# Hi-ClockFlow: Multi-Clock Dataflow Automation and Throughput Optimization in High-Level Synthesis

*(Invited Paper)*

Tingyuan Liang[*], Jieru Zhao[*], Liang Feng[*], Sharad Sinha[†] and Wei Zhang[*]
[*]Department of Electronic and Computer Engineering
The Hong Kong University of Science and Technology, Hong Kong
Email: tliang@connect.ust.hk, jzhaoao@connect.ust.hk, lfengad@connect.ust.hk, wei.zhang@ust.hk
[†]Computer Science and Engineering Department
Indian Institute of Technology (IIT), Goa, India
Email: sharad_sinha@ieee.org

***Abstract*—Tools of high-level synthesis (HLS) are developed to improve the accessibility of FPGAs by allowing designer to describe hardware designs in high-level language, e.g. C/C++. However, the source codes of general applications are not structured as canonical dataflow. Furthermore, clock frequencies are powerful parameters to improve dataflow throughput but currently commercial HLS tools limit themselves to single clock domain. Consequently, in order to benefit from the multiple-clock dataflow design, designers still suffer from manually analyzing the applications, partitioning the source code into modules, optimizing them with appropriate parameters and resource allocation, and finally interconnecting them. We analyze the impact of multiple clock domains for HLS designs and present Hi-ClockFlow, an automatic HLS framework. Hi-ClockFlow can analyze the source code based on Light-HLS, our light weight HLS evaluation framework, explore the large design space, and optimize such parameters as clock frequencies and HLS directives in dataflow. By properly partitioning the source code of an application into parts with various clock domains, Hi-ClockFlow can optimize the dataflow with imbalanced modules and speed up the performance under the specific constraint of resource.**

***Index Terms*—High-Level Synthesis, Dataflow, Multiple Clock Domains, Globally Asynchronous Locally Synchronous, Automation, Design Space Exploration**

## I. Introduction

Field-programmable gate array (FPGA) can achieve high throughput, low latency and less energy overhead for many applications, especially such dataflow designs as image processing and machine learning. High-level synthesis (HLS) is proposed to improve the accessibility of FPGAs by employing high-level language, e.g. C/C++, to describe hardware designs. Moreover, FPGA provides much more flexibility, including the configuration of multiple clocks and different levels of parallelism, for designers to achieve their optimal solutions.

To implement dataflow in HLS, designers need to first partition the source code into modules so the task-level pipeline can be realized by HLS tools. Then, proper HLS directives and clock periods should be assigned to the modules to optimize the overall throughput. The resource and timing of modules are sensitive to the clock period. Consequently, multiple-clock designs allow the modules in dataflow to do a more flexible trade-off between resource and performance.

Nevertheless, currently commercial HLS tools still require designers to manually analyze the applications and assign resource and clock domains to the modules in dataflow. Furthermore, the tools limit themselves to single clock domain and hence, it is hard for dataflow designs to benefit from the multiple clock domains. For example, some HLS directives may be incompatible with the given timing constraint and the scheduling and binding of the design will change significantly as the clock frequency changes.

To expand the design space and find the optimal solution to multiple-clock dataflow, we present Hi-ClockFlow, an open-source automatic HLS framework, which is coupled with Vivado and VivadoHLS, the platforms developed by Xilinx. Hi-ClockFlow can analyze and transform the source code of an application into partitions with various clock domains properly. The partitions will co-operate as globally-asynchronous locally-synchronous (GALS) dataflow. In brief, Hi-ClockFlow is to conduct HLS design space exploration (DSE) to find the proper settings of multiple clock domains and HLS directives to optimize the throughput of the dataflow application. To our best knowledge, Hi-ClockFlow is the first framework which realizes the automatic optimization for multiple-clock HLS dataflow design on FPGA, with its highlights shown below:

1) General HLS C/C++ source code of application is the input of Hi-ClockFlow, which will be analyzed by Light-HLS, our light weight HLS evaluation framework.
2) HLS directives for each module can be automatically set by Hi-ClockFlow to optimize the throughput and the consumption of resource.
3) Clock domain for each module is determined by Hi-ClockFlow to achieve high throughput.

