DASS: Combining Dynamic & Static Scheduling in High-level Synthesis

Jianyi Cheng, *Student Member, IEEE,* Lana Josipović, *Student Member, IEEE,* George A. Constantinides, *Senior Member, IEEE,* Paolo Ienne, *Senior Member, IEEE,* and John Wickerson, *Senior Member, IEEE*

***Abstract*—A central task in high-level synthesis is *scheduling*: the allocation of operations to clock cycles. The classic approach to scheduling is *static*, in which each operation is mapped to a clock cycle at compile-time, but recent years have seen the emergence of *dynamic* scheduling, in which an operation's clock cycle is only determined at run-time. Both approaches have their merits: static scheduling can lead to simpler circuitry and more resource sharing, while dynamic scheduling can lead to faster hardware when the computation has non-trivial control flow. In this work, we seek a scheduling approach that combines the best of both worlds. Our idea is to identify the parts of the input program where dynamic scheduling does not bring any performance advantage and to use static scheduling on those parts. These statically-scheduled parts are then treated as black boxes when creating a dataflow circuit for the remainder of the program which can benefit from the flexibility of dynamic scheduling. An empirical evaluation on a range of applications suggests that by using this approach, we can obtain 74% of the area savings that would be made by switching from dynamic to static scheduling, and 135% of the performance benefits that would be made by switching from static to dynamic scheduling.**

***Index Terms*—High-Level Synthesis, Static Analysis, Dynamic Scheduling.**

I. Introduction

HIGH-level synthesis (HLS) is the process of automatically translating a program in a high-level language, such as C, into a hardware description. It promises to bring the benefits of custom hardware to software engineers. Such design flows significantly reduce the design effort compared to manual register transfer level (RTL) implementations. Various HLS tools have been developed in both academia [1], [2] and industry [3], [4].

The Challenge of Scheduling: One of the most important tasks for an HLS tool is scheduling: allocating operations to clock cycles. Scheduling decisions can be made either during the synthesis process (static scheduling) or at run-time (dynamic scheduling).

The advantage of static scheduling (SS) is that since the hardware is not yet online, the scheduler has an abundance of time available to make good decisions. It can seek operations that can be performed simultaneously, thereby reducing the latency of the computation. It can also adjust the start times of operations so that resources can be shared between them, thereby reducing the area of the final hardware. However, a static scheduler must make conservative decisions about which control-flow paths will be taken, or how long variable-latency operations will take, because this information is not available until run-time.

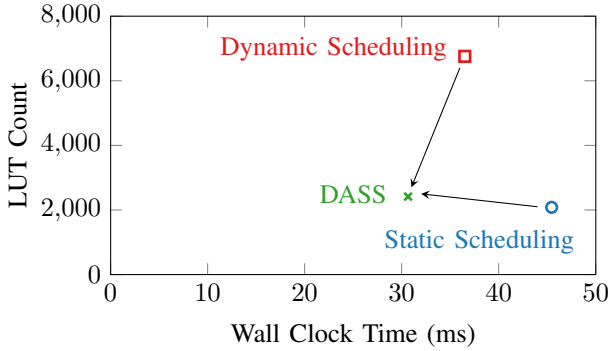


Fig. 1: Area of three scheduling approaches over performance.

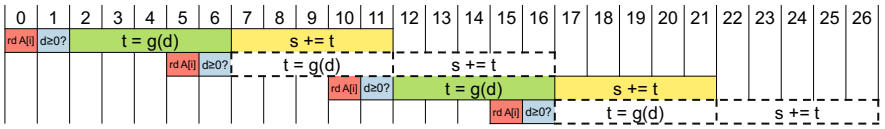
Dynamic scheduling (DS), on the other hand, can take advantage of this run-time information. Dynamically scheduled hardware consists of various components that communicate with each other using handshaking signals. This means that operations are carried out as soon as the inputs are valid. In the presence of variable-latency operations, a dynamically scheduled circuit can achieve better performance than a statically scheduled one in terms of clock cycles. However, these handshaking signals may also cause a longer critical path, resulting in a lower operating frequency. In addition, because scheduling decisions are not made until run-time, it is difficult to enable resource sharing. Because of this, and also because of the overhead of the handshaking circuitry, a dynamically scheduled circuit usually consumes more area than a statically scheduled one.

Our Solution: Dynamic & Static Scheduling: In this article, we propose dynamic and static scheduling (DASS): a marriage of SS and DS that aims for minimal area *and* maximal performance, as sketched in Fig. 1. The basic idea is to identify the parts of an input program that may benefit from SS—typically parts that have simple control flow and fixed latency—and to use SS on those parts. In the current incarnation of this work, it is the user's responsibility to annotate these parts of the program using pragmas, but in the future we envisage these parts being automatically detected. The statically scheduled parts are then treated as black boxes when applying DS on

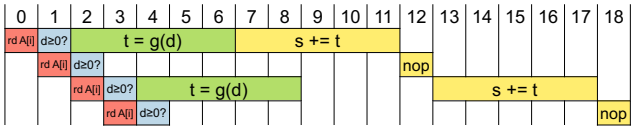
J. Cheng, G. Constantinides and J. Wickerson are with the Department of Electrical and Electronic Engineering, Imperial College London, UK.
E-mail: {jianyi.cheng17, g.constantinides, j.wickerson}@imperial.ac.uk
L. Josipović and P. Ienne are with Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland.
E-mail: {lana.josipovic, paolo.ienne}@epfl.ch
Manuscript received April 19, 2005; revised August 26, 2015.

```
1  double A[N]; // initialised at run
       -time to {1.0, -1.0, 1.0,
       -1.0, ...}
2  double g(double d) {
3    return (((((d+0.2)*d+0.3)*d+0.6)
       *d+0.2)*d+0.7)*d+0.2;
4  }
5  double filterSum() {
6    double s = 0.0;
7    for (int i = 0; i < N; i++) {
8      double d = A[i];
9      if (d >= 0) {
10       double t = g(d);
11       s += t;
12     }
13   }
14   return s;
15 }
```

(a) Motivation code



(b) The schedule of the SS circuit (the loop II = 5).



(c) The DS circuit and DASS circuit have the same schedule (II_g in DASS = 1).

Fig. 2: Motivating example of dynamic and static schedules. DS has better performance when comparing with SS. Our work propose a DASS solution having comparable performance to DS. The latency of function `g` is 59 cycles but is represented as 5 cycles in the figure to save space.

the remainder of the program.

Several challenges must be overcome to make this marriage work. These include: (1) How should statically scheduled parts be correctly and efficiently integrated into their dynamically scheduled surroundings? (2) How should the memory be correctly and efficiently shared between the statically scheduled circuit and the dynamically scheduled circuit?

In this article, we show how these challenges can be overcome. Our evaluation on several realistic benchmarks demonstrates that it is possible to obtain 74% of the area savings that would be made by switching from DS to SS and 135% of the performance benefits that would be made by switching from SS to DS. In other words, DASS can obtain most of the area benefits associated with SS, and can actually outperform both DS and SS.

Article Outline: In Section II, we give a working example to motivate the combined scheduling approach in which some scheduling decisions are taken dynamically at run-time and the others are determined offline using traditional HLS techniques. Section III provides a primer on existing SS and DS techniques. In Section IV, we describe how our proposal overcomes challenges related to component integration and shared memory. Section V details a prototype implementation of DASS that uses Xilinx Vivado HLS [3] for SS and Dynamatic [5] for DS. In Section VI, we evaluate the effectiveness of DASS on a set of benchmarks and compare the results with the corresponding SS-only circuits and DS-only circuits.

Relationship to Prior Publications: This article expands on a conference paper by Cheng *et al.* [6] in two main directions. First, we include an additional three realistic benchmarks: two benchmarks to show more code properties amenable for DASS; and one benchmark with a detailed case study to identify the limits of DASS and show how it affects performance and area. Second, we address a limitation of the work described in our conference paper: that it did not allow memory to be shared between the SS circuit and the DS surroundings. This limitation places severe restrictions on which parts of a program can be statically scheduled. For instance, if a kernel of a tiled loop is to be statically scheduled, then all the operations that access the same array would have to be statically scheduled, too – even those that do additional processing on boundary tiles that might be better suited to dynamic scheduling. Such a code pattern can result in a sub-optimal pipeline solution. In this article, we extend our tool by supporting shared memory between the SS and DS hardware thus allowing a more fine-grained division between SS and DS components. Finally, we add a realistic benchmark to demonstrate the shared memory architecture between DS and SS hardware.

Auxiliary Material: All of the source code of benchmarks and the raw data from our experiments are publicly available [7], [8]. Our prototype tool, which relies on the Vivado HLS and Dynamatic HLS tools, is also open-sourced [9].

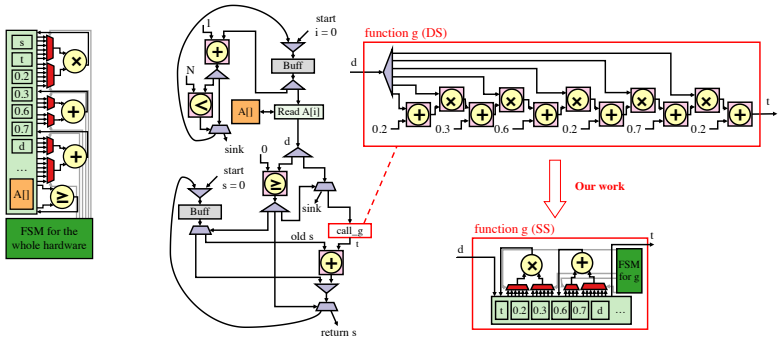
II. Overview

We now demonstrate our approach via a worked example.

