs can be pushed on buffer (avoids *overflow*).

Equation (7) avoids data underflow while Equation (8) avoids overflow. Their formulation requires two additional variables. First, θ_e which is a fixed constant in our case: θ_e is the number of initial tokens present on each buffer, being 0 by default. Second, δ_e which is free and represents the size of buffer e between kernels p and c.

$$\theta_e + a_p \frac{\varphi_{p\to c}}{n} \geq \lambda_c^u + \lambda_p^l + a_p \frac{n+d-1}{n} \tag{7}$$

$$\theta_e + a_c \frac{\varphi_{p\to c}}{d} \leq \delta_e - \lambda_p^u - \lambda_c^l - a_c \frac{n+d-1}{d} \tag{8}$$

These equations express a phase-delay-size trade-off: increasing initial tokens θ_e (fixed in our case) can compensate for the phase $\varphi_{p\to c}$ (variable to compute). But increasing both θ_e and $\varphi_{p\to c}$ increases the buffer size. Note that multiple buffers may exist between the same kernels, hence the δ_e notation. On the contrary, $\varphi_{p\to c}$ is only indexed by its producer and consumer kernels, and is common to all buffers between them. Here, λ is fixed and specific to each buffer side. Buffer e is not specified when there is no ambiguity, as for a_c, n, and d.

Equation (9) is needed for the support of directed and undirected graph cycles (of length k affine relations). This equation ensures that all phases involved in a graph cycle actually compensate for each other: at least one kernel in the cycle has to start before the others and so must have a negative phase with its predecessor. There is one equation per graph cycle in the cycle basis, detected with Paton's algorithm [30], which runs in cubic time.

$$\sum_{i=1}^{k} \Big(\prod_{l=1}^{i-1} d_l\Big)\Big(\prod_{l=i+1}^{k} n_l\Big)\varphi_i = 0 \tag{9}$$

Finally, the ILP objective function is to minimize the sum of buffer sizes. As there is a trade-off between phases and buffer sizes δ, the objective actually minimizes the sum of all of them considering absolute values of phases φ.

$$\sum_{e \in \text{buffers}} \delta_e + \sum_{e_{p\to c} \in \text{buffers}} \frac{a_c}{d} |\varphi_{p\to c}| \tag{10}$$

This ILP formulation suffers from a major drawback: all equations involving quotients need to be scaled by a common denominator. Such common denominator can be arbitrarily large depending on the relative primality of the numerators. To avoid reaching values too large we have relaxed the equations in two ways. A first relaxation is to upgrade the II of producer and consumer kernels to $\max\{II_p, II_c\}$ if $II_p \approx II_c$ (i.e. $< 1\%$ difference); this relaxation will slightly overestimate the worst case λ metric. A second optional relaxation is to consider the Linear Programming (LP) relaxation of the ILP formulation, that is, to allow phases and buffer sizes being real numbers in $\mathbb{R}^+$. This relaxation may produce unsound results and we consider it only if the ILP solver does not find

solutions. Two cases make the solver fail: either an application with directed cycles lacks delays on some buffers, or the integer values of common quotient denominators are too large to be handled correctly by the solver. The latter case occurred in several of our experiments, but can be avoided by replacing the current solver with a more robust one [7]. Rounding up the obtained real values gave valid results in our experiments.

In the end, the benefits of using ADFG theory are its soundness[8], the support of graph cycles, and its computational rapidity if solved as LP relaxation (despite the cycle basis algorithm). However, the periodic abstraction enforces some solutions and can be suboptimal, worsening the overestimation of worst case λ. Also, we do not get the concrete resulting schedule but only a periodic abstraction of it. Considering the overestimation drawback, we overcome this by combining this first formal buffer sizing method with a more classical DSE of buffer sizes thanks to cosimulation, as described in the next section.

5 COSIMULATION BUFFER SIZING

Buffer sizing can be obtained with a DSE starting from a safe overestimation of buffer sizes. The DSE iteratively decreases those initial buffer sizes while checking at each step, with a cosimulation, that the graph still matches our optimization constraints. In our experiment the constraints are

(1) that throughput measured as II_{graph} is respected and
(2) that the graph is still live, i.e. not deadlocked.

If those constraints are not respected, it means that last buffer size decrease was too large, then we say that the cosimulation *failed* and the DSE tries a smaller buffer size decrease. While this method is rather naive, multiple aspects can be optimized. In particular, our method consists of multiple additional heuristics which smartly choose:

- the cosimulation duration, that is the number of kernel executions to run before asserting if steady state was reached, see Section 5.1;
- the order in which buffer sizes are decreased, see Section 5.2;
- and finally the next smaller buffer size to evaluate at each new step, see Sections 5.3 and 5.4.

All of those heuristics are complementary and can be used together; moreover, they are all deterministic.

5.1 Simple bisection

The standard DSE performs a bisection on each buffer size until any further decrease makes the cosimulation fail, and then optimizes the next buffer. However it is not easy to predict when such failure can occur during the cosimulation, and consequently how long to run the cosimulation. *How long* is not measured in time in the case of dataflow graphs: it is measured in the number of graph *iterations*, an iteration being firing every kernel the number of times specified in the repetition vector $\vec{r}$. The cosimulation cannot be directly parameterized by the number of iterations, but it can be indirectly thanks to the amount of input data given to the dataflow graph. If one dataflow graph iteration processes a single image, then the number of iterations is the number of images provided as cosimulation input data.

Before actually performing the DSE, the number of required graph iterations is first evaluated in the following way: starting from 3 graph iterations, a cosimulation is launched until reaching a *steady state*. The steady state is reached when the two last iterations II^*_{graph} are measured to be equal. II^*_{graph} denotes the practical value of II_{graph} (see Equation (4)) as measured by the cosimulation

[7]The PREESM tool uses the solver ojAlgo (https://www.ojalgo.org/), version 53.0. The solver GLPK-online (https://cocoto.github.io/glpk-online/) is able to solve all the ILP formulations otherwise failing with ojAlgo in our benchmarks.
[8]All formal proofs are available in original Bouakaz' thesis [8].

tool. Xilinx Vitis HLS actually provides ET^*_{graph} and we compute $II^*_{\text{graph}}(i)$ of graph iteration i as: $ET^*_{\text{graph}}(i) - ET^*_{\text{graph}}(i-1)$, hence 3 graph iterations as a starting point. If the graph does not reach the steady state, the number of graph iterations is doubled and a new cosimulation runs, until the steady state is detected. Along this entire process the initial buffer sizes are not optimized yet so if they are safely overestimated, then a steady state is always reached[9] and there cannot be any deadlock for those buffer sizes. In the remaining part of the paper, we say that a *cosimulation reaches the steady state* only if both aforementioned optimization constraints about throughput equivalence and absence of deadlock are also respected.

The DSE starts once the required number of graph iterations to reach the steady state is set. The number of graph iterations can actually decrease during the DSE since smaller buffer sizes constrain the graph even more, which brings the steady state earlier. Once a steady state is reached at a smaller graph iteration, this iteration number is set as the new maximum for all subsequent cosimulations (except if it failed).

Note that Xilinx Vitis HLS offers the `cosim_design -enable_fifo_sizing` option to perform buffer sizing, starting with infinite buffer sizes by default, or with the ones provided by the designer otherwise. Yet the Xilinx option does not provide info about the number of graph iterations needed to compute the buffer sizes, and our heuristic to reach the steady state can be used in this context too. In our experiments we always call `cosim_design -enable_fifo_sizing` with the steady state heuristic since we observed that buffer sizes returned by the Xilinx tool do not provide throughput guarantee if not measured at steady state.