## II. Related Works

Conventionally, the previous works in multiple-clock designs are in RTL level and mainly focusing on high-performance FIFOs implementation, handshake protocols or other device level implementations. For example, in the GAPLA architecture [1], the size and shape of each locally synchronous block are programmable and data communications between synchronous blocks are controlled by 2-phase handshaking signals. In [2], Royal et al. proposed to use GALS techniques for FPGAs to overcome the limitation on timing imposed by slow routing.

As for multiple-clock designs in HLS, there are also some works proposed using GALS technology. LegUp [3], an open-source HLS tool, was extended to automatically insert clock-domain-crossing circuitry for signals crossing between two domains. The scheduling and binding phases of HLS are changed to reflect the impact of multiple clock domains on memory. However, they mainly focus on the implementation of the clock domain crossing (CDC) interface, based on Block-RAMs, and the related problem of scheduling and binding. The exploration of parallelism in multi-clock situation is not considered. Furthermore, the selection of frequency of each clock domain is left to designers and the assignment of clock domains is coarse-grained. Mamaghani et al. developed a series of platforms [4] [5] to support asynchronous dataflow and proposed the technique [6] to handle the clock automatically. Nevertheless, in their multiple clock designs, clocks are simply set based upon the ”local” critical path delays and they assumed that the change of frequency would not change the scheduling and binding of HLS components and common HLS directives are not applicable.

In order to optimize the design performance through DSE, multiple clock domains in the dataflow design bring different challenges. Zhao et al. proposed a comprehensive model-based analysis framework, COMBA [7], which selects suitable HLS directives and aims at lowering the overall latency of an application given certain FPGA resource constraints. Different from COMBA, our framework can automatically partition the source code into different clock domains and conduct DSE for all the partitions to optimize the overall throughput. Cong et al. brought a model targeting at a well-defined accelerator microarchitecture [8] and thus it featurs a highly accurate modeling of the utilization of the FPGA on-chip resources, while our framework targets at more general C/C++ descriptions. Shao et al. implemented Aladdin [9], a pre-RTL light weight HLS tool, as a power-performance accelerator simulator for System-on-Chip architecture exploration. However, Aladdin has set many assumptions for the HLS design, for example, not including DSP and the delay of function units is always one cycle, hence, the simulation may not be accurate enough. Moreover, all the above frameworks do not support multi-clock designs. Compared to them, Hi-ClockFlow focuses on the optimization of multi-clock dataflow.

Considering dataflow optimization itself, there are several papers optimizing from different angles. Li et al. [10] [11] developed an algorithm to find the optimal resource usage and initiation intervals for each loop in the applications to achieve maximum throughput under a given area constraint. However, loop unrolling, array partitioning and dataflow were not considered in their work, which means that under their assumption, their design space of application is relatively small and the throughput might not be the optimal one under the specific resource constraint. Finally, their integer linear programming based solution might not be suitable for large-scale applications. There are also application-specific frameworks for some applications like CNN [12], However, those frameworks are limited in specific domains.

## III. Multiple Clock Domains in HLS-Based Dataflow

For single clock design, commercial HLS tools, like Xilinx VivadoHLS, can enable task-level pipelining by allowing functions and loops to overlap in their operation and increasing the overall throughput of the design. However, they do not allow designs to be configured with multiple clock domains. In this section, the mechanism how we make multi-clock dataflow work will be shown and the impact of multiple clock domains in HLS-based dataflow will be discussed as the motivation of this work as well as the corresponding challenges.

### *A. Mechanism of Multi-Clock Dataflow for Hi-ClockFlow*

In dataflow designs, each module, having relatively independent functionality, will consume data generated by its predecessors and produce data for its successors. The interconnections between the modules are synchronizers (CDC channels) to handle the interaction and transfer data. Compared to those single-clock designs, the multi-clock designs have general modules, running in different clock domains, but special interconnection among them.