Fig. 2(a) shows a simple loop that operates on an array `A` of doubles. It calculates the value of `g(d)` for each non-negative `d` in the array, and returns the sum of these values. The `g` function represents the kind of high order polynomial that arises when approximating complex non-linear functions such as `tanh` or `log`. If the values in `A` are provided at run-time as shown at the top of Fig. 2(a), then function `g` is only called on odd-numbered iterations.

To synthesise this program into hardware, we consider three scheduling techniques: SS, DS, and our approach, DASS.

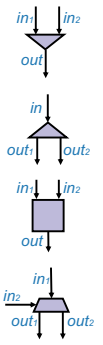
SS – Small Area but Low Performance: The hardware resulting from SS is shown in Fig. 3(a). It consists of three main parts. On the left are the registers and memory blocks that store the data. On the right are several operators that perform the computation described in the code. At the bottom is an FSM that monitors and controls these data operations in accordance with the schedule determined by the static scheduler at compile time. The SS circuit achieves good area efficiency through the use of resource sharing, *i.e.* using



(a) SS circuit (b) DS circuit and its transformation into a DASS circuit

Fig. 3: DS has larger area when comparing with SS, and our work makes the area smaller without losing performance.

TABLE I: Dataflow components in DS circuits.



Merge: receives an input data from one of its multiple predecessors and forwards it to its single successor.

Fork: receives a piece of data at its only input and replicates it to send a copy to each of its multiple successors.

Join: triggers its output only once both of its inputs are available.

Branch: is a control-flow component that passes a piece of data to one of its two successors, depending on the input condition.

multiplexers to share a single operator among different sets of inputs.

The timing diagram of the SS circuit is shown in Fig. 2(b). It is a pipelined schedule with an initiation interval (II) of 5. The II cannot be reduced further because of the loop-carried dependency on `s` in line 11. Since the `if` decision is only made at run-time, the scheduler cannot determine whether function `g` and the addition are performed in a particular iteration. It therefore conservatively reserves their time slots in each iteration, keeping II constant at 5. This results in empty slots in the second and fourth iteration (shown with dashed outlines in the figure), which cause the operations in the next iteration to be unnecessarily delayed.

DS – Large Area but High Performance: The DS hardware is a dataflow circuit with a distributed control system containing several small components representing instruction-level operations [5], as shown on the left of Fig. 3(b). Each component is connected to its predecessors and successors using a handshaking interface. This handshaking, together with the inability to perform resource sharing on operators, causes the area of the DS hardware to be larger than the corresponding SS hardware.

The timing diagram of the DS circuit is shown in Fig. 2(c). It has the property that each operator executes as soon as its inputs are valid, so the throughput can be higher than that for SS hardware. For instance, it can be seen that the read of `A[i]` in the second iteration starts immediately after the read in the first iteration completes. Most stalls in a DS circuit are due to data dependencies. For instance, the execution of function `g` and the addition in the second iteration are skipped as `d = -0.1 < 0`, leading to `s = old_s`. The operation is not carried out immediately after the condition check but stalled until `s += t` in the first iteration completes, since it requires the output from the previous operation as input. Then it is immediately followed by `s += t` in the third iteration.

DASS – Both Small Area and High Performance: The DASS hardware combines the previous two scheduling techniques. It is based on the observation that although the overall circuit's performance benefits from DS, the function `g` does not because it has a fixed latency. Therefore, we replace the dataflow implementation of `g` with a functionally equivalent SS implementation. The SS implementation uses resource sharing to reduce six adders and five multipliers down to just one of each. The rest of the circuit outside `g` is the same as the DS circuit. Because `g` represents a substantial fraction of the overall hardware, this transformation leads to the area of the DASS hardware being close to that of the pure SS hardware, as shown in Fig. 1.

The timing diagram of the DASS circuit is the same as that of the DS circuit, as shown in Fig. 2(c).[1] In the DS circuit, `g`'s schedule is determined at run-time, while in the DASS circuit it is determined by the static scheduler; in both cases, the timing diagram is the same. The data-dependent `if` condition in the loop remains part of the DS circuit to maximise throughput. Hence the DASS hardware and the DS hardware have the same throughput in terms of clock cycles. However, since the SS implementation of `g` optimises the critical path of the system, the DASS hardware can actually run at a higher clock frequency. Therefore, in this example, DASS hardware achieves not merely the 'best of both worlds', but actually achieves *better* performance than DS hardware (in terms of wall clock time), and *comparable* area to SS hardware, as shown in Fig. 1.

In the rest of the paper, we give the details of how to configure the constraints of the static parts for maximising resource sharing and preserving performance, and the methodology for integrating the static parts into the dataflow circuit.

III. Background

In this section, we review the basics of HLS scheduling. We discuss related work in static and dynamic scheduling techniques and contrast them with the approach we present in this work.

A. Scheduling in HLS

In most HLS tools like LegUp [1] and Vivado HLS [3], the tool flow is divided into two steps: frontend and backend. In the frontend, the input source code is compiled into an intermediate representation (IR) for program analysis and

[1] Actually, the latency of function `g` varies slightly between DS and SS for technical reasons, as explained in Section VI.

transformation. In the backend, the IR is transformed into an RTL description of a circuit, during which static scheduling is carried out, as well as allocation and binding.

The scheduling process in most HLS tools starts by converting the IR into a control/data flow graph (CDFG) [10]. A CDFG is a two-level directed graph consisting of a number of vertices connected by edges. At the top level, the graph is represented as a control-flow graph (CFG), where each vertex corresponds to a basic block (BB) in the transformed IR, while edges represent the control flow. At a lower level, a vertex corresponding to a BB is itself a data-flow graph (DFG) that contains a number of sub-vertices and sub-edges. Each sub-vertex represents an operation in the BB and each sub-edge indicates a data dependency.

B. Static Scheduling

In HLS tools, scheduling is the task of translating the CDFG described in the previous section, with no notion of a clock, into a timed schedule [11]. The static scheduler determines the start and end clock cycles of each operation in the CDFG, under which the control flow, data dependencies, and constraints on latency and hardware resources, are all satisfied. One of the most common static scheduling techniques, used by Vivado HLS [3] and LegUp [1], expresses a CDFG schedule as a solution to a system of difference constraints (SDC) [12]. Specifically, it formulates scheduling as a linear programming (LP) problem, where the data dependencies and resource constraints are represented as inequalities. By changing these constraints, various scheduling objectives can be customised for the user's timing requirements.

Besides achieving high performance, static scheduling also takes resource allocation into account, such as modulo scheduling for loop pipelining [13]. It aims to satisfy the given time constraints with minimum possible hardware resources or achieve the best possible performance under the given hardware resource requirements. If the hardware resource constraints are not specified, the binder automatically shares some hardware resources among the operations that are not executed in parallel. This maintains the performance but results in smaller area. In addition, typical HLS tools like Vivado HLS allow users to specify resource constraints via pragmas. In this case, the binder statically fits all operations into a given number of operators or functions based on the given schedule. This may slow down execution if hardware resource is limited.

In summary, static scheduling results in efficient hardware utilisation by relying on the knowledge of the start times of the operations to share resources while preserving high performance. However, when the source code has variable-latency operations or statically indeterminable data and control dependencies, static scheduling conservatively schedules the start times of certain operations to account for the worst-case timing scenario, hence limiting the overall throughput and achievable performance.

C. Dynamic Scheduling

Dynamic scheduling is a process that schedules operations at run-time. It overcomes the conservatism of traditional static scheduling to achieve higher throughput in irregular and control-dominated applications, as we saw in Fig. 3. Similarly, dynamic scheduling can handle applications with memory accesses which cannot be determined at compile time. For instance, given a statement like `x[h1[i]] = g(x[h2[i]])`, the next read of `x` can begin as soon as it has been determined that there is no read-after-write dependency with any pending store from any of the previous loop iterations, *i.e.* `h2` is not equal to any prior/pending store address `h1`. A dynamically scheduled circuit will allow the next operation to begin as soon as this inequality has been determined; otherwise, it will appropriately stall the conflicting memory access.

Initial work on dynamically scheduled hardware synthesis from a high-level language proposed a framework for automatically mapping a program in occam into a synchronous hardware netlist [14]. This work was later extended to a commercial language named Handel-C [15]. However, it still required the designer to manually design and implement hardware optimisation such as pipelining and parallelism. Venkataramani *et al.* [16] proposed a framework that automatically transforms a C program into an asynchronous circuit. They implement each node in a DFG of the design in an intermediate representation called Pegasus [17] into a pipeline stage. Each node represents a hardware component in the netlist, containing its own controlling trigger. This dataflow design methodology was then brought into synchronous design. Recent work [5] proposes a tool flow named Dynamatic that generates synchronous dataflow circuits from C code. It can take arbitrary input code, automatically exploits the parallelism of the hardware and uses handshaking signals for dynamic scheduling to achieve high throughput. In this work, we use Dynamatic to generate dynamically scheduled HLS hardware.