5.2 Bisection by lambda optimization

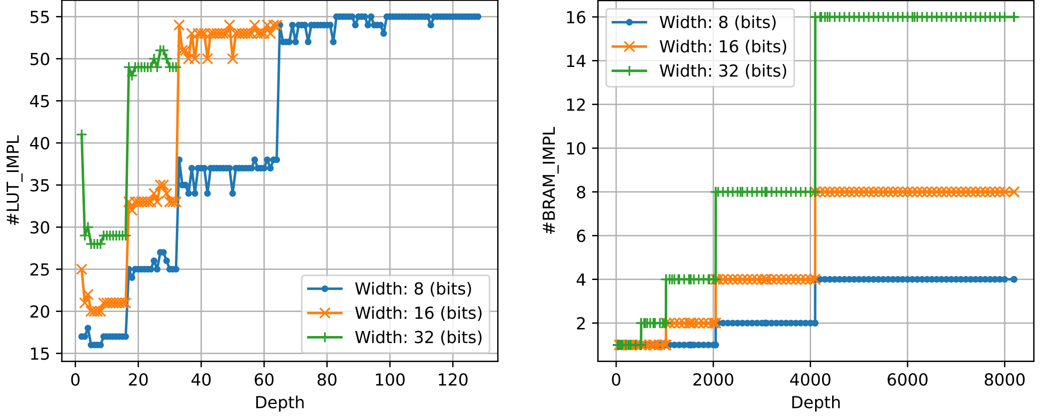
In the simple bisection heuristic, each buffer size is optimized up to its minimum, one after the other. However, the overestimation performed with ADFG theory also gives an indication about which buffer sizes are more likely overestimated than others. This indication lies in the affine bounds depending on λ, as depicted in Figure 5. Large λ values imply numerous possible token placements over the clock ticks, and consequently a larger overestimation since initial buffer sizing considers the worst possible λ. Yet, λ is not defined by the buffer, but by its *end*: one for the producer side and another for the consumer side. Thus the heuristic here is to consider the sum of producer and consumer λ of each buffer e in order to sort them, and to start by optimizing the largest $\lambda_{e_{p\to c}} = \lambda_p + \lambda_c$.

Only considering buffers sorted by $\lambda_{e_{p\to c}}$ do not speed up the simple bisection DSE. To do so, the heuristic also considers large $\lambda_{e_{p\to c}}$ altogether: a single cosimulation is launched for the next smaller buffer size of all buffers having $\lambda_{e_{p\to c}} > 0.5 \times \max_{e\in\text{buffers}}\{\lambda_{e_{p\to c}}\}$. If the cosimulation reaches the steady state, it saves as many cosimulations as the number of buffers considered together minus one. Moreover, the value of $\lambda_{e_{p\to c}}$ is divided by 2 for each optimized buffer to reflect the reduction in size and overestimation. At the opposite, if the cosimulation fails, it costs an extra cosimulation, and each buffer is then optimized independently as in the simple bisection version. In such case, their $\lambda_{e_{p\to c}}$ values are updated independently according to each cosimulation result: if it fails again, $\lambda_{e_{p\to c}}$ is set to 0 to avoid reconsidering buffer e, and otherwise $\lambda_{e_{p\to c}}$ is divided by 2. The DSE iterates with the lambda optimization until all $\lambda_{e_{p\to c}} = 0$. Then it falls back to the simple bisection.

5.3 Initial test optimization

Except in directed or undirected cycles, the buffers do not require large sizes: for example, if the dataflow graph is a tree, all buffers could be of size 2. Even in cycles, the buffers do not need large sizes if the involved kernels have similar timings and access patterns. Thus, the initial test

[9]This is true because we assume II and ET independent to kernel input/output, see Section 4.1.



(a) LUT usage for buffer sizes requiring less than 1 BRAM (post place and route).

(b) BRAM usage for buffer sizes requiring at least 1 BRAM (post place and route).

Fig. 6. Non-linear evolution of resource usage depending on buffer size in number of tokens, a.k.a. depth, and on token width for Xilinx Zynq 7020 SoC target (hardware implementation post place and route).

optimization tries to optimize each buffer directly to its minimum size ≥ 2 before starting the bisection. If the cosimulation reaches the steady state, the tested minimum buffer size is set and it avoids all intermediate steps of the bisection. In the case of a bisection by lambda, $\lambda_{e_{p\to c}}$ is set as well. Otherwise, if the cosimulation fails, the initial test continues and tries to optimize the next buffer. Note that after setting a buffer size to its initial minimum value, the buffer size can still be optimized to the concrete minimum of 2 by the bisection.

The minimum buffer size tested by the heuristic is 5 in our case and this value is not chosen arbitrarily: it minimizes the number of Look-Up Tables (LUTs) on the tested Xilinx Zynq 7020 SoC target. Indeed, we experimented on a micro-benchmark containing only one buffer which is popped and pushed a large number of times in a `for` loop. The experiment aimed to measure the resource usage of that buffer on the hardware implementation (post place and route). Resource usage is measured in function of the width in bits of buffer tokens, and in function of the specified buffer size in number of tokens, a.k.a. depth according to Xilinx terminology. Some results of that experiment are reported in Figure 6, especially Figure 6a for the case of small buffer sizes. Surprisingly it can be seen for widths 8, 16 and 32 that LUT usage is always higher for buffer size 2 than for buffer size 5 (from 41 LUTs down to 29 LUTs for 32 bits width). Then LUT usage increases again with the buffer size, but the minimum LUT usage is observed for buffer size 5. Those small variations are caused by the FIFO access control implementation with hardcoded boundaries values, and are technology dependent. The minimum buffer size can be set per technology in our framework, and the dataflow model allows for larger than needed FIFOs to take advantage of the smaller LUT usage. In the case of bisection by lambda, $\lambda_{e_{p\to c}}$ is then fixed to 4, that is the minimum initial buffer size minus one token.

5.4 Resource-wise optimization

Most buffer optimizations consider resource cost as a linear function of the buffer size. In this optimization we study how to more accurately model the hardware cost of a buffer to reduce the number of steps of our DSE. Figure 6a gives the usage of LUT (measured post place and route)

for small buffer sizes, vaguely corresponding to a step function. Yet, besides the global minimum for buffer size 5, it is hard to model LUT usage given the very low cost and large variation from one buffer size to the next. However, the usage of BRAM clearly corresponds to a monotonic step function according to our measurements and it is possible to establish a model of it. The same monotonic step function is observed for the results obtained after software synthesis.

Equation (11) gives the number of BRAM in function of the buffer size δ_e, the width in bits of tokens stored by the buffer, and BRAM_size whose value is either 18 Kib, or 16 Kib if $\delta_e > 4096$. While this model has been established by reverse-engineering for the Xilinx Zynq 7020 SoC target, it is possible to deduce a similar one for most common platforms. Indeed, this model reflects the fact that BRAMs or equivalents come in discrete quantities with a limited set of read/write ports, meaning that a single FIFO can be mapped to a BRAM even if it does not fill the entire BRAM memory. Such model enables the DSE to optimize the hardware resource usage rather than buffer sizes *per se*.

$$\#\text{BRAM} = \left\lceil 2^{\lceil \log_2(\delta_e) \rceil} \times \frac{\text{token_width}(e)}{\text{BRAM_size}} \right\rceil \tag{11}$$

Finally, resource-wise optimization consists of two heuristics to decide which next buffer size to use in the cosimulations launched by the bisection, instead of simply dividing the current one by 2. First, it is possible thanks to the BRAM model to directly go to the largest buffer size fitting the lower plateau of the step function depending on buffer size in number of tokens and on token bit width. Additionally, the bisection stops if both lower bound and upper bound of the buffer size correspond to the same strictly positive number of BRAM. Second, if the buffer size requires no BRAM at all, our heuristic directly stops the optimization of this buffer because the improvement in number of LUT will be negligible compared to the global number of LUT of the application. Here the LUT model corresponds to a constant cost if no BRAM is involved. We respectively call the two aforementioned heuristics *BRAM buffer model* and *BRAM+LUT buffer model* in the following section presenting our evaluation.