In most dataflow applications, usually the synchronizers for the data transferring are FIFOs and Ping-Pong buffers. FIFOs can store data in a sequential way without address so they can be implemented with less resource but the data in FIFOs have to be processed sequentially. In contrast, Ping-Pong buffers can store data according to address but they have to work in pairs, which take turns to be the output buffer of the predecessor and the input buffer of the successors. In order to handle different applications, which might not have streaming memory access pattern, in Hi-ClockFlow, the modules are generated with Ping-Pong interfaces via common

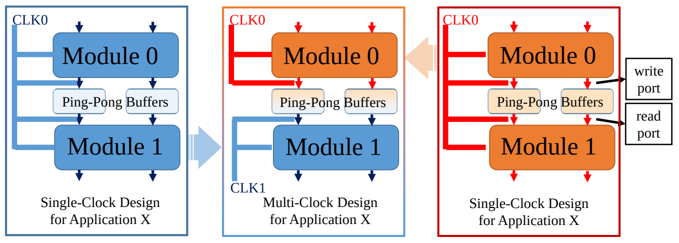

Fig. 1. Example for the Mechanism of Multi-Clock Dataflow Based on Hi-ClockFlow

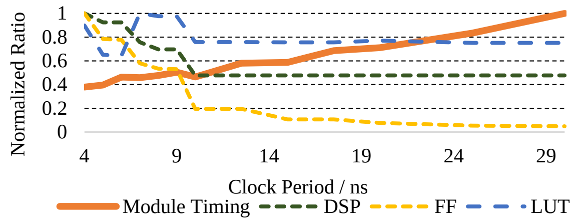

Fig. 2. Example Based on 2mm for Relationship among Clock Frequency, Module Timing and Resource Cost in HLS: The resultant values are normalized and the definition of timing is runtime clock cycles times clock period.

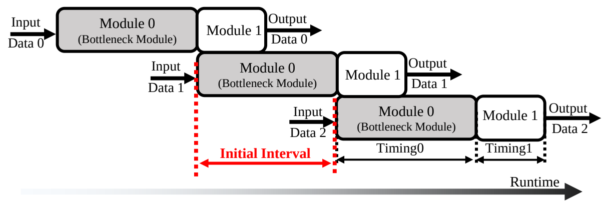

Fig. 3. Example based on 2mm: the initial interval (II) and throughput are determined by the bottleneck module in dataflow, i.e. II=Timing0.

HLS procedures under different timing constraints. Originally, Ping-Pong buffers in commercial HLS tools like VivadoHLS are connected to a shared clock. To meet the requirement of multi-clock designs, in the design based on Hi-ClockFlow, the clocks for the write ports of Ping-Pong buffers will be connected to the one for their predecessor, while the clocks for the read ports of Ping-Pong buffers will be connected to the one for their successor.

As an example of Hi-ClockFlow hardware implementation is shown in Fig. 1. The application can be synthesized under single clock constraint, as shown on the left and right side in the figures. Modules with Ping-Pong buffer interface in single-clock circuits will be extracted and interconnected in a multi-clock design, as shown in the middle, and some of the original Ping-Pong buffers will be changed into dual-clock buffers.

Hi-ClockFlow targets on the DSE of the setting of clocks and HLS directives for multi-clock dataflow, while the optimization of CDC mechanism will be included in Hi-ClockFlow in the future.

## *B. Impact and Challenges of Multiple Clock Domains in HLS-Based Dataflow*

When changing clock frequency, many designers suffer from tuning the parameters of HLS directives and one of the reasons is that some combinations of HLS directives, e.g. small initial interval for loop pipelining or large-scale array partitioning, are not suitable for the specific clock period and HLS tools cannot meet the timing constraints. As result, raising clock frequency might not optimize the performance of the design, e.g. initial interval for loop pipelining might need to be increased. However, in [7], [8] and [9], this situation was not considered which makes some of their resultant solutions not applicable for higher frequency. However, both increasing clock frequency and raising parallelisms with HLS directives are common solutions to performance of applications.

