POLITECNICO DI TORINO
Repository ISTITUZIONALE

A DSP shared is a DSP earned: HLS Task-Level Multi-Pumping for High-Performance Low-Resource Designs

Original
A DSP shared is a DSP earned: HLS Task-Level Multi-Pumping for High-Performance Low-Resource Designs / Brignone, G., Lazarescu, M.T., Lavagno, L.. - ELETTRONICO. - (2023), pp. 551-557. (2023 IEEE 41st International Conference on Computer Design (ICCD) Washington (USA) 06-08 November 2023) [10.1109/ICCD58817.2023.00089].

Availability:
This version is available at: 11583/2981775 since: 2024-01-24T09:31:31Z

Publisher:
IEEE

Published
DOI:10.1109/ICCD58817.2023.00089

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

01 November 2026

A DSP shared is a DSP earned: HLS Task-Level Multi-Pumping for High-Performance Low-Resource Designs

Giovanni Brignone, Mihai T. Lazarescu, Luciano Lavagno
Dipartimento di Elettronica e Telecomunicazioni
Politecnico di Torino
Turin, Italy
{giovanni.brignone, mihai.lazarescu, luciano.lavagno}@polito.it

***Abstract*—High-level synthesis (HLS) enhances digital hardware design productivity through a high abstraction level. Even if the HLS abstraction prevents fine-grained manual register-transfer level (RTL) optimizations, it also enables automatable optimizations that would be unfeasible or hard to automate at RTL. Specifically, we propose a task-level multi-pumping methodology to reduce resource utilization, particularly digital signal processors (DSPs), while preserving the throughput of HLS kernels modeled as dataflow graphs (DFGs) targeting field-programmable gate arrays. The methodology exploits the HLS resource sharing to automatically insert the logic for reusing the same functional unit for different operations. In addition, it relies on multi-clock DFGs to run the multi-pumped tasks at higher frequencies. The methodology scales the pipeline initiation interval (II) and the clock frequency constraints of resource-intensive tasks by a multi-pumping factor (M). The looser II allows sharing the same resource among M different operations, while the tighter clock frequency preserves the throughput. We verified that our methodology opens a new Pareto front in the throughput and resource space by applying it to open-source HLS designs using state-of-the-art commercial HLS and implementation tools by Xilinx. The multi-pumped designs require up to 40% fewer DSP resources at the same throughput as the original designs optimized for performance (i.e., running at the maximum clock frequency) and achieve up to 50% better throughput using the same DSPs as the original designs optimized for resources with a single clock.**

***Index Terms*—Dataflow architectures, FPGA, high-level synthesis, multi-pumping, resource sharing**

I. Introduction

High-level synthesis (HLS) raises the abstraction level of electronic design automation tools to improve the digital hardware designer's productivity. The high abstraction precludes some low-level manual optimizations, making the quality of results (QoR) of HLS circuits inferior to those manually optimized at the register-transfer level (RTL), especially for the area and maximum clock frequency [1]. On the other hand, we deem the HLS description introduces new optimization opportunities at a high level.

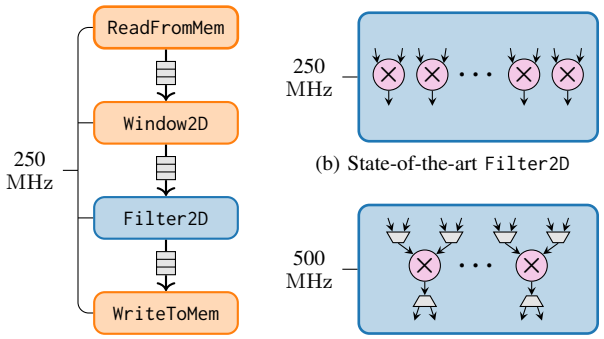
We focus on HLS designs modeled as dataflow graphs (DFGs) (e.g., with *dataflow* in Xilinx Vivado/Vitis HLS [2], *hierarchy* in Siemens Catapult HLS [3], or *task functions* in Intel HLS compiler [4]). Modeling HLS designs as DFGs proved its effectiveness both in industrial [5], [6] and academic [7], [8] projects.

A DFG is a set of parallel computational tasks (C/C++ functions in HLS) communicating asynchronously through first-in-first-out (FIFO) queues. HLS tools typically implement DFGs as single-clock dataflow graphs (SCDFGs), where all the tasks share the same clock signal. Many modern HLS tools do not support multi-clock designs [2], [4]. Nevertheless, we can generalize SCDFGs to multi-clock dataflow graphs (MCDFGs) by assigning each task to a dedicated clock domain. The generalization enhances the tasks' flexibility and maximum frequency, limited only by the critical timing path local to the task rather than the global one. Clock architectures of modern field-programmable gate array (FPGA) system-on-chips (SoCs) seamlessly support multiple clocks, and the area overhead for safe clock domain crossing (CDC) is negligible since the tasks already communicate through FIFOs, which can be configured with independent read and write clocks with comparable resource utilization [9].