As formalised by Carloni *et al.* [18], dynamic scheduling is typically implemented by dataflow circuits which consist of components communicating using handshake signals. Apart from the common datapath operators, a dynamically scheduled dataflow circuit contains a number of dataflow components shown in Table I, used to control the flow of data.

One difficulty for dynamic scheduling is scheduling the memory accesses. In static scheduling, all the memory accesses are scheduled at compile-time, such that there is no memory conflict during the execution. In dynamic scheduling, the untimed memory accesses may affect correctness and performance if the memory arbitration or the memory conflicting accesses are not correctly solved. Hence, dynamically scheduled circuits may use load-store queues (LSQs) [19] to resolve data dependencies and appropriately schedule the arbitrary memory accesses at run-time. However, LSQs cost significant area and add delays to the circuit. To reduce such overhead, the architecture of the LSQ in Dynamatic is optimised by statically identifying the memory aliasing among all the memory accesses [20]. In our work, the memory architecture may contain partially-scheduled memory accesses in the static part and unscheduled memory accesses in the dynamic part. We will detail our approach to handle this situation in Section IV-C.

D. Combining Dynamic & Static Approaches

Several works have explored the integration of certain aspects of dynamic scheduling into static HLS. Alle *et al.* [21] and Liu *et al.* [22] propose source-to-source transformations to enable multiple schedules selected at run-time after all the required values are known. Tan *et al.* [23] propose an approach named ElasticFlow to optimise pipelining of irregular loop nests that contain dynamically-bound inner loops. Dai *et al.* [24], [25] propose pipeline flushing for high throughput of the pipeline and dynamic hazard detection circuitry for speculation in specific applications. These works are still based on static scheduling and work only under stringent constraints, which limits the performance improvements for general cases, such as complex memory accesses. In contrast, our approach adds existing hardware optimisation techniques into dynamically scheduled circuits and supports arbitrary code input under the same restriction with respect to the synthesis tools that DASS replies on independently.[2] Also, these approaches cannot solve the problem of input-dependent behaviour in hardware, while DASS solves it by dynamically scheduling these parts of code to achieve higher throughput.

Carloni [26] describes the theory of how to encapsulate static modules into a latency-insensitive system, and we use a similar integration philosophy. We utilise this approach within our tool, which automates the generation of circuits from high-level code, resulting in the mix of two HLS paradigms in a single synthesis tool.

IV. Methodology

In this section, we show how to partition and synthesise some functions into SS hardware and the rest of the program into a DS circuit. We first discuss which programs are amenable to our approach. We then detail the integration of the SS hardware into the DS circuit using a dedicated wrapper, which ensures that the data is correctly propagated between these two architectures. Finally, we show how to enable memory sharing among SS and DS circuits correctly.

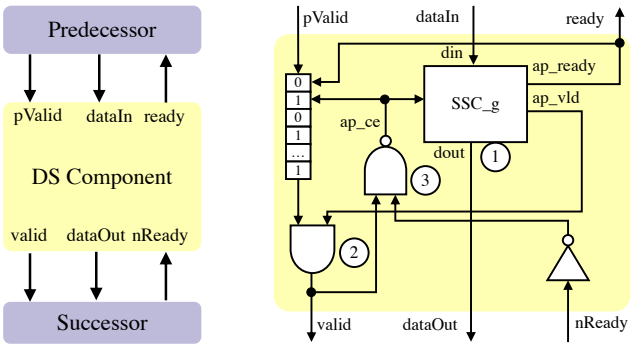
A. Applicability of Our Approach

Our approach is generally applicable, in the sense that it can be used wherever SS or DS can be used. The following conditions indicate scenarios where our approach is likely to yield the most substantial benefits over DS and SS:

1) there is an opportunity to improve throughput using information that only appears at run-time,
2) at least one region of the code has a constant (or low variability) latency, and
3) this code region has an opportunity for resource sharing.

The first condition indicates that the design may be amenable to DS, as explained in Section III-C. The second and third reflect the fact that SS determines a fixed schedule and can take advantage of resource sharing. We emphasise that not *all* of the conditions above need to hold for an input program to benefit from our approach; it is simply that each condition listed above is desirable.

[2]The synthesised hardware from arbitrary code may not be efficient, but still preserves correctness.



(a) Handshaking interface in a DS circuit

(b) SS function g as a DS component

Fig. 4: The statically scheduled (SS) circuit of function g is wrapped with additional control circuitry for interfacing to the DS circuit.

B. Integrating SS Hardware into DS Hardware

In this section, we show how to implement a wrapper of SS hardware around the DS surroundings. A wrapper design of a synchronous circuit around latency-insensitive designs is explained by Carloni [26]. In the design, the wrapper always fires as long as inputs are valid and the successor is ready. In other words, they assume no stall is caused by the internal architecture of the synchronous circuit, *i.e.* the II of the synchronous circuit is always 1. However, in most hardware designs, we want to have efficient hardware architecture as well as high performance. II greater than 1 allows designers to perform more hardware optimisation, like resource sharing, while still preserving high performance. In our work, we support and prefer that II greater than 1, so resource sharing is possible.

A DS circuit is constructed as a dataflow circuit, containing a number of small components, while an SS circuit has a centralised FSM for control. We regard each SS circuit as a component in the dataflow circuit, as indicated in Fig. 3(b). In this section, we explain how to make an SS circuit behave like a DS component so that it can be integrated into the overall DS hardware. Let us look at function g in Fig. 2(a), for example, which is a single-input and single-output function. The multiple-input and multiple-output cases are discussed shortly.

In the DS circuit part, each component communicates with its predecessors and successors using a set of handshaking signals as shown in Fig. 4(a). Each DS component uses the bundled data protocol [27] for communication, where each data connection has request and acknowledgement signals. For instance, the following is the control interface of a component from Dynamatic [5]:

- *pValid:* an input signal indicating that the data from the predecessor is valid,
- *valid:* an output signal informing the successor that the data from the current component is valid,
- *nReady:* an input signal indicating that the successor is ready to take a new input, and

- *ready:* an output signal informing the predecessor that the current component is ready to take a new input.

On the other hand, traditional SS hardware has a different interface to monitor and control the states of the centralised FSM. An example of an HLS tool that generates SS hardware is Vivado HLS [3]. For a typical control interface of an SS function synthesised into SS hardware, its control interface is as follows:

- *ap_ce:* The clock enable signal controls all the sequential operations driven by the clock.
- *ap_ready:* The ready signal from the SS hardware indicates that it is ready for new inputs.
- *ap_vld:* The valid signal indicates the output from SS hardware at the current clock cycle is valid.

The interface of an SS circuit is not compatible with the above handshaking signals in a DS circuit. To overcome this issue, we add a wrapper around each SS circuit, ensuring that the data propagates correctly between the SS circuit and the DS circuit. This wrapper is generated in two steps: 1) In an SS circuit, any output is only valid for one clock cycle. We design a *valid* signal to correctly send the data to the successor and preserve the output when backpressure from the successor occurs; 2) Since there may be a pipeline stall caused by this component, we design a *ready* signal to send the backpressure to the predecessor, ensuring any valid input is not lost.

We now discuss those two steps in more detail.

Constructing the valid signal: In an SS-only circuit, where the entire schedule is determined at compile-time, the arrival time of each input can be predicted. However, this is not the case in DS as the behaviour of the rest of the DS circuit is unknown. Two choices are available: 1) stalling the SS function until valid input data is available, or 2) letting the SS function continue to process data actively in its pipeline, marking and ignoring any invalid outputs. Since the SS function does not have the knowledge of the rest of the DS circuit, the first approach may cause unnecessary stalls. Hence, for performance reasons, we take the second choice with power overhead from keeping SS functions active.

An invalid input read and processed by the SS hardware is named a *bubble*. We use a shift register to tag the validity of the data and propagate only the valid data to the successor, as shown in Fig. 4(b). The shifting operation of the shift register is controlled by the state of the SS hardware to synchronise the data operations in the SS circuit. It shifts to the right by one bit every time the SS hardware takes a new input, as indicated by the *ap_ready* signal. The new bit represents whether the newly taken input data is valid or not. A zero represents a *bubble* and a one represents a valid input. The length of the shift register is determined by the latency and the II of function g: $\lceil latency/II \rceil$, where these time constraints are obtained from the scheduling report by the static scheduler. This ensures that when the output is available from the SS hardware, as indicated by the *ap_vld* signal, its validity is indicated by the oldest bit of the shift register. By checking the oldest bit value, only the valid data is propagated to the successor with the *valid* signal high. In summary, we use the shift register to monitor and control the state of the SS hardware, such that the data can be synchronised between the SS and DS hardware, filtering out the bubbles to ensure the correctness of the function. Similarly, only the memory operations with valid data are carried out.

Constructing the ready signal: The *valid* signal for the successor and the shift register allows data to propagate from the predecessor, through the SS function, to the successor. However, the component is not able to deal with any backpressure from the function or its successor. Backpressure happens when a component is unable to read an input even though it is valid, resulting in its predecessor stalling. In a DS circuit, this issue is solved using handshake signals—the hardware stalls when its output is valid but the successor is not ready, as indicated by its *nReady* signal or the *ready* signal from the successor. We design a control circuit to handle the backpressure between a DS circuit and an SS circuit. Backpressure can arise between a DS circuit and an SS circuit in two ways, which we now discuss.