To overcome this problem, we develop an open-source light weight HLS evaluation framework, Light-HLS, presented in Section IV, which provide APIs for Hi-ClockFlow to evaluate the performance and resource of the input design and collect various information. With the analysis of Light-HLS, we can make sure that the combinations of HLS directives and clock period generated by Hi-ClockFlow can be realized in HLS tools. During the evaluation of the designs, Light-HLS will inform Hi-ClockFlow whether the HLS directives can be achieved under the specific timing constraint.

Apart from the compatibility between frequency and HLS directives, the balance of performance and resource among dataflow modules is also important. In HLS designs, the clock frequency can significantly impact the performance and resource cost, because under different timing constraints, the HLS tools will try to find various approaches to improve the performance of the design.

The scheduling of operations mainly depends on: (1) clock frequency; (2) latency of the operation, as defined by the target device; (3) user-specified HLS directives. As for binding of resource, if the clock period is longer, more operations could be completed within a single clock cycle and fewer resource will be cost, but unnecessary low clock frequency may undermine the performance. Conversely, if the clock period is shorter, HLS tools might automatically schedule the operations over more clock cycles, and some operations might need to be implemented with more resources. In Fig. 2, we take benchmark 2mm from PolyBench [13] as an example of such trade-off, by fixing the other configurations while only changing the clock period. Properly raising clock frequency may improve the performance of the design but extremely high clock frequency will result in high overhead of resource, which might be not necessary.

As for the modules in a dataflow design, some of them could be the bottlenecks of the throughput. As an example based on benchmark 2mm in Fig. 3, there are two top-level loops, extracted into two modules, Module 0 and Module 1. The inner body of Module 0 contains more operations than Module 1. Raising the clock frequencies and parallelism of the bottleneck, e.g. Module 0, can help to optimize the overall throughput. In contrast, the clock frequencies and parallelism of the modules, e.g. Module 1, which do not restrain the throughput, can be lowered to save resource.

The challenge for Hi-ClockFlow is to find the setting for clock domains and HLS directives for each module in dataflow to optimize throughput under the resource constraint. The critical problem is that we allow each module to have flexible clock domain and it will enlarge the design space. Supposing that for each module, the number of the potential choices of clock period is 10, the design space of Hi-ClockFlow is $10^N$ larger than the one of COMBA, where $N$ is the number of modules in dataflow. Therefore, Hi-ClockFlow needs to design a high-efficiency DSE algorithm to handle the large design space at the whole system view, which will be discussed in Section V.

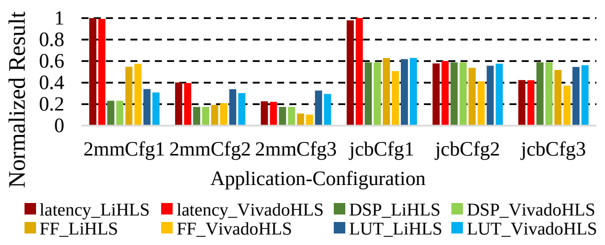

Fig. 4. Light-HLS Validation: The results are normalized for comparison

## IV. Light-HLS: Portable Light Weight HLS Framework

We develop Light-HLS to provide APIs for Hi-ClockFlow to evaluate the performance and resource of the input design and collect various information, with fast speed and high accuracy.

Light-HLS is a light weight HLS framework, which can be called to perform general DSE for FPGA-based HLS design. It covers the abilities of previous works, overcomes the existing limitations and brings more practical features. Light-HLS is modularized and portable so designers can use the components of Light-HLS to conduct various DSE procedures. Light-HLS gets rid of RTL code generation so it will not suffer from the time-consuming synthesis of commercial HLS tools like VivadoHLS, which involves many detailed operations in both its frond-end and back-end.