Multiple clock domains allow optimizations like multi-pumping, which reduces the area while preserving the throughput by reusing M times a resource, usually a digital signal processor (DSP) unit in the FPGA context, clocked at a frequency M times larger than the rest of the system. Designers typically apply the technique at RTL by manually inserting the custom logic to share the resource and safely perform CDC.

In this work, we achieve multi-pumping at the task level by tuning only the high-level parameters of the tasks, in particular the pipeline initiation interval (II), i.e., the clock cycles between the start of successive pipeline executions, and the clock constraint at the task granularity, taking advantage of the MCDFG. The HLS resource sharing algorithm automatically builds the logic for sharing the resource within a dataflow task. At the same time, the inter-task FIFOs allow safe CDC. We focused on DSPs, which are critical in compute-intensive kernels and can run at high frequencies. However, the technique can multi-pump any shareable resource, including entire sub-functions.

For example, consider a 2D Convolution HLS kernel by Xilinx [10], implemented as an SCDFG, as shown in Fig. 1a, where the rectangular nodes and the arrows represent the tasks



(a) SCDFG (b) State-of-the-art `Filter2D` (c) Double-pumped `Filter2D`

Fig. 1. Task-level multi-pumping saves resources at equal throughput for HLS of dataflow graphs (DFGs). The `Filter2D` task from a 2D Convolution kernel [10] (a) is double-pumped (c) by doubling its clock frequency and II to save half of the multipliers of the single-clock solution (b).

and the FIFOs, respectively. At each iteration, the `Filter2D` task processes a convolution window of up to 15×15 elements, which requires computing 225 multiply and accumulate (MAC) operations bound to DSPs. Thus, an II of $1\,\text{cycle}$ requires 225 DSPs. On the other hand, scaling the II to $2\,\text{cycles}$ implies that a new pipeline iteration starts every two clock cycles. Therefore, the pipeline has two cycles to compute the 225 operations. Hence, thanks to resource sharing, the HLS binding allocates only $\lceil 225/2 \rceil = 113$ DSPs, each of which computes two MACs. Assume that we target a throughput of $250\,\text{MSa/s}$. With the state-of-the-art (SOTA) SCDFG flow (Fig. 1b), we set the clock frequency of the whole DFG, including the `Filter2D` task that allocates 225 DSPs at $250\,\text{MHz}$. On the other hand, with our multi-pumping approach (Fig. 1c), we optimize the `Filter2D` task by scaling its II to $2\,\text{cycles}$, to save half of the DSPs, and its clock frequency to $500\,\text{MHz}$, to preserve the throughput.

This paper proposes an area-minimization methodology that preserves the throughput via task-level multi-pumping for FPGA HLS designs described as DFGs. Its effectiveness is validated on open-source designs using the workflow shown in Fig. 2, which generates an optimized multi-pumped intellectual property (IP) block from C/C++ source code using SOTA Xilinx commercial tools [11].

To the best of our knowledge, this is the first work that combines multiple clock domains with resource sharing in HLS of DFGs for task-level multi-pumping. The empirical results show that a new Pareto front opens in the power, performance, and area (PPA) space, with circuits that use up to $60\,\%$ fewer resources at maximum throughput or achieve up to $50\,\%$ higher throughput with the same resources.

II. Related work

Our work is mainly related to QoR improvement of HLS designs by tuning the HLS directives (i.e., the instructions for the HLS compiler to control hardware optimizations such as loop pipelining), focusing on multi-clock designs.

Several works [12]–[16] optimize for performance the HLS directives applied to plain software code not intended for HLS via design-space exploration (DSE). However, the goals of their works differ from ours since we optimize for resources while preserving the throughput of source code already optimized for HLS. In addition, our methodology avoids time-consuming DSEs and analytically computes the multi-pumping factor and, consequently, the corresponding II and clock frequency constraints. Finally, they all consider only single clock designs, except for Liang *et al.* [16] (discussed further in Section II-A).

HLS design optimizations based on multiple clock domains work at the *operation level*, assigning domains at the low-level resource (e.g., adder or multiplier) granularity, typically during scheduling [17]–[19], or at the *task level*, assigning domains at function granularity (i.e., MCDFGs) [16], [20].

A. Operation-level multi-clock in high-level synthesis

Lhairech-Lebreton *et al.* [17] use multiple clock domains in HLS to reduce power consumption while preserving the throughput by halving the operating frequency of two-cycle operations. We instead focus on area and performance optimizations because power is only a secondary quality metric for FPGA designs after performance and area.

Canis *et al.* [18] and Ronak *et al.* [19] design double-pumped DSP modules and use them in HLS with custom resource-sharing algorithms. Theoretically, Xilinx Vitis HLS supports double-pumped MAC operations through user-callable functions from the `dsp_builtins` library, but it is undocumented and faulty [21]. Our approach produces similar results when double-pumping a task. However, our task-level solution does not require custom modules, changes to the HLS sharing algorithm, or changes to the source code. In addition, it can select multi-pumping factors greater than two, resulting in larger resource savings.

B. Task-level multi-clock in high-level synthesis

Ragheb *et al.* [20] focus on extending the LegUp HLS tool to support MCDFGs synthesis but leave the selection of the clock frequencies to a suboptimal, time-consuming profiling-based approach. Our work focuses instead on a general methodology for exploiting the multiple clock domains. The workflow we define for building MCDFGs, based on SOTA Xilinx tools, is just a means to apply our methodology.