6 EXPERIMENTS

This section presents the evaluation of our proposed approach. It first introduces the benchmarks used to evaluate our proposed flow, and then the experimental results obtained using the different methods introduced in Sections 4 and 5.

6.1 Benchmarks

We select a set of benchmarks to demonstrate the buffer sizing opportunities using PREESM on FPGA. The main difficulty is to find existing benchmarks with graph cycles in HLS as a comparison. Up to now, applications with graph cycles are considered challenging due to the very problem of finding buffer sizes, and are not recommended by Xilinx in its dataflow directive, advising against *single-producer-consumer violations, bypassing tasks and channel sizing, and feedback betwen tasks*[10]. In that sense this work is an enabler, and few applications can be reused as is. We have designed 7 dataflow benchmark applications to demonstrate our method, all having at least one undirected cycle; 2 applications have directed cycles.

The only application based on existing work is Gaussian Difference. It is based on the namesake application provided as part of the Xilinx Vitis Libraries, and was rewritten with no dependence to Xilinx Libraries for open-source distribution. The application is showcased in Figure 1, which

[10]https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls/Dataflow-Optimization-Limitations

Table 1. Characteristics of tested applications. Left columns gives application name and input image resolution. Middle columns gives metrics inherent to the dataflow graph. #cycles counts undirected graph cycles. Right columns gives metrics depending on kernels in the application and computed with ADFG theory.

Application	resolution	#kernels	#firings	#buffers	#cycles	max $\lambda_{e_{p \to c}}$	II_{graph}
Gaussian Diff.	720x540	3	777602	4	1	1267	390072
Gaussian Cyclic	720x540	4	1166402	5	1	1267	390072
Color Filter	720x540	8	1944005	8	1	2530	1166400
DWT	352x288	17	304720	20	4	1554	102955
IDWT	352x288	17	304720	20	4	1554	102955
Synthetic undir.	min	8	30	9	3	2	48
Synthetic directed	min	8	30	9	4	2	56
Synthetic undir.	max	32	131070	47	15	2	196608
Synthetic directed	max	32	131070	47	16	2	229376

presents a single undirected graph cycle. We also designed another version of the Gaussian Difference application, integrating a directed cycle: the image difference is performed with the previous iteration result.

The next 3 benchmarks were developed specifically to evaluate the performance of our approach. Color filter is an image processing pipeline similar to the color detect example provided by Xilinx, with the added value of using the result as a filter on the original image, creating a single large graph cycle. DWT and IDWT are the direct and inverse versions of the discrete wavelet transform (a Daubechies 2 implementation), used for signal coding and compression such as JPEG2000. Those applications are multirate, taking advantage of SDF expressivity and the free-running kernels implementation. They include multiple undirected graph cycles and poses the biggest challenge for buffer sizing.

The first 5 benchmarks are listed with their characteristics in the top part of Table 1. Those applications are all in the image processing domain, and need to be able to analyze large resolution, with nearly 2 million firings for Color Filter. This is a challenge to express using a single rate approach and would require a potentially large number of duplicated kernels. The analysis of those applications for fast buffer sizing is made possible by the ADFG theory as presented in the next Section 6.2. Any other approach using token per token and/or single rate analysis is difficult to scale to that resolution. The $\lambda_{e_{p \to c}}$ metric is an indicator of uncertainty on the possible timing of token production or consumption. This is the main source of overestimation of ADFG theory. The high values of maximum $\lambda_{e_{p \to c}}$ (up to 2530) and II_{graph} (up to 1 million) show the variability in kernel characteristics and the challenge posed for analysis. Regarding kernel internal code, multiple different kernels may implement the same function and IDWT, for example, uses 6 different functions on a total of 17 kernels. However, each kernel has its own implementation of the function in the generated code. The other benchmark applications use 3 to 6 different functions in their kernels. All benchmarks are available publicly as part of the PREESM applications repository[11].

The last 2 benchmarks, in the bottom part of Table 1, are synthetic as they do not model any real application. Their number of kernels in the SDF graph are parameterized. We used multiple versions of those applications and report only the minimum and the maximum value of each characteristic. Figure 7 presents a small version of the synthetic application with directed cycles. Kernels form a

[11] https://github.com/preesm/preesm-apps/tree/master/fpga

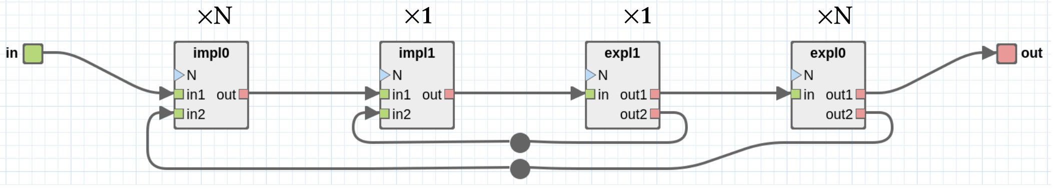


Fig. 7. Synthetic application with directed cycles as a multi-rate SDF graph. Above each actor is its number of firings. The black circles represent delays.

mere chain, connected with extra feedback buffers to create as many directed cycles. Kernels on the left "implode" their $2 \times N$ integer inputs in a single output by accumulating them. Kernels on the right "explode"' their unique input by duplicating it on each output N times. In our case $N = 2$, makes the number of firings grow exponentially: $2^{\frac{\#\text{kernel}}{2}-1}$ for the kernels at both ends. A large number of delays, up to $2^{\frac{\#\text{kernel}}{2}+3}$ in the outermost feedback buffer, ensures that no deadlock occurs. In the version with undirected cycles, feedback buffers are replaced by forward ones without any delay, and kernels are reversed to "explode" tokens on the left and "implode" tokens on the right to keep the graph consistent. These two benchmarks are not representative of real applications; but despite their simplicity, they highlight different behaviors when using our buffer sizing method or the one of Vitis HLS.

6.2 Results

To evaluate our approach, we base our experiments on the previously presented open-source PREESM dataflow framework and open-source benchmarks. Synthesis, cosimulation and bitfile generation are performed on an Ubuntu 20.04 desktop with an Intel core i9 10900, 64 GB of RAM and a 1 TB SSD, using Xilinx Vitis HLS 2021.2. All results in this section are reported after HLS synthesis, with additional implementation results after place and route reported for Table 2. Most of the reported exploration time stems from cosimulation, a fully sequential operation as implemented by Xilinx Vitis HLS with `cosim_design` tool. Target hardware is the Zynq 7020 SoC with a synthesis frequency of 100 MHz, and implementations are tested both on the Zedboard for the OpenCL and bare metal interfaces, and on the PYNQ Z2 for the PYNQ interface.