### A. *Implementation*

Light-HLS, based on LLVM-9.0, consists of front-end and back-end. The front-end of Light-HLS accounts for IR generation and a series of IR optimizations while the back-end will schedule the IR instructions and accomplish hardware resource binding.

*1) Light-HLS Frond-End:* The goal of Light-HLS frond-end is to generate IR code close enough to those generated via commercial tools, like VivadoHLS, for DSE purpose. In the front-end of Light-HLS, initial IR codes generated via Clang will be processed by HLS optimization passes consisted of three different levels: (a) At instruction level, Light-HLS modifies, removes or reorders the instructions, e.g. reducing bitwidth, removing redundant instruction and reordering computation. (b) At loop/function level, functions will be instantiated and loops may be extracted into sub-functions. (c) As for memory access level, redundant load/store instructions will be removed based on dependency analysis.

*2) Light-HLS Back-End:* The back-end of Light-HLS is developed to schedule and bind for the optimized IR codes, so it can predict the resultant performance and resource cost accurately based on the given settings. The IR instructions can be automatically characterized by Light-HLS and a corresponding library, which records the timing and resource of different types of operations, will be generated. For scheduling, based on the generated library, Light-HLS maps most operations to corresponding cycles based on as-soon-as-possible (ASAP) strategy. For some pipelined loops, the constraints of the port number of the BRAMs and loop-carried dependencies are considered. Moreover, some operations might be scheduled as late as possible (ALAP) to lower the II. As for binding, Light-HLS accumulates the resource cost by each operation and the chaining of operations is considered. The reusing of hardware resource is an important feature in HLS and Light-HLS reuses resources based on more detailed rules, e.g. the source and type of input.

### B. *Validation*

We validate Light-HLS against benchmarks from PolyBench [13] by comparing its predictive performance and resource with the evaluation results from commercial tools, i.e. VivadoHLS.

We randomly generate design configurations, including HLS directives and clock frequency, and run Light-HLS and VivadoHLS for the benchmarks. Some of the comparison results, based on benchmark 2mm and jacobi-2d, with different memory access patterns and different data types, are shown in Fig. 4. The relative error of performance evaluation of Light-HLS is 3.1% averagely and limited within 8.3%. The relative errors of resource evaluation of Light-HLS are limited within 2.7%, 17.9%, 10.1% and 0.0% for DSP, FF, LUT and BRAM respectively. The relatively large error of FF estimation maybe comes from that Light-HLS does not consider some limitations in RTL generation. Since FFs are usually abundant on FPGA, this relative error is acceptable.

## V. Algorithm of Design Space Exploration

### A. *Problem Formulation*

In the design space, the application is described as dataflow, which consists of multiple modules. Each module can has its own setting of HLS directives, i.e. parallelism, and clock frequency. Hi-ClockFlow will find the proper setting for each module to maximize the overall throughput of the application while meeting the resource constraint of the target FPGA device and the number constraint of clock domains. Noticeably, as mentioned in Section III, the throughput is determined by the slowest module in the dataflow, i.e. *bottleneck module*, and therefore, the problem can be formulated as minimizing a maximum, which is shown below:

$$\begin{aligned} \underset{f_i,L_i,A_i}{\text{minimize}} \quad & max(Timing(M_i,f_i,L_i,A_i)),\ i=1,\ldots,N. \\ \text{subject to} \quad & \sum Util(M_i,f_i,L_i,A_i) \le 1,\ i=1,\ldots,N. \\ & A_1 = A_2 = \ldots = A_N,\ i=1,\ldots,N. \\ & |\{f_1,...,f_N\}| \le C_{tot},\ i=1,\ldots,N. \end{aligned}$$

where $N$ is the number of modules in the dataflow, $C_{tot}$ is the limitation of the number of clock domains, for each module, $M_i$ is its IR code, $f_i$ is its clock frequency, $L_i$ is the setting of loops in it, e.g. loop unrolling and loop pipelining, $A_i$ is the setting of arrays in it, i.e. array partitioning, $Timing(M,f,L,A)$ is the function of module timing, i.e. runtime cycle number times clock period, $Util(M,f,L,A)$ is the function of module resource utilization, which is a linear combination of the utilization of different resources. The definition of $A_i = A_j$ is that if both $A_i$ and $A_j$ set