Liang *et al.* [16] propose a DSE methodology for maximizing the throughput under area constraints for HLS of MCDFG designs. They iteratively push for performance the HLS loop directives applied to the bottleneck tasks. If a task is still a bottleneck after maximally pushing the directives (e.g., when the pipeline II constraint is $1\,\text{cycle}$), they relax the directives of every task, increase the clock frequency of the bottleneck task, and restart the procedure. The goal of our work is different since we minimize the area while preserving the throughput. The optimization approaches differ, too, since we optimize all the resource-intensive tasks independently of whether they are bottlenecks, and we never push the II constraints, which the HLS compiler may fail to meet (e.g., due to data dependencies).

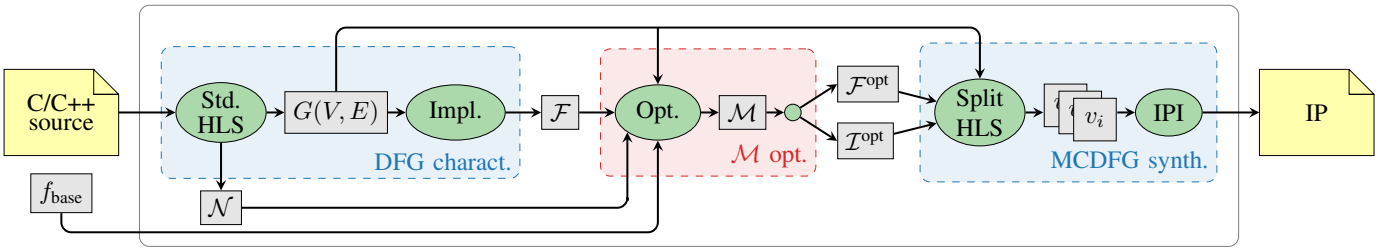


Fig. 2. Given the C/C++ source code of a dataflow graph (DFG) application and its base clock frequency, the proposed workflow builds the optimized multi-pumped IP by (a) analyzing the DFG (*DFG charact.*), (b) optimizing the multi-pumping factors ($\mathcal{M}$ *opt.*), and (c) synthesizing the multi-pumped IP (*MCDFG synth.*).

III. Background

Given the DFG throughput model defined in Section III-A, our multi-pumping methodology exploits the resource sharing executed by the HLS binding step to build the sharing logic. The relaxed timing mode for the HLS scheduling step ensures that the II of the pipelines is independent of the target clock frequency, as explained in Section III-B.

A. Dataflow graph

A DFG $G(V, E)$ is a set of tasks $v \in V$ running in parallel and communicating asynchronously through FIFO channels $e \in E$.

In HLS, each task is described as a C/C++ function whose core computational part typically consists of a pipelined loop. Given a task v_i clocked at frequency f_i and whose core loop is scheduled with initiation interval II_i, an approximation of its throughput is

$$\Phi_i := \frac{f_i}{II_i}. \tag{1}$$

The maximum external dynamic random access memory (DRAM) bandwidth can also limit throughput. However, this is out of the scope of our methodology since it does not change the overall throughput and, consequently, the DRAM bandwidth requirements.

The overall DFG throughput matches the one of the *bottleneck task* (i.e., the task with the lowest throughput)

$$\Phi_G := \min_{v_i \in V} \Phi_i. \tag{2}$$

According to (1), the high-level knobs for tuning the throughput of task v_i are its clock frequency (f_i) and initiation interval (II_i). All tasks share the same clock in single-clock dataflow graphs (SCDFGs). Thus, f_i is the same for all tasks and is bounded by the global (i.e., among all tasks) critical path. Therefore, only the II_i can be tuned independently for each task. In an MCDFG, on the other hand, the clock frequency can be set individually for each task. This additional degree of freedom allows for higher flexibility and tasks frequencies than SCDFG since the clock frequency of a task is limited only by its local critical path and not the one of the whole DFG.

B. High-level synthesis

Our multi-pumping technique relies on two key concepts of the HLS tools: (1) the minimum pipeline II is independent of the clock frequency constraint when the scheduler works in *relaxed timing* mode (i.e., the clock frequency is subordinate to meet the II constraints), and (2) the level of *resource sharing* is directly dependent on the II. Both are implemented by the HLS back-end that generates the hardware description. The timing model is used during *scheduling* and the resource sharing is done during *binding* [2].

1) Scheduling: Scheduling assigns operations to specific clock cycles; thus, it also implements loop and function *pipelining*. Designers can constrain the II of the pipelines, which is lower bound by the resource constraints and the data dependencies. Consider the data dependence graph (DDG) modeling the data dependencies in a kernel. Given a cycle θ in the DDG, we define $delay_\theta$ as the sum of the delays of the operations along θ and $dist_\theta$ as the total loop-carried dependence distance along θ. The lower bound of the II is

$$II^{\min} := \max_{\theta \in \text{DDG}} \left(\frac{delay_\theta}{dist_\theta} \right). \tag{3}$$

The associated cycle is called *critical* [22].

