Combining Dynamic & Static Scheduling in High-level Synthesis

Jianyi Cheng
Imperial College London
London, UK
jianyi.cheng17@imperial.ac.uk

Lana Josipović
EPFL
Lausanne, Switzerland
lana.josipovic@epfl.ch

George A. Constantinides
Imperial College London
London, UK
g.constantinides@ic.ac.uk

Paolo Ienne
EPFL
Lausanne, Switzerland
paolo.ienne@epfl.ch

John Wickerson
Imperial College London
London, UK
j.wickerson@imperial.ac.uk

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.

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; **Logic synthesis**; **Modeling and parameter extraction**.

KEYWORDS

High-Level Synthesis, Static Analysis, Dynamic Scheduling

ACM Reference Format:
Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-level Synthesis. In *2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375297

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
FPGA '20, February 23–25, 2020, Seaside, CA, USA.
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7099-8/20/02.
https://doi.org/10.1145/3373087.3375297

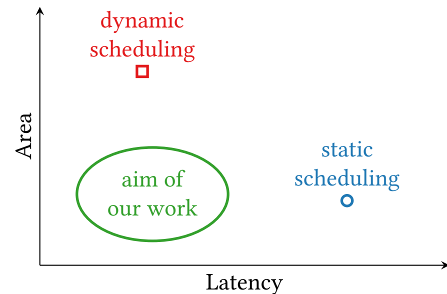


Figure 1: A sketch comparing the design quality of different scheduling approaches.

1 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 [3, 6] and industry [20, 28].

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.

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 paper, we propose dynamic and static scheduling (DSS): 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 the remainder of the program.

Several challenges must be overcome to make this marriage work. These include: (1) How should one pick suitable loop initiation intervals (IIs) when scheduling the SS parts of the circuit? (2) How should statically scheduled parts be correctly and efficiently integrated into their dynamically scheduled surroundings?

In this paper, we show how these challenges can be overcome, and that, once we have done so, 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. Over several benchmarks, we show that by using DS we can obtain hardware that is 23% faster than SS but uses 182% more area, and by using DSS we can obtain hardware that is 75% faster than SS and only uses 40% more area.

Paper outline. In Section 2, 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 3 provides a primer on existing SS and DS techniques. In Section 4, we describe how our proposal overcomes challenges related to II selection and component integration. Section 5 details a prototype implementation of DSS that uses Xilinx Vivado HLS [28] for SS and Dynamatic [22] for DS. In Section 6, we evaluate the effectiveness of DSS on a set of benchmarks and compare the results with the corresponding SS-only circuits and DS-only circuits.

Auxiliary material. All of the source code of benchmarks and the raw data from our experiments are publicly available [9, 11]. Our prototype tool, which relies on the Vivado HLS and Dynamatic HLS tools, is also open-sourced [10].

2 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, DSS.

SS can lead to small area but low performance. The hardware resulting from SS is shown in Fig. 2(b). 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; that is, using multiplexers to share a single operator among different sets of inputs.

The timing diagram of the SS circuit is shown in Fig. 2(d). It is a pipelined schedule with II = 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.

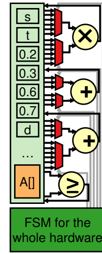
DS can lead to large area but high performance. The DS hardware is a dataflow circuit with a distributed control system containing several small components representing instruction-level operations [22], as shown on the left of Fig. 2(c). Each component is connected to its predecessors and successors 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(e). 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 = s_old`. 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.

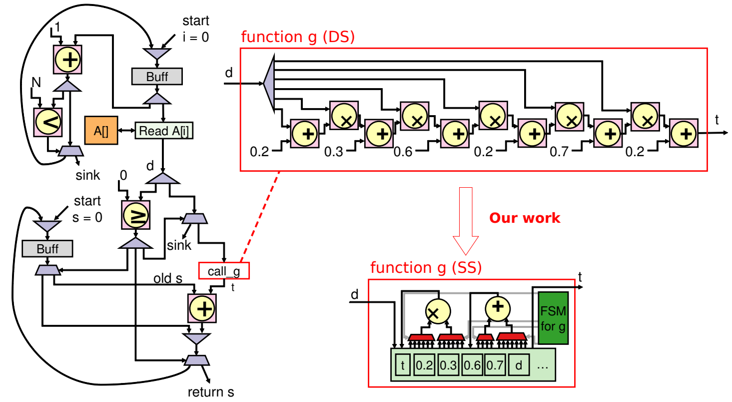
DSS can lead to small area and high performance. The DSS hardware combines the previous two scheduling techniques. It is based on the observation that although the overall circuit's performance benefits from DS, the function `g` does not because it has a fixed latency. Therefore, we replace the dataflow 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

```
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 }
```