**Algorithm 1** Pushing-Relaxation DSE

```
Require:
  1. Application Source Code
  2. Resource Constraint and Clock Constraint
Ensure:
  1.Config$_{best}$ : HLS directives and clocks for modules
  2.II$_{best}$ : the lowest achieved II

  Config ← Reset()
  while true do
    pushSuccess ← true
    while pushSuccess do
      Mod$_{push}$ ← BotteleneckModule(Modules,Config)
      if $\text{II}_{best} > \text{Timing}(\text{Mod}_{push}, \text{Config})$ then
        $\text{II}_{best} \leftarrow \text{Timing}(\text{Mod}_{push}, \text{Config})$
        Config$_{best}$ ← Config
      end if
      $\text{Config}, \text{pushSuccess} \leftarrow \text{Push}(\text{Mod}_{push}, \text{Config})$
    end while
    Mod$_{relax}$ ← BotteleneckModule(Modules,Config)
    Clk ← FindNextClock(Mod$_{relax}$, Config)
    if Clk is NULL then
      break;
    else
      $\text{Config} \leftarrow \text{Relax}(\text{Config}, \text{Mod}_{relax}, \text{Clk})$
    end if
  end while
  return  $\text{II}_{best}, \text{Config}_{best}$
```

array partitioning for the same target array, the configuration of array partitioning is the same. The purpose of this constraint of array partitioning is to ensure the feasibility of parallel data transferring for partitioned arrays between modules. In the hardware implementation for the multi-clock designs based on Hi-ClockFlow, the CDC data transferring is handled by Ping-Pong buffers (BRAMs) so it will not be the performance bottleneck of the dataflow and the throughput will be determined by the *bottleneck module*.

## *B. Pushing-Relaxation Heuristic Algorithm*

First, Hi-ClockFlow will flatten the entire application into modules based on their granularity. In current version of Hi-ClockFlow, each module mainly consists of a top-level loop and its peripheral regions. Secondly, the iterative optimization of the bottleneck in the dataflow will start from an given initial configuration, where all the modules share single slow clock domain and there is no HLS directive set for the modules.

The iterative heuristic optimization consists of two parts: pushing and relaxation, as shown in Algorithm 1.

**Procedure of Pushing:** The procedure of pushing is to improve the performance of the *bottleneck module* by setting proper HLS directives. When pushing a module, with a given setting of clock domain for the module, Hi-ClockFlow will find a configuration of HLS directives, for the *bottleneck module* to make it change into a *non-bottleneck module* with the lowest resource overhead. As shown in Algorithm 1, Hi-

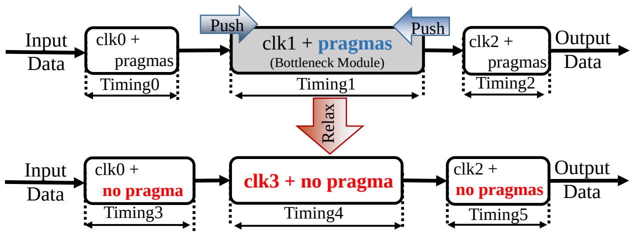

Fig. 5. Example of Pushing-Relaxation Heuristic Algorithm

ClockFlow will keep finding the current bottleneck module in the dataflow to push it, improve its parallelism and reduce its latency, until the constraint of resource is violated, i.e. pushSuccess is false. The achieved minimum initial interval of *bottleneck module* will be recorded during DSE.