For example, consider the following loop to be scheduled:

```
for (int i = 0; i < N; i++)
    a = a + b;
```

The read-after-write dependency on `a`, produced at the i-th iteration and consumed at the $i+1$-th iteration, introduces a cycle θ in the DDG. $delay_\theta$ is the latency of the adder computing `a+b`. $dist_\theta$ is 1 since `a` is consumed at the iteration after it is produced. Therefore, (3) implies that the minimum II for this loop equals the latency of the adder.

The clock constraints determine how many operations fit within a clock cycle, thus affecting the depth of the pipelines. The pipeline depth determines the latencies of its operations, impacting the critical cycle and, in turn, the II lower bound. However, the II constraints take precedence over clock constraints in *relaxed timing* mode, yielding lower II pipelines in exchange for potential HLS timing violations. These are usually acceptable at HLS time since HLS timing estimations may be overly pessimistic [1], and downstream implementation steps may resolve them.

2) Binding: Binding assigns each operation to a compatible functional unit, depending on resource and performance (e.g., clock frequency, II) constraints.

Resource sharing is a crucial binding optimization that maps operations of the same type to the same functional unit, scheduled on different clock cycles or under mutually exclusive conditions (e.g., on different if-then-else branches). The II constraints directly affect the degree of resource sharing. In particular, if a pipeline scheduled with an II of II_i cycles computes N_i^{OP} operations (OPs) of the same kind at each iteration, the binding step allocates N_i^{FU} functional units (FUs), with

$$N_i^{\mathrm{FU}} := \left\lceil \frac{N_i^{\mathrm{OP}}}{\mathrm{II}_i} \right\rceil . \tag{4}$$

Note that the operations can be either computations or memory accesses. The functional units associated with the memory operations are ports proportional to the partitioning factors (i.e., the number of submemories into which a memory resource is divided to increase its parallelism). Therefore, larger II values result in fewer functional units and smaller memory partitioning factors.

Consider the `Filter2D` task from the 2D Convolution kernel introduced in Section I, whose source code is in Fig. 3a. Assuming a filter of size 2×2 (i.e., `FILTER_V_SIZE` = `FILTER_H_SIZE` $= 2$), with the schedule with an II of 1 cycle (shown in Fig. 3b, where the nodes represent the operations, and the edges the data dependencies), at the steady-state, four multiplications are computed in parallel on different data within the same clock cycle (highlighted by the red rectangle), thus requiring four DSP-mapped multipliers. With an II of 2 cycles instead (see Fig. 3c), only two multiplications are computed per clock cycle. Therefore, the binding step allocates only two multipliers and shares these among two multiplications.

IV. Task-level multi-pumping

We multi-pump the resources of task v_i by simultaneously scaling by a multi-pumping factor M_i the II and the clock frequency of v_i.

The underlying principles of our approach are:

- (2) allows tuning each task independently without reducing the overall DFG throughput, as long as the throughput of the task does not get lower than the bottleneck task one.
- As discussed in Section III-B2, scaling the II of a pipelined loop by a factor M_i allows reusing the same functional unit for M_i operations in different clock cycles.
- (1) implies that the task throughput is unchanged if we scale by M_i the task clock frequency together with the II.

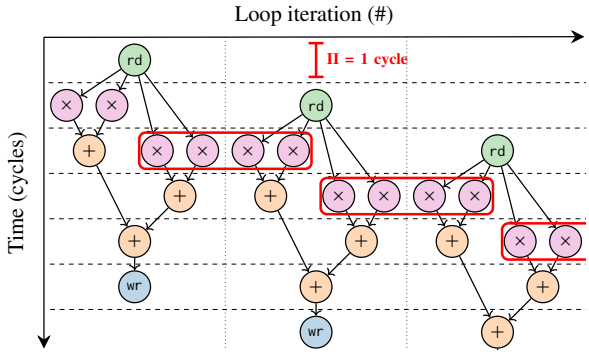
Assume that v_i meets the timing constraints up to $f_i^{\max}$ and computes N_i^{OP} operations mapped to DSPs. Moreover, the non-multi-pumped tasks are clocked at f_{base} (i.e., the clock constraint given by the designer). The maximum multi-pumping factor for task v_i is

$$M_i^{\max} := \min\left(\left\lfloor \frac{f_i^{\max}}{f_{\mathrm{base}}} \right\rfloor , N_i^{\mathrm{OP}} \right) . \tag{5}$$