Backpressure from SS function to its predecessor. In this case, the *ready* signal indicates whether the SS hardware is ready to take an input so *ap_ready* is directly connected to the *ready* signal of the wrapper. It sends feedback to the predecessor such that the predecessor can be stalled, holding the valid input to the SS hardware until the SS hardware is ready.

Backpressure to SS function from its successor. Since the SS hardware only holds the output for one cycle when running, we stall the process in the SS hardware to preserve the output data. This is achieved by disabling the clock signal, $ap\_ce = 0$, so the SS hardware stops all the sequential processes, preserving the output. The condition for such a stall to occur is that the next output from the SS hardware is valid ($valid = 1$) but the successor is not ready ($nReady = 0$). The SS hardware continues running after the *nReady* signal is set to high, indicating that the successor is ready to accept the output data of the SS hardware. This additional circuitry ensures that the data exchanged between an SS circuit and a DS circuit is not lost when any stall occurs.

Handling multiple inputs and multiple outputs: The example above shows the wrapper for a function with a single input and an output. However, it is also common to have a function with zero or more than one inputs or outputs. If there is no input and output, the external DS circuit would have no corresponding data port for the component, hence no corresponding handshaking signal is needed. Here we focus on the cases of multiple inputs and outputs.

For multiple inputs, we construct a set of handshaking signals as shown in Fig. 4(b) for each input and synchronise data with the help of *join* components in Table I, such that the SS hardware always takes all the input simultaneously. A *join* component is used to preserve the valid inputs to the SS circuit until all the inputs are valid. This is similar to a DS component with multiple inputs.

For multiple outputs of the SS function, each component has its own handshaking signals. The output handshaking signals are implemented in two parts, the *valid* and *nReady* signals. First, each output has its own *valid* signal. For each output, the SS circuit has a *ap_vld* signal indicating whether the corresponding output is valid. Each *ap_vld* signal is *AND*ed with

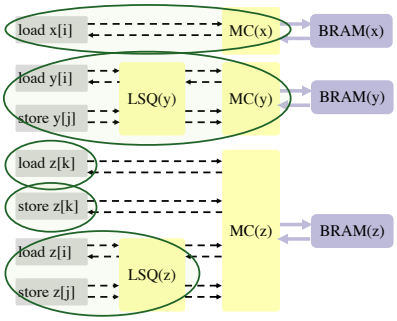


Fig. 5: An example of the netlist of all memory accesses for a purely DS circuit. The analysis in DS reports which memory node may cause memory conflicts. Each green circle means the minimum unit to be integrated in a SS hardware.

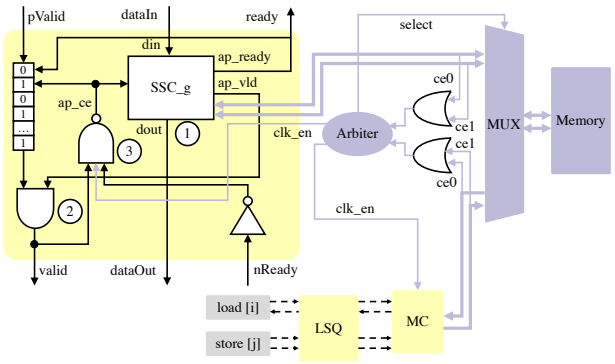


Fig. 6: Shared memory architecture in the DASS hardware. An arbiter is used to decide which hardware to access the memory in each clock cycle.

the oldest bit of the shift register ($sr(0)$) as the corresponding *valid* signal: $valid_i = ap\_vld_i \wedge sr(0), i = 0, 1, 2, ...$ Second, any unset *nReady* signal from the output when the data is valid can disable the clock in the SS hardware as back pressure: $ce = \neg \wedge ((\neg nReady_i) \wedge valid_i)$.

C. Shared Memory Between SS and DS Circuits

The SS hardware has its memory accesses pre-scheduled at compile time and can interact with the memory at run time in a predictable sequence, while as explained in Section III-C, the DS hardware requires an load-store queue (LSQ) [19] that schedules the memory accesses at run-time before accessing the data. In DASS hardware, a combination of the two memory architectures needs to be handled. An LSQ is beneficial for programs that have irregular memory accesses. It does not bring performance improvements when implementing regular computation in the form of an SS circuit. The DS tool uses static analysis, such as polyhedral analysis, to identify the memory accesses in the dataflow graph that cannot cause any memory conflict [20]. Then these memory access nodes are directly connected to the BRAM through an arbiter, as the memory controller, instead of being scheduled by an LSQ.

Figure 5 shows an example of the memory architecture of purely DS hardware. The grey nodes on the left are the memory nodes, and the yellow blocks are the memory components in the DS hardware. The blocks on the right are the BRAM blocks on FPGAs. Each block represents an array in the input program. The dotted edges are handshaking interface, while the solid edges are the block memory interface. In DS hardware, each memory node performs a single load/store operation, and all these nodes are connected to the memory components through handshaking signals. The LSQ schedules the memory with dependencies, while the memory controller (MC) is a simple memory arbiter which issues independent memory accesses to memory. Then the memory components serialise the requests from these nodes and perform the corresponding memory operations with the BRAM through the block memory interface.

In the figure, the DS hardware has seven memory access nodes targeting three arrays. Here we assume `load z[k]` cannot have conflicts with other accesses to `z`, the same as `store z[k]`. Since these arrays are separate, there are three memory controllers to manage the accesses to the memory. Firstly, the array `x` is only accessed by a single *load* so the node can be directly connected to the memory controller. Secondly, the *load* and *store* with the indices `i` and `j`, whose values are determined at run time, may depend on each other when accessing the same array `y`. Therefore, an LSQ is required to ensure that those memory accesses are carried out in the correct sequence. Finally, the memory accesses to array `z` have both regular and irregular patterns. The DS analysis proves that the regular memory accesses do not conflict with the irregular memory accesses. Hence, these regular memory accesses can be safely connected to the memory controller, skipping the LSQ. One advantage of such an approach is to minimise the overhead caused by the LSQ and maintain the dynamic mechanism of the memory architecture. The DS compiler analyses the program and constructs efficiency memory architecture above for the DS hardware.

In DASS, we use the results of the above analysis to identify whether the memory accessed by the SS hardware can be shared safely with the DS hardware. We inline all the SS functions at the top-level program and send it to the memory analyser used by DS. The DS compiler outputs a memory architecture graph at the top level, such as Fig. 5. The graph shows the connection of each memory node to the memory controller if the whole hardware is dynamically scheduled. Based on that graph, our tool divides all these memory nodes into several node sets, shown as green circles. All the nodes of a single set must belong to the same hardware region. For a SS function that contains unpredictable memory accesses, the top-level SS function cannot be pipelined. The reason is that the external DS surroundings ignore the memory dependencies inside the SS function. For instance, a SS function in a loop that may require the maximum loop throughput to be 0.2 in Fig. 2(b), while the dynamically scheduled loop feeds data at a throughput of 1 set of data per clock cycle in Fig. 2(c) as it does not see the inter-iteration dependency hidden in the SS function.

For instance, in Fig. 5, all the nodes are divided into 5 sets indicated by the green circles: 1) Any node directly connected to the memory controller is a single set; 2) any node sharing the same LSQ belongs to the same set. The

memory accesses in each green circle may cause conflicts, so they have to be scheduled together either dynamically or statically. The reason is that each SS hardware is treated as a black box by the LSQ, and the behaviour of SS hardware accessing the memory is unknown for a single input. This is different from purely DS circuits, in which each memory node only accesses the memory once every time the single node is triggered. A SS component may access memory multiple times in one computation if it contains contains multiple memory statements or loops. Therefore, our approach only allows complete sets in the SS hardware. Such a method ensures that there is no memory conflict between the SS hardware and the DS surroundings whenever they access the shared memory. If a set of nodes using an LSQ is considered to be statically scheduled, the LSQ is no longer needed. The performance may be affected by the conservatism in SS, however there is significant area reduction as the LSQ is no longer needed.

Constructing Shared Memory Interface in DASS: In SS, the nodes in Fig. 5 are no longer distributed but scheduled at compile time. The memory interface is the block memory interface instead of the handshaking signals. Once the access to a shared memory block occurs, our tool automatically adds the memory interface to the wrapper. Fig. 6 shows an example of a shared memory architecture in the DASS hardware. The yellow block on the left is the wrapper for the SS hardware, as shown in Fig. 4(b). Apart from that, an additional memory wrapper is added shown as the circuits on the right. As mentioned previously, the SS hardware directly accesses the memory through the BRAM interface shown as the thick arrows. The DS hardware uses the memory controllers to transform the handshaking signals in dotted arrows to the same interface as the SS hardware. In order to ensure that there is no conflict among these two systems, our tool adds a memory arbiter in the memory system. In every clock cycle, the arbiter grants access to either the DS hardware or a SS circuit, and stalls the others by controlling the clock enable bits *ce* of these components. If more than one hardware, either SS competing with DS or SS competing with SS, request to access the memory, the memory arbiter chooses one to grant in a round-robin fashion and stalls the rest of the hardware for data synchronisation. The priority of the SS hardware is always higher than the DS hardware since we expect the SS hardware to run at the highest throughput.