Our main comparison point is the `-enable_fifo_sizing` option provided by Xilinx Vitis HLS `cosim_design` tool. This option is used considering the heuristic to reach the steady state, as presented in Section 5.1. To evaluate this heuristic, we record the number of cosimulations launched by the buffer sizing method, denoted *#step*, as well as the total number of graph iterations, denoted *#iteration*. We noticed that `-enable_fifo_sizing` option actually runs two cosimulations one after the other; this behavior is taken into account to measure the *#step* and *#iteration* metrics.

Main synthesis results are reported in Table 2. For each image processing application we measure the resource usage in BRAM and LUT, at the end of different buffer sizing methods. For each method, total exploration time (including all syntheses and cosimulations) and total number of steps and graph iterations are also reported. The different buffer sizing methods respectively are, from top to bottom:

- BS-1 initial buffer sizing with ADFG theory (see Section 4), which is used as a first step of all subsequent methods;
- BS-2 Xilinx Vitis HLS cosimulation with `-enable_fifo_sizing` option, the main comparison method;

BS-3 simple bisection (see Section 5.1);
BS-4 simple bisection with BRAM buffer model (see Section 5.4);
BS-5 simple bisection with BRAM+LUT buffer model (see Section 5.4);
BS-6 initial test (see Section 5.3) then bisection by lambda (see Section 5.2) with simple bisection (see Section 5.1) as fallback;
BS-7 finally, all heuristics of Section 5 with BRAM+LUT buffer model.

For small applications Gaussian Difference and Color Filter in Table 2, respectively having only 3 and 8 kernels, our methods BS-4 to BS-7 find the reference number of BRAM of the Xilinx method BS-2. Similarly, number of LUTs also stays in the same order of magnitude. Method BS-3 on Gaussian Difference is an exception, requiring 6 BRAM instead of the reference 5. The order in which buffers are optimized is causing our heuristic to reach a local minimum: the first two buffers are reduced to size 2 and prevent correctly optimizing the last two buffers, left at sizes 787 and 986. On the contrary, Xilinx reference method BS-2 optimizes the first two buffers to size 3 only, the third one to size 730 and finally the last one to size 3 again. Thus a too tight optimization of each buffer one after the other does not always result in a good solution. But most importantly, our methods are not competitive w.r.t. exploration time for small applications: for the fastest method BS-7, exploration time is up to 10 times longer than the reference (for Color Filter).

However, while our methods fall behind Xilinx for the exploration time on small applications, an opposite conclusion is found for larger applications in Table 2. For both DWT and IDWT applications, our method BS-7 outperforms the reference method BS-2 in the number of BRAM by a factor up to 2 (for IDWT). Moreover, exploration time is of same order of magnitude for both methods BS-2 and BS-7; it is slightly better for method BS-7 in the case of DWT: 6953 sec. against 8100 sec. for the reference. In only one case, for DWT method BS-2, we noticed a significant difference between the number of BRAM predicted by synthesis, 56, higher than the one after implementation, 52. Nevertheless, all methods BS-3 to BS-7 gives a better buffer sizing reducing the number of BRAM compared to the reference.

The preceding observations show that method BS-7 combining all heuristics is the most effective for the buffer sizing of our two large applications, but not for the two small ones. We shall now focus on some particular aspects of the heuristics and the applications, including the synthetic ones, before concluding on those experiments.

Number of graph iterations. The first heuristic, presented in Section 5.1, limits the number of graph iterations; the minimum is 3 iterations and the maximum is reached at steady state for ADFG initial buffer sizing. The same graph iteration heuristic is used by all methods BS-2 to BS-7 but the order in which they reduce buffer size influence the number of iterations to reach the steady state. The small applications Gaussian Difference and Color Filter do not benefit from this heuristic, since their number of graph iterations is always the minimum of 3 for each launched cosimulation. This result can be seen by dividing the #iteration column by the #step column in Table 2. On the contrary, reaching the steady state for large applications DWT and IDWT requires up to 24 graph iterations[12] for method BS-2. We notice that the average number of graph iterations is larger for Xilinx method BS-2, around 11 iterations, than for all subsequent methods, around 3 for DWT and 4 for IDWT. The instrumentation introduced by method BS-2 is increasing the number of iterations which, combined with two cosimulations causes this overhead. For methods BS-3 to BS-7, the initial maximum number of graph iterations to reach the steady state is only 6 whereas the heuristic uses the same initial ADFG sizing. The difference in average, 11 iterations for method BS-2 versus 3 and 4 for others, also highlights the fast decrease of maximum graph iteration during the bisection,

[12]As the number of graph iterations doubles until reaching the steady state, the maximum can be retrieved for Xilinx method BS-2 (not using bisection but launching two cosimulations per try): $90 = (3 + 6 + 12 + 24) \times 2$.

Table 2. Results of a selection of buffer sizing methods for the 5 image processing benchmarks. BRAM and LUT resource usage is given by the last successful cosimulation during DSE, and exploration time includes the failed ones. Exploration time also includes HLS synthesis + cosimulation runtime over at least 3 graph iterations, except for pure ADFG (no need for DSE). #step is the total number of launched cosimulations and #iteration is the total number of graph iterations.

	Buffer Sizing method	#BRAM	#LUT	time (sec.)	#step	#iteration
Gaussian Diff.	BS-1 [ADFG]	12	1711	< 1	–	–
	BS-2 [+ Vitis HLS]	**5**	**1681**	**175**	2	6
	BS-3 [+ bisection]	6	1702	5091	48	144
	BS-4 [+ bisection/BRAM]	5	1669	3109	30	90
	BS-5 [+ bisection/BRAM+LUT]	5	2020	969	9	27
	BS-6 [+ init/λ-bisection]	5	1683	1762	18	54
	BS-7 [+ init/λ-bisection/BRAM+LUT]	5	1666	642	7	21
Gaussian Cyclic	BS-1 [ADFG]	266	1911	< 1	–	–
	BS-2 [+ Vitis HLS]	–	–	–	–	–
	BS-3 [+ bisection]	265	1916	7227	53	159
	BS-4 [+ bisection/BRAM]	262	1897	1751	15	45
	BS-5 [+ bisection/BRAM+LUT]	262	2014	842	8	24
	BS-6 [+ init/λ-bisection]	262	1883	4551	38	114
	BS-7 [+ init/λ-bisection/BRAM+LUT]	(86*) **262**	**1881**	**855**	8	24
Color Filter	BS-1 [ADFG]	61	5044	< 1	–	–
	BS-2 [+ Vitis HLS]	**18**	**4986**	**1260**	2	6
	BS-3 [+ bisection]	18	4988	68707	92	276
	BS-4 [+ bisection/BRAM]	18	4988	36700	49	147
	BS-5 [+ bisection/BRAM+LUT]	18	5455	15629	21	63
	BS-6 [+ init/λ-bisection]	18	5019	33854	46	138
	BS-7 [+ init/λ-bisection/BRAM+LUT]	18	4988	12380	17	51
DWT	BS-1 [ADFG]	96	31490	< 1	–	–
	BS-2 [+ Vitis HLS]	(52*) 56	31430	**8100**	8	90
	BS-3 [+ bisection]	40	31404	54737	164	498
	BS-4 [+ bisection/BRAM]	40	31380	32480	96	294
	BS-5 [+ bisection/BRAM+LUT]	40	31514	10663	30	96
	BS-6 [+ init/λ-bisection]	40	31404	25012	74	227
	BS-7 [+ init/λ-bisection/BRAM+LUT]	**40**	**31370**	**6953**	20	65
IDWT	BS-1 [ADFG]	98	33015	< 1	–	–
	BS-2 [+ Vitis HLS]	82	32987	**8351**	8	90
	BS-3 [+ bisection]	42	32887	79241	173	723
	BS-4 [+ bisection/BRAM]	38	32879	47384	107	425
	BS-5 [+ bisection/BRAM+LUT]	38	33045	14630	31	133
	BS-6 [+ init/λ-bisection]	38	32884	27784	72	239
	BS-7 [+ init/λ-bisection/BRAM+LUT]	**38**	**32865**	**9074**	20	83

Table 3. Results of buffer sizing cosimulation in terms of number of buffers optimized. A buffer is considered as optimized if its size is reduced compared with the initial ADFG sizing. Numbers in the last row are included in those of the above row.