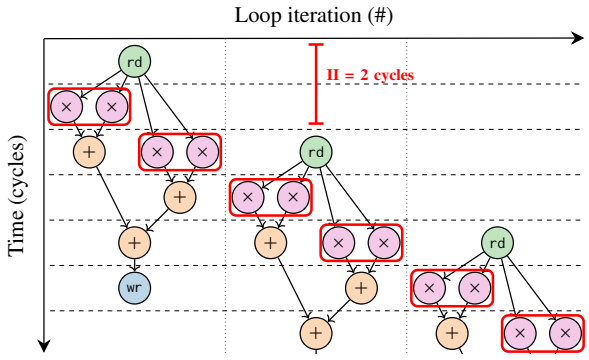
It is worth noting that our task-level multi-pumping *changes only the HLS directives while using the HLS tool as a black box and without requiring manual source code restructuring*. The automation of this step will be the subject of future work.

```
void Filter2D(hls::stream<window> &window_stream,
    hls::stream<char> &pixel_stream)
{
  for (int x = 0; x < width * height; x++) {
#pragma HLS PIPELINE II = II_Filter2D
    window w = window_stream.read();
    int sum = 0;
    for (int row = 0; row < FILTER_V_SIZE; row++) {
      for (int col = 0; col < FILTER_H_SIZE; col++)
        sum += w.pix[row][col] * coeffs[row][col];
    }
    pixel_stream.write(sum);
  }
}
```

(a) Source code



(b) Execution with II 1 cycle



(c) Execution with II 2 cycles

Fig. 3. The pipeline initiation interval (II) directly affects the resource sharing. For example, in the `Filter2D` task (a), the pipeline with II $= 1$ cycle (b) computes four multiplications per clock cycle in steady state, while the one with II $= 2$ cycles (c) only two. Thus, the latter datapath allocates half of the multipliers.

V. Multi-pumping workflow

To validate our task-level multi-pumping, we define a workflow from the C/C++ source code to an optimized MCDFG IP block compatible with Xilinx tools [11], as shown in Fig. 2. The main steps of the workflow are (A) *DFG characterization* to extract the maximum clock frequency and the number of DSP operations of each task, needed by the later steps, (B) *multi-pumping factor optimization* to select the multi-pumping factor of each task, and (C) *MCDFG synthesis* to generate the multi-pumped IP.

A. Dataflow graph characterization

For each task in the DFG $G(V, E)$, we collect the number of DSP operations ($\mathcal{N} = \{N_i^{\text{OP}}, \forall v_i \in V\}$) from the reports of the standard SCDFG HLS. We collect the maximum frequency meeting the timing constraints ($\mathcal{F} = \{f_i^{\max}, \forall v_i \in V\}$) from the post-implementation reports of the SCDFG. We execute the implementation with a tight clock constraint (e.g., $500\,\text{MHz}$) and at the lowest pipeline II, which is the worst case for the critical cycle (defined in Section III-B). Indeed, when multi-pumping increases the II, it relaxes the critical cycle, allowing deeper pipelines and shorter critical paths, thus higher clock frequencies.

We do not extract $\mathcal{F}$ from the earlier-available HLS clock frequency estimations since they are unreliable [1]. We run the SCDFG implementation only once, so the overhead is usually acceptable. However, if a fast flow is required (e.g., in early design phases), we can run only the logic synthesis step without placement and routing. The timing estimations at the logic synthesis step are more accurate than the one of the HLS compiler since they have access to lower-level information. When the estimated maximum frequency is less than the actual one, we miss chances of saving resources because of lower multi-pumping factors, as per (5). On the contrary, if the frequency is overestimated, the timing fails during implementation.

B. Multi-pumping factor optimization

We select the multi-pumping factors ($\mathcal{M} = \{M_i, \forall v_i \in V\}$) that minimize the DSP utilization. If v_i contains operations mapped to DSP, we set $M_i = M_i^{\max}$, as defined by (5). Otherwise, we do not apply multi-pumping to v_i.

C. Multi-clock dataflow graph synthesis

Xilinx Vitis HLS cannot synthesize MCDFGs directly since it supports only one clock domain per the design. However, the `dataflow` directive generates several independent modules, one for each task, and interconnects them in a top-level module. Thus, we run a *split* HLS, synthesizing each task separately (i.e., setting it as the top module) with its clock constraint.

The Xilinx HLS binding algorithm guarantees optimal resource sharing if guided by resource constraints only. Therefore, we constrain the number of DSPs according to (4). For instance, if we multi-pump with a factor M_i a task v_i that originally uses N_i^{DSP}, we constrain its DSPs to $\lceil N_i^{\text{DSP}}/M_i \rceil$.

In principle, we could also scale down the memory partitioning factors by M_i to reduce on-chip memory resource usage, namely block random access memories (BRAMs) and registers. However, we cannot apply this optimization to the test cases considered in Section VI with Xilinx HLS. Indeed, the tool ignores the coarser partitioning directives and automatically partitions the memories, presumably to minimize the pipelines II, regardless of the provided directives. We plan to revisit the issue as a more recent version of the HLS tool is available.

Finally, we interconnect the tasks synthesized separately using the Vivado intellectual property integrator (IPI).

The Xilinx HLS tools use FIFOs as inter-task communication channels when data are produced and consumed in the same order; otherwise, ping-pong buffers. Our method could support both, but since the Xilinx IPI flow does not provide a configurable multi-clock ping-pong buffer, we currently only support FIFO channels using the Xilinx FIFO generator [23]. FIFOs are configured with independent clocks for read and write ports when interconnecting tasks assigned to different clock domains.

VI. Evaluation

We verify the applicability and the advantages in the PPA space of our task-level multi-pumping workflow, described in Section V, by applying it to open-source HLS designs.