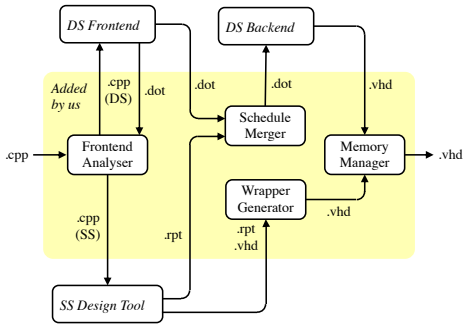
Summary: We identify code that is amenable for DASS, where the design quality of the resulting hardware can be improved. With our wrapper, the SS hardware can work correctly in a DS circuit. Finally, we show how to automatically validate the memory correctness between the SS and DS hardware, and synthesise efficient shared memory architecture in the DASS hardware. Our experiments have shown that the proposed shared memory interface allows us to remove all the LSQs in the benchmarks in Section VI.

V. Implementation

Our approach is generic and can be used with various SS and DS HLS tools. For our work, we choose Vivado HLS [3] and Dynamatic [5] to synthesise the SS and DS hardware

```
1 g(...){
2 #pragma SS II=N
3   ...
4 }
5 f(...){
6   ...
7   call g;
8   ...
9 }
```

(a) Source



(b) DASS tool flow

Fig. 7: With user-specified constraints in pragmas, our tool automatically generates a combined dynamically and statically scheduled circuit.

respectively. Our tool flow is shown in Fig. 7. The user-defined scheduling constraints are configured using *pragmas*. DASS takes the input C++ code and splits the functions into two groups based on the pragmas specified by the user, representing the SS and the DS functions.[3] We synthesise a function without any scheduling constraints to DS hardware by default. Our tool supports the integration of multiple SS functions into a DS function. A front end analysis is carried out to identify whether there is inter-iteration dependence or shared memory between the SS function and the DS function. Then each SS functions is synthesised by Vivado HLS. If a SS function has no inter-iteration dependence, the II of the function is either an II defined by the user or the optimal II determined by Vivado HLS. If a SS function has an inter-iteration dependence, the it is synthesised with a sequential schedule. The resultant SS hardware is then automatically wrapped up to ensure compatibility with the DS circuit interface, as described in Section IV-B. Each input variable or output variable of the function is constructed as a data port with a set of handshaking signals. The memory port for exclusive array accesses from the SS function is directly forwarded to the memory block. If the array is shared by other hardware, then an arbiter is generated to serialise the memory accesses.

In the DS function that contains SS functions, each SS function appears as a single component in the DS hardware netlist. We access the dataflow graph in Dynamatic that contains the timing constraints of all these DS components and update the II and latency of each SS function in terms of the corresponding scheduling report from Vivado HLS. This ensures correct hardware optimisation in the backend of Dynamatic. Finally, the resultant RTL files represent the final DASS hardware of the top function.

VI. Experiments

We evaluate our work on DASS on a set of benchmarks, comparing with the corresponding SS-only and DS-only designs. We assess the impact of DASS on both the circuit area and the wall clock time.

[3]The SS region of code is required to be a function, such that it can be scheduled by Viavdo HLS.

We evaluate our approach on the latency and the area of the whole hardware compared to existing scheduling approaches. Specifically, we select a number of benchmarks, where the DS approach generates hardware with lower latency than the SS approach, and show how the area overhead can be reduced while preserving low latency. To ensure fairness, we present the best SS solution from Vivado HLS and the best DS solution from Dynamatic for each benchmark as a baseline. In addition, we assume that the designer has no knowledge of the input data distribution for the DASS hardware and show that the area and execution time can still be reduced. This means we use the conservative II automatically obtained from Vivado HLS, *i.e.* the smallest possible II determined by only the topology of the circuit like loop carried dependency. The timing results of our work are shown as a range of values that depends on the input data distribution. We obtain the total clock cycles from ModelSim 10.6c and the area results from Post & Synthesis report in Vivado. The FPGA family we used for result measurements is xc7z020clg484 and the version of Vivado software is 2018.3. All our designs are functionally verified in ModelSim on a set of test vectors representing different input data distributions.

A. Benchmarks

The HLS hardware can benefit from the dynamic scheduling for the highest throughputs. We select a number of benchmarks that are amenable for DS, and evaluate our work, DASS, on further optimising the design quality in terms of the performance and area. The first two benchmarks are made artificially to demonstrate simple examples. The third and fourth benchmarks are the sparse form of the corresponding benchmarks from the paper by Josipović *et al.* [5]. The two `getTanh` benchmarks apply the existing approximation algorithms on sparse data arrays. These benchmarks are all made publicly available [7].

We apply our approach to eleven benchmarks and selects the SS parts based on the formulation given in Section. IV-A:

- 1) `sparseMatrixPower` performs dot product of two matrices, which skips the operation when the weight is zero.
- 2) `histogram` sums various weight onto the corresponding features but also in a sparse form.
- 3) `filterSum` sums a number of polynomial results from the array elements that meet the given conditions where the difference between two elements from the arrays is non-negative.
- 4) `filterSumIf` is similar but the SS function returns one of two polynomial expressions based on the value of the difference.
- 5) `getTanh` performs the approximated function *tanh(x)* onto an array of integers using the CORDIC algorithm [28] and a polynomial function.
- 6) `getTanh(double)` is similar but uses an array of doubles.
- 7) `BNNKernel` is a small binarised neural network [29].
- 8) `bubbleSort` is a bubble sort algorithm that repeatedly swaps the elements in the one-dimensional array until the sequence is in ascending order.
- 9) `LFK7` is one of the 7th kernel, equation of state fragment, in the Livermore loops [30], a well-known benchmark set for loop kernels.
- 10) `distSum` evaluates the probability of three events in a specific domain by accumulates the three probability density functions (pdf).
- 11) `getIntersection` measures the intersection of polyhedrons, used for modelling tumors in biophotonic cancer treatments [31].

B. Overall Experimental Results

In most benchmarks, our approach has less area and execution time than the corresponding DS hardware. Fig. 8 shows the overall design quality of our approach compared to the SS and DS solutions, complementing the detailed results in Tab. II. In the figure, we show three arrows for each benchmark: the best case (all inputs take the short path), the worst case (all long), and a middle case (half short, half long). The axes are normalised to the corresponding SS solutions at (1,1). The starting point of an arrow represents the LUT usage and execution time of the DS hardware, while the corresponding result of the DASS hardware is at the end of the arrow. The II of the SS function in the DASS hardware is chosen only considering the worst case of the execution patterns, where all the iterations are long, that is $\eta = 1$, assuming the user does not know the input data distribution. With fixed hardware architecture, we show the results of all seven benchmarks with different input data distributions. Generally, our DASS designs sit at the top left of the corresponding SS hardware. In addition, for the same benchmark, most DASS hardware designs are on the bottom left of the DS hardware. It shows that the DASS hardware can be smaller than the DS hardware and have better performance. The arrows point to the top right indicate that the benchmark is not suitable for DASS and will be explained in the later section.

The results of those arrows in the figure show different patterns over the performance, attributing to the variety of code patterns in the benchmarks. For instance, some arrows for the same benchmarks position at a noticeable distance from each other, like groups of 3, 4, 5, 8, 11. The reason is that the performance of the benchmark depends on the distribution of the input data. In contrast, some arrows for the same benchmarks overlap completely, like groups of 7, 9, 10. These arrows indicate that the input data does not affect the performance of the circuit. Similarly, the benchmarks between these two categories result in the partially overlapping or closely positioned arrows, like groups of 1, 2, 6. Besides, from the area point of view, the arrows for the same benchmark has the same area reduction as the hardware architecture is fixed.

Detailed results of these benchmarks are shown in Tab. II, considering all the cases of possible input distribution (from all long to all short). In general, some values in the "Total Cycles" column is a range because the control decisions taken in the code depend on input values. For the SS case, the number of total cycles is often independent of the input due to pipelining worst-case assumptions made by the SS