#buffers in the application	Gaussian Diff.	Gaussian Cyclic	Color Filter	DWT	IDWT
in total	4	5	8	20	20
optimized by BS-2 [Vitis]	4	–	7	10	2
optimized by BS-7 [ours]	4	2	7	12	12
↪ directly to size 5	3	2	4	8	10

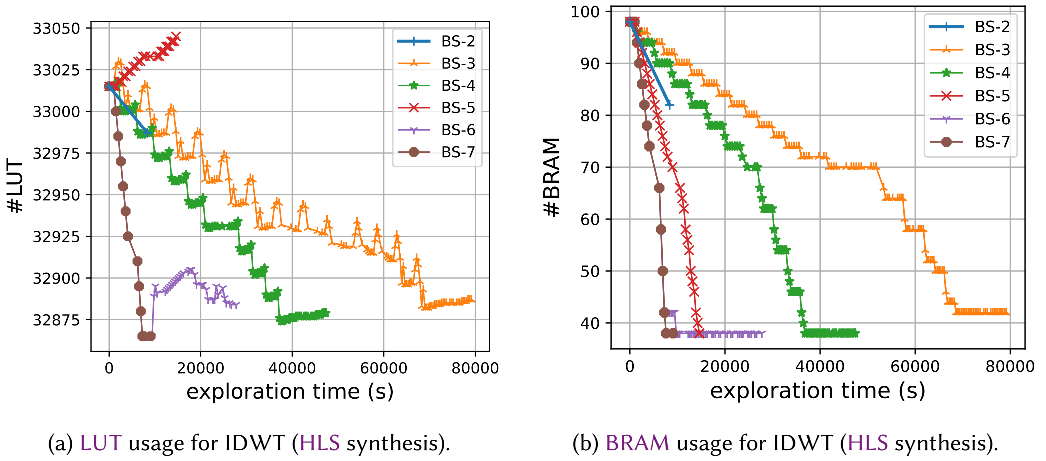
while optimizing buffer sizes. Whether or not this fact is correlated to the efficiency of our method for DWT and IDWT applications requires more investigation. Nevertheless, the exploration time is dominated by the total number of graph iterations, and decreasing the number of graph iterations per cosimulation is crucial.

Number of optimized buffers. Another way to evaluate our buffer sizing method is to consider the number of buffers whose size is actually reduced between the initial ADFG sizing overestimation and the final result. Number of optimized buffers is reported in Table 3 for two buffer sizing methods and all benchmarks. For DWT and IDWT applications, around half of the buffers are not modified by the best method BS-7. This result suggests a relatively good efficiency of the ADFG initial buffer sizing: it overestimates only half of the buffers. Table 3 also provides the number of buffers directly reduced to size 5 according to the heuristic presented in Section 5.3. This heuristic is quite efficient for all applications: it reduces the size of around half of the buffers. In the end, the other heuristics in method BS-7 optimize a limited number of buffers, which is the difference between last two rows: for example, 4 buffers and 2 buffers over 20 respectively for DWT and IDWT applications.

Focus on Gaussian Difference. Note that our implementation differs from the Xilinx Library[13]: we offer the same functionality without relying on the Xilinx Vision Vitis Library. The main buffer size between the Duplicate and Difference kernels (see Figure 1) has a hardcoded size of 15360 pixels, that is almost a full image, in the Xilinx version for resolution 128x128. In our implementation considering resolution 720x540, the same buffer has a size of 2541 pixels using only ADFG, 2048 pixels if considering BRAM+LUT model, and 727 otherwise. This implementation demonstrates the difficulty of setting buffer sizes even for simple applications. The buffer size computed by ADFG is already an order of magnitude lower without resorting to cosimulation. A smaller buffer size is also an advantage for cosimulation methods to reduce the number of iterations needed to reach the steady state, directly proportional to the method exploration time.

Focus on Gaussian Cyclic. This variation of the Gaussian Difference application features one directed cycle and makes the buffer sizing problem challenging. Our method is able to compute buffer sizes in such case, but Xilinx Vitis HLS stops as soon as it detects a cycle and so there is no result for method BS-2 on the Gaussian Cyclic application in Tables 2 and 3. Yet, Xilinx Vitis HLS does not always detect deadlocks on cycles during regular cosimulations without its buffer sizing option: we observed that the cosimulation runs indefinitely in such case, with all signals blocked. Thus, a timeout has been added to detect deadlocked cosimulations, so that Method BS-2 always terminates. Moreover, the number of BRAM after place and route is significantly reduced compared

[13] https://github.com/Xilinx/Vitis_Libraries/tree/main/vision/L1/examples/gaussiandifference



(a) LUT usage for IDWT (HLS synthesis).

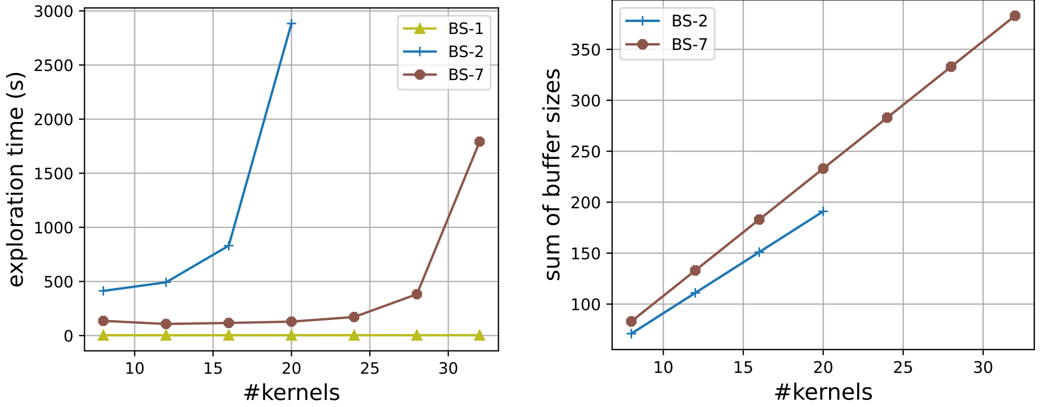
(b) BRAM usage for IDWT (HLS synthesis).

Fig. 8. Resource usage comparison of IDWT for different cosimulation buffer sizing methods. Cumulative exploration time of each method is only reported for valid buffer sizes, but includes failed cosimulations.

to the synthesis results of Vitis HLS. We expect this to come from the implemented delay which requires a large amount of BRAM, and the application was still validated on hardware.