Our experiments target the embedded platform Zynq UltraScale+ FPGA SoC hosted by the Avnet Ultra96v1 board [24]. We use Vitis HLS 2022.2 [2] and Vivado HLS 2019.2 [25] for the synthesis and Vivado 2022.2 [11] for the implementation.

We collect the resource utilization from the post-implementation reports and the power estimations from the post-implementation static power analysis. We verify that the throughput (i.e., the number of output samples produced in the unit of time) matches the theoretical one by measuring the time for $10\,000$ executions in auto-restart mode [2] (to make the time overhead for control negligible) of the kernels in hardware, using the PYNQ application programming interfaces [26].

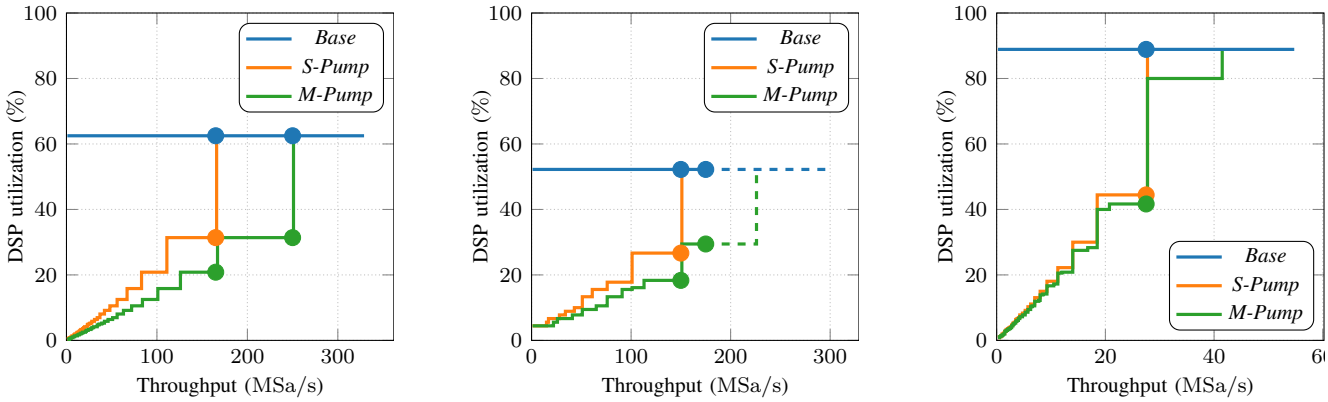
We apply our flow to some open-source HLS designs, including (a) the *2D Convolution* from the Vitis Tutorials [10] already introduced in Section I, (b) the *Optical Flow* from the Rosetta suite [8], and (c) the *virtual molecule screening (VMS)* [27], a drug discovery accelerator.

For each design, we compare the multi-pumped implementations (*M-Pump*) with the original ones (*Base*) and with the best SCDFG implementations without source code changes (*S-Pump*). For the *S-Pump* implementations, we apply our flow without the generalization to MCDFG. Thus, if task v_i is "single-pumped" by a factor S_i, we scale by S_i its II, as with our original workflow, and the clock frequency of the whole kernel. The maximum "single-pumping" factor for each task is lower than the corresponding maximum multi-pumping factor (defined by (5)) since it is at most

$$S_i^{\max} := \left\lfloor \frac{\min\limits_{\forall v_i \in V} f_i^{\max}}{f_{\text{base}}} \right\rfloor . \tag{6}$$

Figure 4 shows the tradeoffs between the DSP utilization and the throughput obtained by varying the base clock frequency within the range allowed by the critical path of the designs. The dashed lines represent computation throughputs that exceed the memory throughput. Thus, the effective throughputs are, in practice, clipped to the maximum non-dashed value, corresponding to the maximum memory throughput.

The number of DSPs used by *Base* designs is independent of the clock frequency. The plots of the *Pump* designs are characterized by a step shape, whose discontinuities correspond to the IIs changes, which only assume integer values. The *Pump* solutions provide different tradeoffs in the throughput



(a) 2D Convolution (b) Optical Flow (c) VMS

Fig. 4. Digital signal processors (DSPs) allocated for a given throughput. The *M-Pump* designs are optimized using the proposed task-level multi-pumping technique. The *M-Pump* designs are Pareto-optimal compared to the *Base* designs, whose DSP utilization is constant since they are optimized by tuning the clock frequency only, and to the *S-Pump* designs, which are optimized for area by changing both the II and the global clock frequency of the tasks. The dashed lines represent the theoretical throughputs achievable with the allocated DSPs, which are unreachable in practice due to memory bandwidth limitations. The dots show the design points implemented in hardware.