There are three potential directions to improve the performance of a module, unrolling a loop with a greater factor, pipelining a loop with a smaller II and partitioning arrays with a greater factor. For most of applications, when a loop is unrolled, the arrays accessed in the loop will be partitioned to allow more parallel accesses and make loop unrolling affect. Therefore, with Light-HLS, Hi-ClockFlow can find the arrays related to the specific loop and partitioning them accordingly when the loop is unrolled. Hi-ClockFlow will select the slowest last-level sub-loop in the slowest top-level loop to optimize. If the target loop can be pipelined with a smaller II, the smaller II will be set for the loop, otherwise, the loop will be unrolled with a larger factor, until it is unrolled completed, i.e. it is not a loop anymore. For example, in Fig. 5, the directives, i.e. pragmas, of *bottleneck module* are pushed to have high parallelism and reduce clock cycles.

**Procedure of Relaxation:** As shown in Algorithm 1, when the pushing procedure cannot further improve the performance under the given resource constraints, the setting of clock domain for the current *bottleneck module* will be updated to achieve possible further performance improvement. A set of clock periods, size of which can be more than ten, is given. Hi-ClockFlow will select the next higher frequency than the current one for the *bottleneck module*. However, some original HLS directives may not be applicable because of the change of clock frequency and the configuration may be trapped in a local optima. To resolve this issue, Hi-ClockFlow will remove all the HLS directives for all the modules in dataflow after clock frequency update and restart the procedure of pushing. For example, in Fig. 5, during relaxation, the clock of *bottleneck module* is changed from $clk1$ to $clk3$ and the directives in the design are removed.

Since the throughput of dataflow is determined by the *bottleneck module*, during DSE, Hi-ClockFlow will continuously try to resolve current bottleneck with the lowest resource overhead and in this way Hi-ClockFlow can optimize the throughput. Please note that the options for pushing and relaxation should be under the constraints listed in Section V-A, otherwise they will be disabled. For example, pushing will be stopped when the constraint of resource is violated, and relaxation will bypass an option of clock period to another one when the number of the clock domains exceeds the give number.

## VI. Evaluation of Hi-ClockFlow

Based on benchmarks from Polybench [13] and hand-written simple CNN source code, we evaluate the DSE method of dataflow. The reason why we implement the simple CNN as benchmark is that actually many dataflow applications can be regarded as a series of filters and CNN is a typical example. The library of characterized IR instructions is collected by Light-HLS for Xilinx Zynq-7020 and the designs for the benchmarks are implemented on Zedboard. For each benchmark, we set fixed resource constraints. To show the benefit brought by multiple clock domains, we also conduct DSE for single clock domain for different clock frequencies, by reducing the clock set into one specific clock and only exploring the configuration of HLS directives. The throughput of dataflow can be represented by the initial interval (II) of the design, which we collect in the experiments. The results for the experiments are shown in Fig. 6 and the setting and acceleration ratio of multiple clock domains are listed in Table. I. The acceleration ratio of multiple clock domains is calculated by dividing the shortest II achieved by single clock design by the one achieved with multiple clocks.

According to experimental results for benchmark deriche, 2mm and CNN, we can notice that higher clock frequency does not means better performance, which verifies the statement in Section III and emphasizes the importance of the clock setting in HLS designs.

As found in Fig. 6, compared with those single-clock design, based on pushing-relaxation DSE, multiple-clock designs can achieve lower initial interval of dataflow, i.e. higher the throughput. However, the extent of the benefit of multiple clock domains will be effected by the application pattern, especially whether the modules in dataflow are balanced. Such balance between modules means they have similar combinations of computations and memory accesses. For example, benchmark 2mm is not balanced, because its modules have different numbers of multiplications. A more significant example of imbalanced modules is CNN, since each layer in CNN may have various filter and strides. For these imbalanced applications, multiple clock domains can help to balance the resource and performance among modules and increase the throughput. In contrast, if the application is originally balanced, it cannot benefit from multiple clock domains. For example, in benchmark jacobi-2d, the modules in dataflow are almost the same and consequently, Hi-ClockFlow does not make difference because it will end with single clock domain for the design as shown in Table I. A special situation is that for some applications, the dataflow based on multiple clock domains may have the throughput similar to those of single clock designs. For this situation, for some applications, like fdtd-2d, some of their modules can have lower clock frequencies, which could reduce the power consumption.