Focus on exploration time. IDWT results of methods BS-2 to BS-7 in Table 2 are also reported in Figure 8 in function of the cumulative exploration time of the buffer sizing method at each new valid buffer sizing (cumulative exploration time includes runtime of failed cosimulations). All methods can be clearly distinguished, method BS-7 being the fastest on the left of both plots. On Figure 8a representing LUT usage, method BS-5 using the BRAM+LUT buffer model is the only one to increase over exploration time. This phenomenon is due to the fact that the heuristic of method BS-5 stops the bisection at the end of each BRAM plateau, which uses more LUT if no BRAM is required (see corresponding LUT usage in Figure 6a). This shows the limit of our BRAM+LUT buffer model, but the resource overhead is still negligible with an increase of less than 0.1% in LUT usage. On Figure 8b, an interesting point is the presence of small plateaus for methods BS-3 and BS-4 because they optimize buffer sizes down to the exact token even if there is no resource gain. On the contrary, method BS-6 has only one plateau at the end. This phenomenon is due to the initial test optimization (see Section 5.3) which first optimizes all buffers one by one to size 5 if possible, and then performs a bisection. As small buffers are already optimized to size 5, any further reductions up to minimum size 2 made by the bisection do not improve the BRAM usage. Best performing method BS-5 and especially BS-7 avoids such plateaus thanks to the BRAM+LUT buffer model. Similar plots are obtained for all the 5 image processing benchmarks.

Focus on synthetic benchmark applications. For the two synthetic applications, we measured the exploration time and the sum of buffer sizes given by the proposed method BS-7 and by Vitis HLS method BS-2, for an increasing number of kernels (and even more firings) in each application. The runtime of method BS-1 has been separated from the exploration time. It remains constant and low (< 1 sec.) even when the number of kernels, firings and cycles increase. For the version of the application with undirected cycles, the results are plotted in Figure 9. In this case, the exploration time of both methods quickly explodes, but the explosion comes faster for method BS-2 than for the proposed method BS-7: we had to stop method BS-2 after 1 hour whereas ours gave a result in a few minutes for a graph with 24 kernels. Regarding the quality of the solution, measured



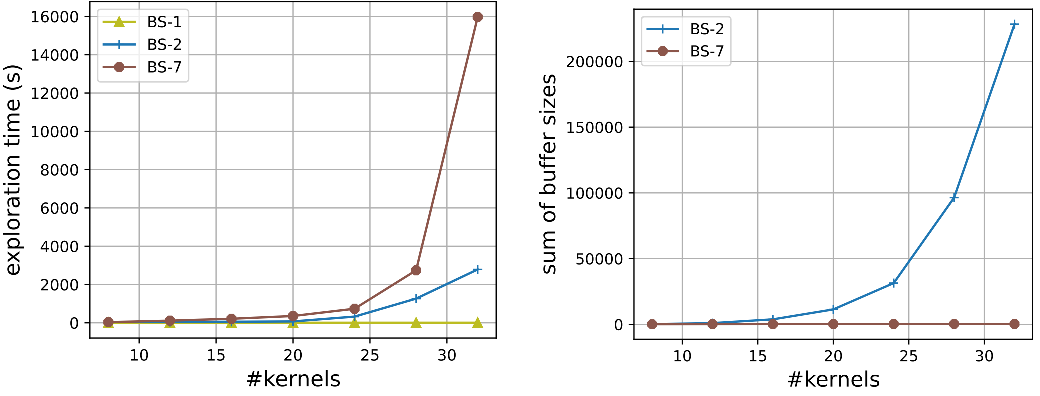
(a) Total exploration time of methods BS-2 and BS-7 (HLS synthesis), not including the common time for method BS-1.

(b) Sum of buffer sizes computed by methods BS-2 and BS-7 (HLS synthesis).

Fig. 9. Results of methods BS-2 and BS-7 on the synthetic benchmark with undirected cycles.

as the sum of buffer sizes, both methods reach the same order of magnitude. Our method BS-7 requires slightly larger sizes because it does not further improve a buffer size once it holds in LUTs only and no BRAM (see the BRAM+LUT buffer model in Section 5.4). For the version of the application with directed cycles, the results are plotted in Figure 10 and show an opposite behavior: our method BS-7 is quite slower than the Vitis HLS method BS-2 (more than 5× slower for 32 kernels) but it drastically improves the buffer sizes at such point that no BRAM is required. This second result is also related to the difficulty of computing the number of required delays: it appears that way less delays than initially designed are necessary to avoid deadlocks. Computing the minimal number of delays is still an open-problem for which only sufficient conditions are known [28]. The Vitis HLS method BS-2 appears to be conservative and sets buffer sizes so that all delays hold in it; on the contrary our method BS-7 continues to optimize buffer sizes, taking advantage of the fact that not all delays are necessary.

Conclusion on experiments. The evaluation of our sizing methods (especially method BS-7) asserts their efficiency for 2 large image processing applications. For small applications, only the ADFG sizing method presented in Section 4 appears useful (by providing an initial buffer sizes refined by Vitis HLS only, see method BS-2). For the 2 tested synthetic dataflow applications with different number of kernels and firings, our method BS-7 outperforms Vitis HLS method BS-2, regarding either exploration time or buffer sizes but not both at the same time. Unfortunately, we lack more benchmarks to strengthen these observations. For example, the bisection by lambda (see method BS-6) shows no clear benefit since it seemed to be outperformed by the other heuristics. Moreover, we cannot yet characterize precisely the applications for which our approach is particularly efficient, as for DWT and IDWT applications. Important application characteristics to consider for future evaluations are the access patterns of the kernels since our current synthetic applications contain only kernels with two very simple functions. To our knowledge, there is no current HLS dataflow benchmark able to represent this diversity of characteristics, especially considering SDF graphs. In this sense our work makes a wider category of dataflow applications amenable for the problem of buffer sizing, and we call for new benchmarks benefiting from it. In particular, we are able to



(a) Total exploration time of methods BS-2 and BS-7 (HLS synthesis), not including the common time for method BS-1.

(b) Sum of buffer sizes computed by methods BS-2 and BS-7 (HLS synthesis).

Fig. 10. Results of methods BS-2 and BS-7 on the synthetic benchmark with directed cycles.

analyze application graphs with directed cycles, whereas Xilinx Vitis HLS is not able to refine their buffer sizes in our experiments. Last but not least, we provide to Vitis HLS initial buffer sizes in order to perform its own refinement of buffer sizes, which the tool is currently unable to provide. Those sizes guarantee to reach optimal throughput at the cost of an overestimation, based on the ADFG theory.

7 RELATED WORK

The buffer sizing problem has been studied both from a dataflow model perspective and from an HLS compilation chain perspective.

7.1 SDF analysis for buffer sizing

Multiple buffer sizing results exist for SDF graphs, but usually for CPU architecture using static schedules. For example, the APGAN algorithm [4] clusters dataflow actors together to minimize the memory of the static schedule and to ease the buffer size analysis. This is not applicable in our case: clustering kernels increases their memory requirements while forcing the designers to manually set the internal buffer sizes. Yet, the SDF model has already been used at coarse grain and with static schedules in the context of FPGA, as in the Grape-II tool [1, 23] (limited to a subset of SDF).

At the opposite, memory *access patterns* [16], model SDF applications at the cycle level when a token is pushed or popped from a buffer. Due to cycle level precision, SDF with access patterns, a.k.a. SDF-AP, requires a high computation time [37] to solve buffer sizing with ILP. SDF-AP is close to CSDF for which optimal buffer sizing methods exist [21, 39]; however, such generic methods do not handle the specificity of FPGA such as Equation (2) in our case. SDF-AP too does not match perfectly with FPGA implementations and has been refined to handle stalls and direct token consumption [13], still facing the same complexity and poor scalability. SDF-AP has also been used in the generation of HLS code from templated Haskell [15]; this work is limited by the access patterns required as an input from the user. In any case, it is difficult to know the exact access patterns, especially for closed-source implementations (IP). Our method works without knowledge of the access patterns, using instead the ADFG abstraction to reduce the analysis complexity.