TABLE I
POWER, PERFORMANCE, AND AREA OF THE BENCHMARKS TARGETING A ZYNQ ULTRASCALE+ SOC

Design	**Throughput** (MSa/s)	**Implem.**	**Base clock** (MHz)	**Pump factors** (1)	**DSP** (%)	**LUT** **Logic** (%)	**LUT** **Memory** (%)	**FF** (%)	**BRAM** (%)	**Power** **Static** (W)	**Power** **Dynamic** (W)	**Clock routing** (%)
2D Convolution [10]	165	Base	165	–	64	14	12	9	4	0.3	1.8	1.0
		S-Pump	330	2	33	15	12	18	4	0.3	2.3	1.0
		M-Pump	165	3	**23**	14	12	18	4	0.3	2.2	2.4
	250	Base	250	–	64	13	12	10	4	0.3	2.0	1.0
		M-Pump	250	2	**33**	15	12	19	4	0.3	2.8	2.4
Optical Flow [8]	150	Base	150	–	55	36	65	23	20	0.3	2.5	1.0
		S-Pump	300	2	29	37	65	26	20	0.3	3.2	1.0
		M-Pump	150	2, 3	**21**	37	64	27	20	0.3	3.0	3.8
	175	Base	175	–	55	36	65	23	20	0.3	2.5	1.0
		M-Pump	175	2	**33**	38	65	27	20	0.3	2.9	2.4
VMS [27]	28	Base	110	–	89	32	23	31	67	0.3	2.0	1.0
		S-Pump	220	2	44	30	23	31	67	0.3	2.4	1.0
		M-Pump	110	2, 3	**42**	32	23	37	67	0.3	2.9	3.8

versus DSP space, thanks to the tuning of the pipelines' II. The additional degree of freedom of the *M-Pump* implementations (i.e., the task clock frequency) makes them always Pareto optimal.

Both *M-Pump* and *S-Pump* designs degenerate to *Base* designs (i.e., all the pumping factors to one and no resource savings) at the highest throughputs since they need the lowest IIs to reach the best performance. Note that the *M-Pump* designs consistently degenerate to *Base* at throughputs higher than *S-Pump* since the multiple clock domains let the multi-pumped tasks run at the maximum frequency their local critical path allows. Therefore, the *M-Pump* designs achieve up to $52\,\%$ higher throughput than *S-Pump* with the same DSPs in the 2D Convolution test case. Moreover, with the Optical Flow benchmark, the *M-Pump* reaches the maximum effective throughput using $40\,\%$ fewer DSPs than the *Base*.

Table I reports the post-implementation PPA data for the design points marked with the dots in Fig. 4. We select those points since their throughputs are the upper extremes of the last steps of *M-Pump* and *S-Pump* within the memory bound.

Comparing the *M-Pump* designs with the *Base* ones, the consistent DSP saving ($54\,\%$ on average) implies power and flip-flops (FFs) overheads. The additional power ($24\,\%$ on average) is because the multi-pumped tasks are characterized by greater switching activity due to higher resource reuse and clock frequencies. The additional FFs ($33\,\%$ on average) are inserted by the HLS tool in the multi-pumped tasks to build deeper pipelines and reach higher clock frequencies.

As expected [9], the PPA overhead for CDC in *M-Pump* is negligible. The overhead for routing multiple clocks is also marginal, as each additional clock domain allocates only $1.4\,\%$ of the available clock routing resources.

In general, the *M-Pump* solutions Pareto dominate the *S-Pump* ones. In fact, at the same throughput, they allocate fewer

DSPs, similar look-up tables (LUTs) and FFs, and consume less power. This is because the *M-Pump* designs take advantage of the multiple clock domains to increase the clock frequency of the multi-pumped tasks only, thus reaching higher multi-pumping factors and avoiding power and FF overheads in the non-multi-pumped tasks. The VMS test case is the only exception because only a small fraction of its logic runs at the base clock frequency, while the rest is double or triple-pumped; thus, the lower-frequency tasks are not enough to balance the power and FF overhead for the multi-pumped tasks.

VII. Conclusion

We propose a task-level multi-pumping technique for saving hardware resources while maintaining the original throughput for HLS dataflow designs for FPGAs.

Given an SOTA single-clock DFG, our approach first generalizes it to a multi-clock DFG. Secondly, it tunes the tasks' high-level parameters (i.e., clock frequency and pipeline II) to multi-pump their functional units. The overhead for generalization is negligible, thanks to the DFGs structure, which consists of independent blocks communicating via FIFOs, allowing for safe CDC, and modern FPGA clock architectures, which seamlessly handle multiple clock domains even if current HLS tools do not exploit them.

The experimental results reported in Section VI prove that our method opens a new Pareto front in the performance versus DSPs space, saving up to $40\,\%$ of resources at maximum throughput. Moreover, our method does not require any manual architecture changes from the designer, since it acts only on the high-level parameters of the tasks and uses the HLS binding algorithm to automatically generate the resource sharing logic. Finally, the generalization to multi-clock DFGs simply requires replacing single-clock with multi-clock FIFOs. Therefore, our technique is well suited for a fully automated HLS optimization pass, which will be the subject of future work.

Acknowledgment

This work was partially supported by the Key Digital Technologies Joint Undertaking under the REBECCA Project with grant agreement number 101097224, receiving support from the European Union, Greece, Germany, Netherlands, Spain, Italy, Sweden, Turkey, Lithuania, and Switzerland.

References

[1] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS Today: Successes, Challenges, and Opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, pp. 1–42, Aug. 2022. DOI: 10.1145/3530775.