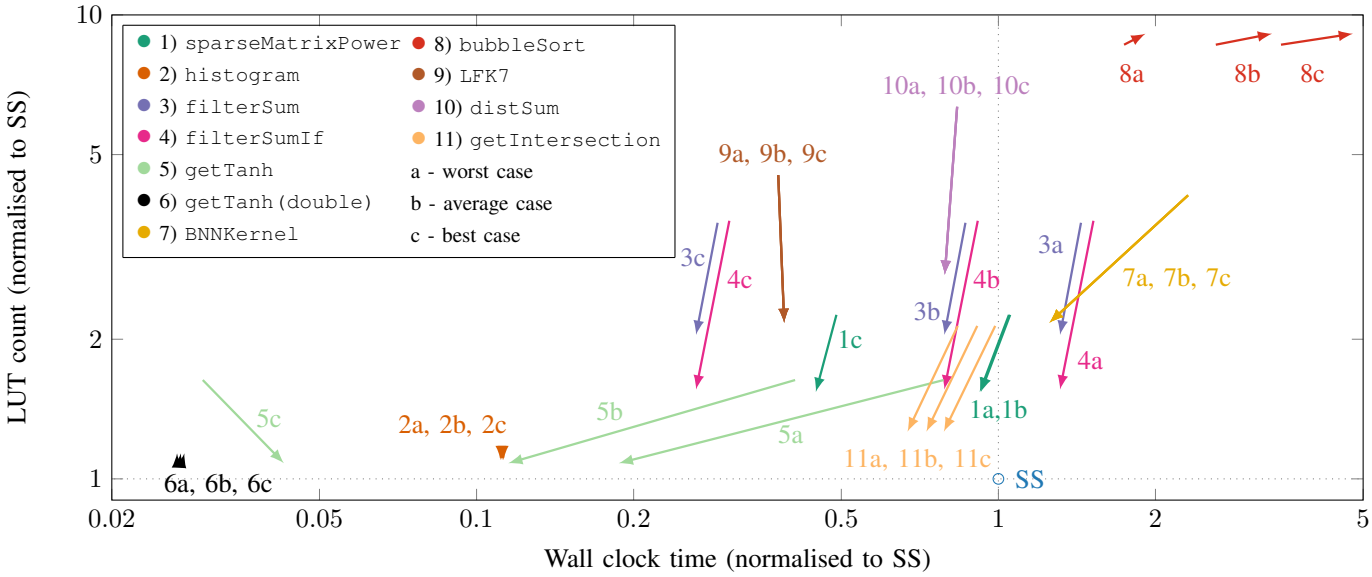


Fig. 8: The overall effects of our approach over the eleven benchmarks from Tab. II. Each benchmark is given three different data distributions: (a) worst case, (b) average case, and (c) best case. Each arrow shows a change in area and performance by switching from DS to DASS, both relative to SS. Most of the arrows lie entirely in the second quadrant, which means both DS and DASS are faster but larger than SS. Arrows that point left mean that DASS is faster than DS; arrows that point down mean that DASS is smaller than DS.

TABLE II: Evaluation of design quality of DASS over eleven benchmarks. Assuming the data distribution is unknown, the II of the static function in DASS is selected as the II in the worst case. The average values are taken except `bubbleSort` as it is not amenable for DASS.

	DASS II	LUTs			DSPs			Registers			Total Cycles			Fmax/MHz			Wall Clock Time/μs		
		SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS
● 1	299	206	465	317	3	6	3	191	489	198	306-30.7k	141-30.0k	141-29.7k	64.9	60.2	67.8	4.7-473	2.3-498	2.1-437
● 2	1	902	1002	990	3	3	3	639	637	809	9.01k	1.01k-1.02k	1.01k-1.01k	111.4	111.4	111.4	80.9	9.0-9.1	9.0-9.1
● 3	5	2209	7874	4514	17	79	23	2592	5552	3960	5.08k	1.02k-5.07k	1.02k-5.08k	111.4	77.3	84.9	45.6	13.2-65.6	12-59.8
● 4	5	3352	12068	5222	31	152	37	3903	9440	5188	5.08k	1.02k-5.07k	1.02k-5.08k	111.4	73.2	85.0	45.6	13.9-69.3	12-59.8
● 5	11	3768	6154	4072	6	12	6	2172	6418	2422	55.0k	2.51k-66.0k	2.51k-11.0k	42.4	64.7	45.2	1298.7	38.8-1.02k	55.5-243
● 6	1	2272	2453	2579	50	50	50	2236	2154	2797	38.0k	1.01k-1.04k	1.01k-1.04k	111.4	111.4	111.4	341.2	9.1-9.3	9.1-9.4
● 7	303 & 402	306	1250	664	3	9	3	142	1606	519	30.9k	30.4k	30.4k	143.3	61.1	112.8	215.7	498	270
● 8	3	91	785	829	0	0	0	109	643	677	2.00M	1.00M-2.00M	1.00M-2.50M	224.2	64.4	58.6	8.92k	15.5k-31.0k	17.1k-42.6k
● 9	4	1466	6620	3179	14	40	10	1831	5943	3475	10043	1460	1464	120.0	46.1	45.0	83.7	31.7	32.5
● 10	10 & 10 & 10	1746	11068	4807	12	128	28	1784	10509	4481	20065	10896	10899	120.0	78.2	82.6	167.2	139.4	132.0
● 11	4 & 13	2241	4788	2832	5	39	19	1642	4808	3180	95-210	89-105	92-110	120.0	83.8	108.1	1.3	1.1-1.3	0.9-1.0
Normalised geometric mean		1	**2.48**	**1.52**	1	**2.65**	**1.08**	1	**3.34**	**1.6**	1	**0.29-0.76**	**0.29-0.61**	1	**0.89**	**0.92**	1	**0.51-1.04**	**0.34-0.73**

scheduler. However, in the case of 1) `sparseMatrixPower` and 11) `getIntersection`, the SS scheduler decides to implement the outer loop of the circuit without pipelining, resulting in sequential execution of iteration and hence also variable execution time. There is also a small difference between the total clock cycles of the DS hardware and the DASS hardware. One of the reasons is that the existence of bubbles causes pipeline stalls at startup and then the throughput is stabilised. The cycle count of the SS hardware in the DASS hardware may also be different from the corresponding DS hardware due to different retiming approach, which also affects the critical path (like function `g` in Section II).

In some of the benchmarks like 3) `filterSum`, the II of the top function is high limited by the topology of the circuit, leading to more area saving. For the benchmarks that contain sparse data operations like 1) `sparseMatrixPower`, although the memory is shared, it can be proved that there is no memory conflict between the SS part and the DS part. Therefore the design quality of the hardware can still be improved by DASS. The DS pipelining capabilities are not always as powerful as those of SS when pipelining more complex loops (*i.e.* the DS hardware sometimes contains more restrictive synchronisation logic which may prevent complete loop pipelining). Hence, in 5) `getTanh`, the DASS design benefits in cycle count by introducing the fully pipelined SS function. The benchmark 7) `BNNKernel` shows that multiple SS functions can be synthesised using our tool. Ideally, all the regular operations in the input code can be synthesised as SS hardware to maximise area efficiency and performance. Besides, 11) `getIntersection` has both an unbounded loop and the data dependent `if` conditions that cannot be fully pipelined by SS. However, the program contains several operations that can be shared as the performance bottleneck is at the memory bandwidth. We identify the SS functions

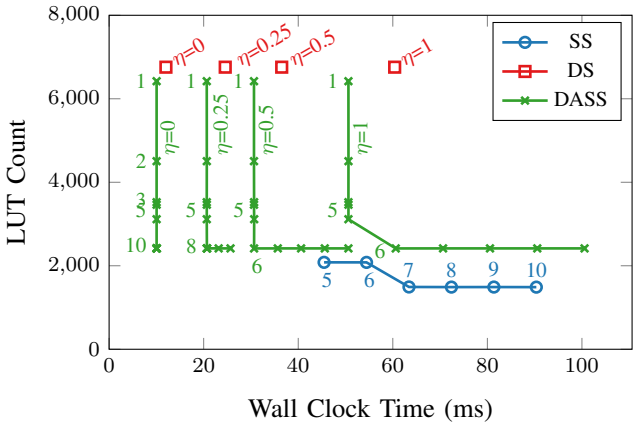


Fig. 9: LUT usage of different scheduling approaches over the performance for the example from Fig. 2. Each data point on DASS is labelled with the II of function `g`. Each data point on SS is labelled with the loop II. η indicates the fractions of long latency operations.

based on the conditions given in Section IV-A, and the tool synthesises these functions from the original program into SS hardware that both access the same array. The DS analysis shows that these two functions have no memory conflict, as they never compute in parallel. The average results include an unsuitable benchmark 8) `bubbleSort` causing large bias on the average. Over the 10 benchmarks amenable for DASS, DS achieves 0.51×-0.85× of the execution time and 3.03× area, while DASS achieves 0.38×-0.63× of the execution time and 1.68× area. If the input data distribution is known, the design quality of the DASS hardware for all these benchmarks can be further improved (as in the case study 1).

C. Case Study 1: II Exploration

Now let us take the motivating example in Section II for a case study. We discuss how the variation of II affects the hardware performance and some principles that guide the selection of an appropriate II for the SS portion of a DASS circuit.

The Effects of II Selection: Let us first consider the case when the entire circuit is generated using SS. As an example, in Fig. 2 the minimum II of the loop in function `filterSum` is 5, because of a loop-carried dependency on `s` that takes 5 cycles. However, a user can also choose a larger II. This can lead to smaller area (because of more opportunities for resource sharing) but higher latency, as shown in Fig. 9 (blue circles). In this case, if the II is increased from 5 to 7, the LUT count is reduced by 28%, at the cost of increasing the latency by 39%.

Now let us consider the DASS case. For the example in Fig. 2, there are various choices of II for function `g`. The most aggressive solution is to set II = 1 for the highest possible throughput. However, due to the aforementioned loop-carried dependency on `s`, there is actually no performance benefit if the II is set to anything below 5—the loop-carried dependency dictates that the time between two calls to `g` is at least 5 cycles (when `g` is called from two consecutive loop iterations). The time between calls to `g` will only increase if the iterations that call it are further apart. Hence, if the user's primary objective is to maximise performance, an II of 5 cycles is sensible.