In [3], authors develop a method to compute the maximum achievable throughput of a dataflow graph of actors similar to FPGA kernels. Their analysis takes benefit of an extra Finite State Machine (FSM) separating the kernel phases into an initial read (*rush-in*), a cyclic computation phase and a final write (*rush-out*). Their method requires the current buffer sizing as an input, and is run iteratively to discover which buffer sizes enable the maximal throughput. However this is possible only if the FSM controlling the kernel rush-in and rush-out phases is known. In our work we have no hypothesis on such kernel phases.

Same authors also proposed an iterative buffer sizing method [17, 36] without needing for FSMs. Their input is still a pure CSDF [5] graph, and thus they benefit from dedicated throughput analysis methods to assert the throughput of each buffer size distribution. Instead PREESM rely on Xilinx Vitis HLS cosimulations to do so, as we consider the application kernels as black boxes, especially not knowing the precise start of rush-in and rush-out phases. Moreover to accelerate our iterative method, PREESM benefits from the initial buffer size distribution computed with ADFG, and from the resource-wise optimization to reduce the number of iterations. Both techniques could be reused to accelerate the method in [17, 36]. As their implementation is not open-source, we were not able to compare with it. We expect their implementation to be close to our method BS-6 at best or method BS-3 at worst, as they do not model hardware ressource usage as in method BS-7.

To overcome the lack of a kernel internal model, other approaches such as CAPH [33] language rely on SystemC simulation to refine bufffer sizing. Similarly, the FLASH [11] tool provides fast cycle-accurate simulation extracted from the HLS C++ model. As FLASH extracts FSM to model the communications of kernels, such FSM may help to refine the way we overestimate the λ metric (see Equation (5)) in PREESM. Interesting future work could also reuse the FLASH model in the iterative cosimulation, but FLASH is not open-source at the moment.

7.2 HLS tools with buffer optimization

In a recent review [29] on HLS from 2020, buffer sizing was mentioned only for the CAPH [33] language and related compilation tool, but has been a big interest recently. For example, both FINN-R [6] and hls4ml [7] HLS tools perform dataflow buffer sizing, relying on the the Xilinx Vitis HLS tool (see method BS-2).

More precise buffer sizing methods come at the cost of their execution time: up to an hour [20] for fine grain single-rate dataflow graphs whose buffer placement is solved at the same time by ILP. In single-rate dataflow, all kernels are executed the same number of times, simplifying analysis. Same authors proposed the DASS [10] tool which can reuse memory thanks to dynamic scheduling. Our tool, PREESM, does not support dynamic scheduling but supports multi-rate dataflow.

A recent interest in dataflow-inspired DSL synthesis using HLS has spurred new work addressing buffer sizing for FPGA. Aetherling [14] compiler is also able to compute precise buffer sizing thanks to its type system embedding the clock delay implied by each computational operator it supports. More precisely, Aetherling performs retiming using registers to balance the operator delays, instead of streams sizes in our case. As Aetherling relies on the Halide [32] language, unknown operators are not supported. On the contrary, PREESM considers kernels as black boxes without assumptions on their inner C++ code. HeteroFlow [40] too takes Halide as an input, via the HeteroHalide [27] compiler. While HeteroFlow supports data transfer directives written by the designer, as well as buffer reuse, it does not embed a delay type system to perform buffer sizing. The FLOWER [2] tool relies on AnyDSL [26] to get input dataflow graphs. Similarly to PREESM, FLOWER generates glue code linking kernels with streams, and separates a top kernel from an input and output one. To our understanding, HeteroFlow and FLOWER are limited to single-rate dataflow.

The AutoSA [38] tool focus on the synthesis of systolic arrays on FPGA thanks to polyhedral techniques applied on C input code. Clockwork [19] also relies on polyhedral techniques to compile

image processing applications but does not focus on systolic arrays. Clockwork supports multi-rate dataflow by extracting an SDF graph from the polyhedral representation of an application. This polyhedral representation also enables Clockwork to perform optimizations such as loop fusion. With an analysis time ranging from seconds to minutes, Clockwork is proving efficient in the range of applications it can address. Both AutoSA [38] and Clockwork [19] are complementary tools to our approach, able to provide an optimized and statically scheduled part of the kernels present in the input dataflow graphs, similarly to the approach presented in DASS [10].

8 CONCLUSION

In this paper, we presented an original analysis and DSE heuristics to optimize buffer sizes. The ADFG-based analysis enables quick prototyping with performance guarantees based on high level information provided by the HLS static scheduler. Our DSE further improves buffer sizes of two large image processing applications, with up to a factor 2 improvement in BRAM usage compared with the approach proposed by Xilinx, whereas the exploration time is of same order of magnitude. The improvement regarding buffer sizes is even greater for applications having directed cycles. However, this comes at the cost of a slower exploration time in some cases, including two small image processing applications. In any case, Vitis HLS does not provide initial overestimates of buffer sizes in order to refine them. Our other contribution, adapting the ADFG theory to FPGA, improves this point by rapidly providing to Vitis HLS such overestimated sizes. Overall we demonstrate the buffer sizing of multi-rate SDF applications on FPGA, without prior knowledge on the access patterns of the kernels. Our contributions were demonstrated on an open-source framework with functional code generation on Xilinx FPGA based on HLS. Both our analysis method and code generation support cycles in the dataflow graph, which is the main challenge of buffer sizing and makes our tool an enabler for multi-rate dataflow applications. Future work aims at extending our approach by taking advantage of kernel abstractions such as the work by Choi et al. [11] to both speedup cosimulation during DSE and potentially enhance kernel description in the ADFG theory. Looking further, a natural extension of our analysis would be to consider external memory and heterogeneous mapping, which would challenge our model by requiring multiple abstraction levels.

ACKNOWLEDGMENTS

This work was part of the EXCELCAR 2 project[14], which received funding from the European Union and the Région Bretagne (France) as part of the ERDF programme. We thank Pedro Ciambra for his valuable comments.

REFERENCES

[1] M. Ade, R. Lauwereins, and J.A. Peperstraete. 1994. Buffer memory requirements in DSP applications. In *Proceedings of IEEE 5th International Workshop on Rapid System Prototyping*. 108–123. https://doi.org/10.1109/IWRSP.1994.315904

[2] Puya Amiri, Arsene Perard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leiba, and Sebastian Hack. 2021. FLOWER: A comprehensive dataflow compiler for high-level synthesis. In *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, Auckland, New Zealand, 1–9. https://doi.org/10.1109/ICFPT52863.2021.9609930

[3] Hadi Alizadeh Ara, Amir Behrouzian, Martijn Hendriks, Marc Geilen, Dip Goswami, and Twan Basten. 2018. Scalable Analysis for Multi-Scale Dataflow Models. *ACM Trans. Embed. Comput. Syst.* 17, 4, Article 80 (aug 2018), 26 pages. https://doi.org/10.1145/3233183

[4] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. 1999. Synthesis of Embedded Software from Synchronous Dataflow Specifications. *Journal of VLSI signal processing systems for signal, image and video technology* 21 (1999), 151–166.