[2] Xilinx, *Vitis High-Level Synthesis User Guide*, Dec. 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls.

[3] Siemens EDA, *Catapult® Synthesis HLS Bluebook*, Apr. 2021. [Online]. Available: https://resources.sw.siemens.com/en-US/e-book-high-level-synthesis-hls-blue-book.

[4] Intel, *Intel® High Level Synthesis Compiler Pro Edition Reference Manual*, Dec. 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683349/22-4/pro-edition-reference-manual.html.

[5] Xilinx, *Vitis Accelerated Libraries*, 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/Vitis_Libraries.

[6] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Int. Symp. Field-Prog. Gate Arrays*, 2017, pp. 65–74. DOI: 10.1145/3020078.3021744.

[7] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis," in *Int. Symp. Field-Prog. Gate Arrays*, 2020, pp. 244–254. DOI: 10.1145/3373087.3375296.

[8] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs," in *Int. Symp. Field-Prog. Gate Arrays*, 2018, pp. 269–278. DOI: 10.1145/3174243.3174255.

[9] Xilinx, *Resource Utilization for FIFO Generator v13.2*, 2022. [Online]. Available: https://www.xilinx.com/cgi-bin/docs/ndoc?t=ip+ru;d=fifo-generator.html.

[10] ——, *Design and Analysis of Hardware Kernel Module for 2-D Video Convolution Filter*, May 2022. [Online]. Available: https://xilinx.github.io/Vitis-Tutorials/2022-1/build/html/docs/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/README.html.

[11] ——, *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973)*, Oct. 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug973-vivado-release-notes-install-license/Release-Notes.

[12] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, no. 4, pp. 1–27, Feb. 2022. DOI: 10.1145/3494534.

[13] Q. Sun, T. Chen, S. Liu, J. Miao, J. Chen, H. Yu, and B. Yu, "Correlated Multi-objective Multi-fidelity Optimization for HLS Directives Design," in *Design Autom. Test Eur. Conf. Exhib.*, 2021, pp. 46–51. DOI: 10.23919/DATE51398.2021.9474241.

[14] Xilinx, *Merlin Compiler*, 2022. [Online]. Available: https://github.com/Xilinx/merlin-compiler.

[15] C. Lo and P. Chow, "Model-based optimization of High Level Synthesis directives," in *Int. Conf. Field-Prog. Logic Appl.*, 2016, pp. 1–10. DOI: 10.1109/FPL.2016.7577358.

[16] T. Liang, J. Zhao, L. Feng, S. Sinha, and W. Zhang, "Hi-ClockFlow: Multi-Clock Dataflow Automation and Throughput Optimization in High-Level Synthesis," in *Int. Conf. Comput.-Aided Des.*, 2019, pp. 1–6. DOI: 10.1109/ICCAD45719.2019.8942136.

[17] G. Lhairech-Lebreton, P. Coussy, and E. Martin, "Hierarchical and Multiple-Clock Domain High-Level Synthesis for Low-Power Design on FPGA," in *Int. Conf. Field-Prog. Logic Appl.*, 2010, pp. 464–468. DOI: 10.1109/FPL.2010.94.

[18] A. Canis, J. H. Anderson, and S. D. Brown, "Multi-pumping for resource reduction in FPGA high-level synthesis," in *Design Autom. Test Eur. Conf. Exhib.*, 2013, pp. 194–197. DOI: 10.7873/DATE.2013.053.

[19] B. Ronak and S. A. Fahmy, "Multipumping Flexible DSP Blocks for Resource Reduction on Xilinx FPGAs," *IEEE Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 36, no. 9, pp. 1471–1482, Sep. 2017. DOI: 10.1109/TCAD.2016.2629421.

[20] O. Ragheb and J. H. Anderson, "High-Level Synthesis of FPGA Circuits with Multiple Clock Domains," in *Int. Symp. Field-Prog. Custom Comput. Mach.*, 2018, pp. 109–116. DOI: 10.1109/FCCM.2018.00026.

[21] Xilinx, Ed. "double pumping DSP." (2019), [Online]. Available: https://support.xilinx.com/s/question/0D52E00006iHilaSAC/double-pumping-dsp (visited on 08/31/2023).

[22] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 367–432, Sep. 1995. DOI: 10.1145/212094.212131.

[23] Xilinx, *FIFO Generator*, 2022. [Online]. Available: https://www.xilinx.com/products/intellectual-property/fifo_generator.html.

[24] Avnet, *Ultra96 Hardware User's Guide*, Mar. 2018. [Online]. Available: https://www.avnet.com/opasdata/d120001/medias/docus/187/Ultra96-HW-User-Guide-rev-1-0-V0_9_preliminary.pdf.

[25] Xilinx, *Vivado Design Suite User Guide High-Level Synthesis*, Jan. 2020. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls.

[26] ——, *PYNQ: Python productivity for Adaptive Computing platforms*, 2022. [Online]. Available: https://pynq.readthedocs.io.

[27] T. V. Aa, T. Haber, T. J. Ashby, R. Wuyts, and W. Verachtert, *Virtual Screening on FPGA: Performance and Energy versus Effort*, 2022. arXiv: 2210.10386.