However, if the user knows more about the expected data distribution of the input, they can choose an even better II for `g`. Let us explore these effects on the motivating example. We denote the fraction of the loop iterations where `d >= 0` as η, since these have long latency through function `g` and the accumulation in `s`. The fraction of the iterations where `d < 0` is then 1 - η. The input data distribution affects η over all the loop iterations. Fig. 9 shows the LUT usage and wall clock time of the hardware by three scheduling approaches over several values of η, the fractions of long latency operations. The circuit is buffered for throughput by Dynamatic [32], and we assume the decision of the `if` condition is uniformly distributed over the iterations. Unbalanced structures are usually handled by the buffers, so here we only consider the average case. In the figure, it can be seen that the best II for function `g` varies in terms of the input data distribution. For example, if only the odd loop iterations are long and the rest are short, η is 0.5 and the optimal II for function `g` is 6.

More generally, a suitable II can be selected for a function `f`, where $1/II$ of a component can be considered as its maximum rate of processing data (also known as maximum throughput). The maximum II of function `f` that does not affect the overall program execution time is defined as its *DASS optimal II* (II_{opt}). It depends on two constraints, the maximum production rate allowed by its predecessors, $1/II_p$, and the maximum consumption rate allowed by its successors, $1/II_s$.

II Selection for the Motivating Example: Now let us show how to select the *DASS optimal II* (II_{opt}) for the motivating example. Other works have investigated these effects [33], [34], [32] in related problems. Here is an example of how it works. To analyse the circuit, we first show how to formalise the input rates and output rates for each component. Then we show how to use the formulation to find the *DASS optimal II* (II_{opt}) for the example.

There are two types of components in the DS circuit: non-control-flow components and control-flow components. The non-control-flow components only perform predictable operations like addition and multiplication. The control-flow components perform unpredictable operations like merge and branch.

For a non-control-flow component g that has N predecessors and M successors, let the actual data rate of its i^{th} predecessor be r_{pi}, the actual data rate of its i^{th} successor be r_{si}. Then we can see that all these rates are equal, where we introduce r to represent the value that these rates are all equal to:

$$\forall i, j \in [1, N],\ r_{pi} = r_{sj} = r$$

The reason is that the handshaking interface of the component stalls the inputs until all the inputs are valid and all the predecessors are ready. This balances the rates at the inputs and outputs. Also, the data rate is limited by the physical IIs of the component and all the surrounding hardware.

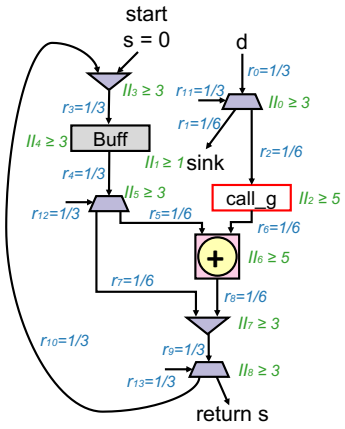


Fig. 10: Rate analysis for the motivating example.

$$r \leq \min(1/II_g, 1/II_{p1}, \ldots, 1/II_{pN}, 1/II_{s1}, \ldots, 1/II_{sM})$$

On the other hand, the processing rates of control flow components depend on the topology of the circuit and the input data distribution. The rate changes when the data goes through these components, as detailed below:

$$\text{Merge:} \quad r_{\text{out}} = r_{\text{in1}} + r_{\text{in2}}$$
$$\text{Branch:} \quad r_{\text{in1}} = r_{\text{in2}} = r_{\text{out1}} + r_{\text{out2}}$$

The rate analysis of a portion of the dataflow circuit from Fig. 3(b) is shown in Fig. 10. The green labels show the II constraints of each component and the blue labels represent the actual rate along each edge between two components. Due to the loop-carried dependency on the adder, where the output `s` is sent back to the input, the II of the circuit is limited by the latency of the feedback loop containing an adder and a buffer. That latency is 5 cycles, hence any value of the II of that adder smaller than 5 does not cause a performance bottleneck. In this case, the input and output rate of the adder is limited: $II_6 \geq 5$. Since there is no dataflow component in the path, $II_2 = II_6 \geq 5$. The top component consuming `d` is a *branch* component, which sends data to one of two outputs according to the `if` condition. The rate of a *branch* component with the loop-carried dependency has an additional constraint that:

$$1/r_{\text{branch,in1}} \geq \max(II_{\text{in}},\ II_{\text{out1}} \times p_1 + II_{\text{out2}} \times (1-p_1)) \quad (1)$$

where II_{in} is the II of its predecessor, II_{out1} is the II of its first successor, p_1 is the fraction of the data going into its first successor, and II_{out2} is the II of its second successor. In the figure, the predecessor is known not to be the bottleneck as the upper loop in Fig. 3(b) can feed `d` every clock cycle. In addition, one of the successors, *sink*, has $II_1 \geq 1$ as it can take data every clock cycle, and the other is function `g` with $II_2 \geq 5$. With half of the iterations being long, that is $p_1 = 0.5$, we have $II_0 \geq 0.5 \times 1 + 0.5 \times 5 = 3$. This means the highest overall rate is $r_0 = 1/3$, where the component consumes 1 set of data every 3 cycles on average. This agrees with the schedule in Fig. 2(c) that the hardware consumes 2 sets of data every 6 cycles and repeats. At the highest rate, the rate of the input is split into two edges through the *branch* component in terms of the fraction of the data going into the corresponding successor:

$$r_{\text{out1}} = p_1 \times r_{\text{in}}, \quad (2)$$
$$r_{\text{out2}} = (1 - p_1) \times r_{\text{in}} \quad (3)$$

In this case, $r_2 = r_0/2 = 1/6$. Similar analysis can be performed on other *branch* components in the circuit, resulting in the rate of each edge shown in Fig. 10. In conclusion, the rate to the function `g` is $1/6$ at the highest overall rate, and the *DASS optimal II* of function `g` is $II_{\text{opt}} = 6$. Smaller IIs may cause less area saving and larger IIs may cause performance degrading.

For all input data distributions in Fig. 9, the SS approach appears as a single line. The DS solution is always the same hardware architecture (*i.e.* constant LUT count) but with performance varying with the changing input data distribution. Our approach is shown as multiple green lines, one for each input data distribution. The design with *DASS optimal II*, shown as the elbows in the DASS lines, can have better performance than the DS hardware by improving the maximum clock frequency with SS implementation. In addition, the DASS hardware can also have comparable area efficiency compared to the SS hardware in terms of LUT and DSP usage.

By performing the rate analysis above, we have the *DASS optimal II* of function `g` equal to $II_{\text{opt2}} = 1/\eta + 4$. This can be justified as follows. Knowing $II_d = 1$, we have $II_0 = (1-\eta) \times II_1 + \eta \times II_2$. Then knowing $II_1 \geq 1$ and $II_2 \geq 5$, we have $II_0 \geq 1 + 4L$. For best performance, $II_0 = 1 + 4\eta$. Ultimately, knowing $r_0 = 1/II_0, r_2 = r_0 \times \eta$ and $r_2 = 1/II_{\text{opt2}}$, we have $II_{\text{opt2}} = 1/\eta + 4$, as required.

For instance, at $\eta = 1$, the *DASS optimal II* is 5, the same as the minimum loop II from SS. When $\eta = 0$, function `g` and the adder for `+=` are never used, so the II of the function does not affect the latency of the whole program. In this case, the *DASS optimal II* is infinity, *i.e.* function `g` is no longer needed.

In this work, we let users manually determine the optimal II for the SS function in the DASS hardware. In general, finding the optimal II for an SS function can be difficult as it depends on both the topology of the circuit and the input data distribution. However, even if users have only some information on the circuit, such as the minimum II achievable due to loop-carried dependencies, the hardware optimisation is still promising. In the figure, the DASS hardware with $II = 5$ for the SS function does not have minimum area but still achieves significant area reduction compared to the DS hardware. Although the hardware is underperforming, the difference in area reduction by switching from $II = 5$ to $II = 6$ is significantly smaller than that by switching from $II = 1$ to $II = 2$.

D. Case Study 2: BubbleSort

There are two throughput overheads caused by DASS. These overheads are usually minor but can still affect the

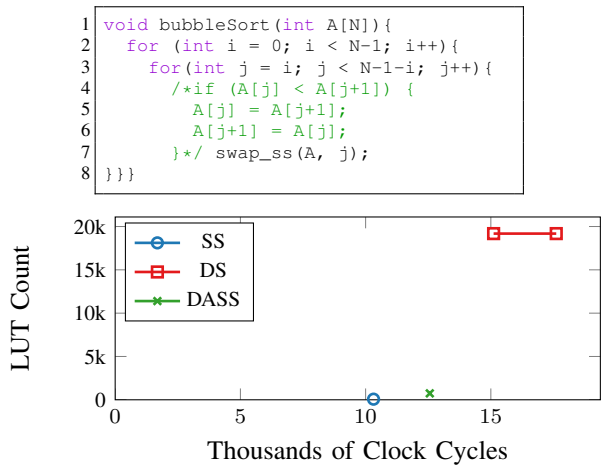


Fig. 11: Design quality of different scheduling approaches for `bubbleSort` benchmark. Bad choice may result in worse performance and larger area.

design quality with a bad choice of source for SS functions. Here we take `bubbleSort` for example, as illustrated in Fig. 11. The swapping process is chosen to be a SS function named `swap_ss`. The first overhead is caused by the inter-iteration dependence inside `swap_ss`. A load from `A[j]` in an iteration depends on the conditional store to `A[j+1]` in the last iteration. This memory dependence forces the schedule `swap_ss` to be sequential to preserve correctness. Second, the functionalisation of SS hardware instead of inlining causes one cycle of additional latency. This additional latency is usually hidden in a pipeline. In `bubbleSort`, the sequential schedule of `swap_ss` causes throughput of the loop depending on the latency of `swap_ss`, slowing down the computation.