[14]https://kohesio.ec.europa.eu/en/projects/Q3687075

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. 1996. Cycle-static Dataflow. *Trans. Sig. Proc.* 44, 2 (Feb. 1996), 397–408. https://doi.org/10.1109/78.485935

[6] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfigurable Technol. Syst.* 11, 3, Article 16 (Dec. 2018), 23 pages. https://doi.org/10.1145/3242897

[7] Hendrik Borras, Giuseppe Di Guglielmo, Javier Duarte, Nicolò Ghielmetti, Ben Hawks, Scott Hauck, Shih-Chieh Hsu, Ryan Kastner, Jason Liang, Andres Meza, Jules Muhizi, Tai Nguyen, Rushil Roy, Nhan Tran, Yaman Umuroglu, Olivia Weng, Aidan Yokuda, and Michaela Blott. 2022. Open-source FPGA-ML codesign for the MLPerf Tiny Benchmark. arXiv:2206.11791 [cs.LG]

[8] Adnan Bouakaz. 2013. *Real-time scheduling of dataflow graphs.* Theses. Université Rennes 1. https://tel.archives-ouvertes.fr/tel-00945453

[9] A. Bouakaz, J.-P. Talpin, and J. Vitek. 2012. Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications. In *Proceedings of the 2012 12th International Conference on Application of Concurrency to System Design.* ACM, Hamburg, Germany, 183–192. https://doi.org/10.1109/ACSD.2012.16

[10] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. 2021. DASS: Combining Dynamic and Static Scheduling in High-level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2021), 628–641. https://doi.org/10.1109/TCAD.2021.3065902

[11] Young-Kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. 2020. FLASH: Fast, Parallel, and Accurate Simulator for HLS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4828–4841. https://doi.org/10.1109/TCAD.2020.2970597

[12] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4 (Dec. 2022), 1–42. https://doi.org/10.1145/3530775

[13] Ke Du, Stéphane Domas, and Michel Lenczner. 2019. Actors with stretchable access patterns. *Integration* 66 (2019), 44–59. https://doi.org/10.1016/j.vlsi.2019.01.001

[14] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 408–422. https://doi.org/10.1145/3385412.3385983

[15] H. H. Folmer, R. de Groote, and M. J. G. Bekooij. 2022. High-Level Synthesis of Digital Circuits from Template Haskell and SDF-AP. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Marc Reichenbach, and Matthias Jung (Eds.). Springer International Publishing, Cham, 3–27.

[16] A. Ghosal, R. Limaye, K. Ravindran, S. Tripakis, A. Prasad, G. Wang, T. N. Tran, and H. Andrade. 2012. Static dataflow with access patterns: Semantics and analysis. In *DAC Design Automation Conference 2012.* 656–663. https://doi.org/10.1145/2228360.2228479

[17] Martijn Hendriks, Hadi Alizadeh Ara, Marc Geilen, Twan Basten, Ruben Guerra Marin, Rob de Jong, and Steven van der Vlugt. 2019. Monotonic Optimization of Dataflow Buffer Sizes. *Journal of Signal Processing Systems* 91, 1 (Jan. 2019), 21–32. https://doi.org/10.1007/s11265-018-1415-2

[18] A. Honorat, H. N. Tran, L. Besnard, T. Gautier, J.-P. Talpin, and A. Bouakaz. 2017. ADFG: a scheduling synthesis tool for dataflow graphs in real-time systems. In *International Conference on Real-Time Networks and Systems* . Grenoble, France, 1–10. https://doi.org/10.1145/3139258.3139267

[19] Dillon Huff, Steve Dai, and Pat Hanrahan. 2021. Clockwork: Resource-Efficient Static Scheduling for Multi-Rate Image Processing Applications on FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE, Orlando, FL, USA, 186–194. https://doi.org/10.1109/FCCM51124.2021.00030

[20] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20).* Association for Computing Machinery, Seaside, CA, USA, 186–196. https://doi.org/10.1145/3373087.3375314

[21] Jaime Koh and Bruno Bodin. 2022. K-Periodic Scheduling for Throughput-Buffering Trade-Off Exploration of CSDF. *ACM Trans. Embed. Comput. Syst.* 22, 1, Article 19 (oct 2022), 28 pages. https://doi.org/10.1145/3559760

[22] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.* 31, 4 (Dec. 1999), 406–471. https://doi.org/10.1145/344588.344618

[23] R. Lauwereins, M. Engels, M. Ade, and J.A. Peperstraete. 1995. Grape-II: a system-level prototyping environment for DSP applications. *Computer* 28, 2 (1995), 35–43. https://doi.org/10.1109/2.347998

[24] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. 2003. *Polychrony for system design.* Research Report RR-4715. INRIA. https://hal.inria.fr/inria-00071871

[25] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[26] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 119 (oct 2018), 30 pages. https://doi.org/10.1145/3276489

[27] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[28] Olivier Marchetti and Alix Munier-Kordon. 2009. A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research* 197, 2 (Sept. 2009), 532–540. https://doi.org/10.1016/j.ejor.2008.07.037

[29] Mostafa W. Numan, Braden J. Phillips, Gavin S. Puddy, and Katrina Falkner. 2020. Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains. *IEEE Access* 8 (2020), 174692–174722. https://doi.org/10.1109/ACCESS.2020.3024098

[30] Keith Paton. 1969. An Algorithm for Finding a Fundamental Set of Cycles of a Graph. *Commun. ACM* 12, 9 (sep 1969), 514–518. https://doi.org/10.1145/363219.363232

[31] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. 2014. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In *European Embedded Design in Education and Research Conference (EDERC)*. 36–40. https://doi.org/10.1109/EDERC.2014.6924354

[32] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (jul 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

[33] Jocelyn Sérot, François Berry, and Sameer Ahmed. 2013. *CAPH: A Language for Implementing Stream-Processing Applications on FPGAs.* Springer New York, New York, NY, 201–224. https://doi.org/10.1007/978-1-4614-1362-2_9

[34] Irina Smarandache, Thierry Gautier, and Paul Le Guernic. 1999. Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints. In *World Congress on Formal Methods in the Development of Computing Systems (FM'99) (LNCS vol. 1709)*. Springer, Toulouse, France, 1364–1383. https://doi.org/10.1007/3-540-48118-4_22

[35] Stephanie Soldavini, Karl F. A. Friebel, Mattia Tibaldi, Gerald Hempel, Jeronimo Castrillon, and Christian Pilato. 2022. Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics. *ACM Transactions on Reconfigurable Technology and Systems* (Sept. 2022), 3563553. https://doi.org/10.1145/3563553 arXiv:2203.10850 [cs].

[36] Steven van der Vlugt, Hadi Alizadeh Ara, Rob de Jong, Martijn Hendriks, Ruben Guerra Marin, Marc Geilen, and Dip Goswami. 2019. Modeling and Analysis of FPGA Accelerators for Real-Time Streaming Video Processing in the Healthcare Domain. *Journal of Signal Processing Systems* 91, 1 (Jan. 2019), 75–91. https://doi.org/10.1007/s11265-018-1414-3

[37] G.Q. Wang, R. Allen, H.A. Andrade, and A. Sangiovanni-Vincentelli. 2014. Communication storage optimization for static dataflow with access patterns under periodic scheduling and throughput constraint. *Computers & Electrical Engineering* 40, 6 (2014), 1858 – 1873. https://doi.org/10.1016/j.compeleceng.2014.05.002

[38] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Virtual Event USA, 93–104. https://doi.org/10.1145/3431920.3439292

[39] Maarten H. Wiggers, Marco J.G. Bekooij, and Gerard J.M. Smit. 2007. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *2007 44th ACM/IEEE Design Automation Conference*. 658–663.

[40] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 11.

Received 22 February 2023; revised 9 June 2023; revised 18 August 2023; accepted 1 September 2023