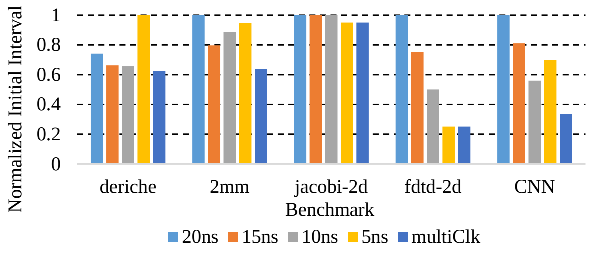

Fig. 6. Comparison between Single-Clock Dataflow and Multi-Clock Dataflow: Initial intervals for the application with different clock settings are normalized to the maximum value.

TABLE I
Speed-Up Ratio and Clock Domains for Benchmarks

| benchmark | deriche | 2mm | jacobi-2d | fdtd-2d | CNN |
|---|---|---|---|---|---|
| Clocks / ns | 15/17.5/20 | 5/15 | 5 | 5/20 | 7/12.5/20 |
| SpeedUp | 1.07x | 1.33x | 1.00x | 1.00x | 1.81x |

## VII. Conclusion

In this paper, we present Hi-ClockFlow, an automatic HLS framework, which can analyze the source code based on Light-HLS, our light weight HLS evaluation framework, explore the design space and optimize clock frequencies and HLS directives in dataflow. Based on experiments, we analyze the effect of clock in HLS design and demonstrate that multi-clock design can improve the performance for dataflow with imbalanced modules. Hi-ClockFlow, Light-HLS and mentioned test cases are open and available to the community on https://eeweiz.home.ece.ust.hk/.

## References

[1] X. Jia and R. Vemuri, "The gapla: a globally asynchronous locally synchronous fpga architecture," in *FCCM '05*, April 2005, pp. 291–292.

[2] A. Royal and P. Y. Cheung, "Globally asynchronous locally synchronous fpga architectures," in *FPL '03*. Springer, 2003, pp. 355–364.

[3] O. Ragheb and J. H. Anderson, "High-level synthesis of fpga circuits with multiple clock domains," in *FCCM '18*, April 2018, pp. 109–116.

[4] M. J. Mamaghani, J. Garside, and D. Edwards, "De-elastisation: From asynchronous dataflows to synchronous circuits," in *DATE '15*, March 2015, pp. 273–276.

[5] M. J. Mamaghani, D. Sokolov, and J. Garside, "Asynchronous dataflow de-elastisation for efficient heterogeneous synthesis," in *ACSD '16*, June 2016, pp. 104–113.

[6] M. J. Mamaghani, M. Krstic, and J. Garside, "Automatic clock: A promising approach toward galsification," in *ASYNC '16*, May 2016, pp. 83–84.

[7] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance modeling and directives optimization for high level synthesis on fpga," *IEEE TCAD*, pp. 1–1, 2019.

[8] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," in *DAC '18*, June 2018, pp. 1–6.

[9] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 97–108.

[10] P. Li, L.-N. Pouchet, and J. Cong, "Throughput optimization for high-level synthesis using resource constraints," in *IMPACT14*, 2014.

[11] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong, "Resource-aware throughput optimization for high-level synthesis," in *FPGA '15*. New York, NY, USA: ACM, 2015, pp. 200–209.

[12] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A framework for generating high throughput cnn implementations on fpgas," in *FPGA '18*. New York, NY, USA: ACM, 2018, pp. 117–126.

[13] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.