The bottom of Fig. 11 shows how how these overheads affect the performance. Both axes are in log scale. In the figure, it can be seen that the optimal choice for `bubbleSort` is to statically schedule the whole program. Compared to the SS hardware, the DASS hardware has lower throughput due to the additional latency caused by the wrapper. It also loses in the circuit area due to the handshaking interface in the DS circuit. In addition, although the DS hardware can solve the memory dependence using an LSQ. However, the LSQ has a large area and a long memory latency. The memory latency is usually hidden in the pipeline but not for `bubbleSort` as explained before. Such a long latency results in a low throughput in the DS hardware. Compared to the DS hardware, the DASS hardware does not have an LSQ and has a smaller memory latency since all the memory accesses are statically-scheduled.

VII. Conclusion

In high-level synthesis, dynamic scheduling is useful for handling irregular and control-dominated applications. On the other hand, static scheduling can benefit from powerful optimisations to minimise the critical path and resource requirements of the resulting circuit. In this work, we combine existing dynamic and static HLS approaches to strategically replace regions of a dynamically scheduled circuit with their statically scheduled equivalents: we benefit from the flexibility of dynamic scheduling to achieve high throughput as well as the frequency and resource optimisation capabilities of static scheduling to achieve fast and area-efficient designs.

Across a range of benchmark programs that are amenable to DASS, our approach on average saves 45% of area in comparison to the corresponding dynamically scheduled design, and results in 1.98× execution time speedup over the corresponding statically scheduled design. In certain cases, the knowledge of the input data distribution allows us to further increase the design quality and may result in additional performance and area improvements. Our current approach relies on the user to annotate via pragmas parts of the code which do not benefit from dynamic scheduling and can, therefore, be replaced with static functions. The current version of DASS only support pipeline-related pragmas. Our future work will support more pragmas and explore the automated recognition of such code and these pragmas.

Acknowledgment

This work is supported by the EPSRC (EP/P010040/1, EP/R006865/1), the Royal Academy of Engineering, and Imagination Technologies. Lana Josipović is supported by a Google PhD Fellowship in Systems and Networking.

References

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. Monterey, CA, USA: ACM, 2011, pp. 33–36.

[2] V. G. Castellana, A. Tumeo, and F. Ferrandi, "High-level synthesis of memory bound and irregular parallel applications with bambu," in *2014 IEEE Hot Chips 26 Symposium (HCS)*. Cupertino, CA: IEEE, Aug 2014, pp. 1–1.

[3] Xilinx Vivado HLS, 2017. [Online]. Available: https://www.xilinx.com/

[4] Intel HLS Compiler, 2017. [Online]. Available: https://www.altera.com/

[5] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. Monterey, CA: ACM, 2018, pp. 127–136.

[6] J. Cheng, L. Josipović, P. Ienne, G. Constantinides, and J. Wickerson, "Combining dynamic & static scheduling in high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. Monterey, CA: ACM, 2020.

[7] J. Cheng, "HLS-benchmarks," 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3561115

[8] "Datasets for Combining Dynamic & Static Scheduling in High-level Synthesis," 2019. [Online]. Available: http://doi.org/10.5281/zenodo.3406553

[9] "DSS: Combining Dynamic & Static Scheduling in High-level Synthesis," 2019. [Online]. Available: https://github.com/JianyiCheng/DSS

[10] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8–17, July 2009.

[11] V. J. M. III and G. D. Micheli, "Hardware/software co-design of run-time schedulers for real-time systems," *Design Automation for Embedded Systems*, vol. 6, no. 1, pp. 89–144, Sep 2000.

[12] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *2006 43rd ACM/IEEE Design Automation Conference*. San Francisco, CA: IEEE, 2006, pp. 433–438.

[13] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. San Jose, CA: IEEE, 2013, pp. 211–218.

[14] Ian Page and Wayne Luk, "Compiling occam into Field-Programmable Gate Arrays," in *FPGAs, W. Moore and W. Luk, Eds., Abingdon EE&CS Books*, 1991.
[15] Celoxica, "Handel-C," 2005. [Online]. Available: http://www.celoxica.com
[16] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein, "C to asynchronous dataflow circuits: An end-to-end toolflow," in *IEEE 13th International Workshop on Logic Synthesis (IWLS)*. Temecula, CA: IEEE, Jun 2004.
[17] M. Budiu and S. C. Goldstein, "Pegasus: An efficient intermediate representation," Carnegie Mellon University, Tech. Rep. CMU-CS-02-107, May 2002.
[18] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.
[19] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 125:1–125:19, Sep. 2017.
[20] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne, "Shrink it or shed it! minimize the use of LSQs in dataflow designs," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 197–205.
[21] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in High-Level Synthesis," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. Austin, TX: IEEE, May 2013, pp. 51:1–51:10.
[22] J. Liu, S. Bayliss, and G. A. Constantinides, "Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. Vancouver, BC: IEEE, May 2015, pp. 159–162.
[23] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "ElasticFlow: A complexity-effective approach for pipelining irregular loop nests," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Austin, TX: IEEE, Nov 2015, pp. 78–85.
[24] S. Dai, M. Tan, K. Hao, and Z. Zhang, "Flushing-enabled loop pipelining for high-level synthesis," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. San Francisco, CA: IEEE, June 2014, pp. 1–6.
[25] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. Monterey, CA: ACM, 2017, pp. 189–194.
[26] L. P. Carloni, "From latency-insensitive design to communication-based system-level design," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2133–2151, Nov 2015.
[27] Charles Seitz, *System Timing*, 1980.
[28] J. Duprat and J.-M. Muller, "The CORDIC algorithm: New results for fast VLSI implementation," *IEEE Trans. Comput.*, vol. 42, no. 2, pp. 168–178, Feb. 1993.
[29] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Rethinking Inference in FPGA Soft Logic," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. San Diego, CA: IEEE, 2019, pp. 26–34.
[30] F. H. McMahon, "The Livermore Fortran kernels: A computer test of the numerical performance range," Lawrence Livermore National Lab., CA (USA), Tech. Rep., Dec. 1986.
[31] T. Young-Schultz, L. Lilge, S. Brown, and V. Betz, "Using OpenCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 86–96. [Online]. Available: https://doi.org/10.1145/3373087.3375300
[32] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. Monterey, CA: Association for Computing Machinery, 2020, p. 186–196. [Online]. Available: https://doi.org/10.1145/3373087.3375314
[33] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. Samos, Greece: IEEE, July 2011, pp. 404–411.
[34] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij, "Throughput analysis of synchronous data flow graphs," in *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*. Turku, Finland: IEEE, June 2006, pp. 25–36.

Jianyi Cheng (S'20) received an MSc in Analogue and Digital Integrated Circuit Design from Imperial College London in 2018 and a BEng in Electrical and Electronic Engineering from University of Nottingham in 2017. Currently, he is a PhD student in Electrical and Electronic Engineering from Imperial College London. His research interests include reconfigurable computing, high-level synthesis, program analysis and formal verification. He is a Student Member of the IEEE and the ACM.

Lana Josipović (S'16) received a MSc (2015) and BSc (2013) in Electrical Engineering and Information Technology from the University of Zagreb, Croatia. Currently, she is a PhD student in Computer and Communication Sciences at EPFL, Switzerland. Her research interests include high-level synthesis, compilers, and reconfigurable computing. She is a recipient of the Google PhD Fellowship for Systems and Networking and the Best Paper Award at FPGA'20.

George A. Constantinides (S'96, M'01, SM'08) received the Ph.D. degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Royal Academy of Engineering / Imagination Technologies Research Chair, Professor of Digital Computation, and Head of the Circuits and Systems research group. He has served as chair of the FPGA, FPL and FPT conferences. He currently serves on several program committees and has published over 150 research papers in peer refereed journals and international conferences. Prof Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.

Paolo Ienne (S'90, M'96, SM'10) received the *laurea* degree in Electrical Engineering from *Politecnico di Milano*, Italy, in 1991 and the Ph.D. degree in Computer Science from EPFL, Switzerland, in 1996. Since 2000, he has been a Professor with the School of Computer and Communication Sciences, EPFL. He serves on the steering committee of the ARITH, FPL, and FPGA conferences, and is an associate editor of ACM CSUR and ACM TACO. Ienne has published over 200 articles in peer reviewed journals and international conferences, some of which have received the Best Paper Awards at prestigious venues (including the FPGA, FPL, CASES, and DAC conferences). He is a Senior Member of the IEEE and a Member of the ACM.

John Wickerson (M'17, SM'19) received a Ph.D. in Computer Science from the University of Cambridge in 2013. He is a Lecturer in the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include high-level synthesis, the design and implementation of programming languages, and software verification. He is a Senior Member of the IEEE and a Member of the ACM.