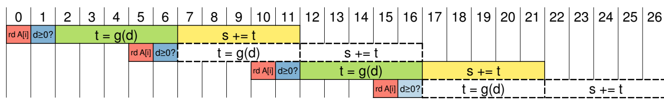
(a) Motivation code



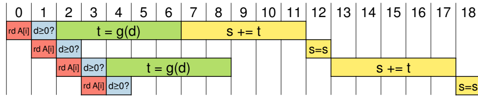
(b) SS circuit



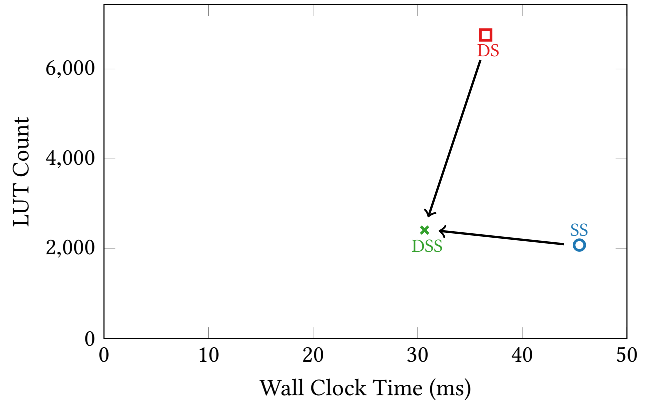
(c) DS circuit and its transformation into a DSS circuit



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



(e) The DS circuit and the DSS circuit have the same schedule (II_g in DSS = 1).



(f) LUT usage of different scheduling approaches over the performance.

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

to the area of the DSS hardware being close to that of the pure SS hardware, as shown in Fig. 2(f).

The timing diagram of the DSS circuit is the same as that of the DS circuit, as shown in Fig. 2(e).[1] In the DS circuit, g's schedule is determined at run-time, while in the DSS 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 DSS 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 DSS hardware can actually run at a higher clock frequency. Therefore, in this example, DSS 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. 2(f).

[1]Actually, the latency of function g varies slightly between DS and SS for technical reasons, as explained in Section 6.

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.

3 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.

3.1 Scheduling in HLS

In most HLS tools like LegUp [3] and Vivado HLS [28], 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 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) [13]. 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.

3.2 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 [19]. 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 [28] and LegUp [3], 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 [29]. 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.

3.3 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. 2. 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.

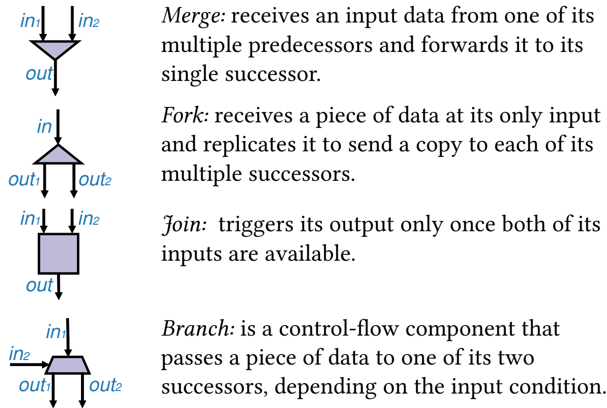
Initial work on dynamically scheduled hardware synthesis from a high-level language proposed a framework for automatically mapping a program in occam into a synchronous hardware netlist [18]. This work was later extended to a commercial language named Handel-C [7]. However, it still required the designer to select the constraints to achieve hardware optimisation such as pipelining and parallelism. Venkataramani *et al.* [26] proposed a framework that automatically transforms a C program into an asynchronous circuit. They implement each node in a DFG of Pegasus [2] 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 [22] proposes a toolflow 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.* [5], dynamic scheduling is typically implemented by dataflow circuits which consist of components communicating using handshake signals. Apart from all common hardware operators, a dynamically scheduled dataflow circuit contains a number of dataflow components shown in Tab. 1 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 use load-store queues (LSQs) [21] to resolve data dependencies and appropriately schedule the arbitrary memory accesses at run-time. 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 issue in Section 4.3.

3.4 Combining Dynamic & Static Approaches

Several works have explored the integration of certain aspects of dynamic scheduling into static HLS. Alle *et al.* [1] and Liu *et al.* [23] propose source-to-source transformations to enable multiple schedules selected at run-time after all the required values are known. Tan *et al.* [25] propose an approach named ElasticFlow to optimise pipelining of irregular loop nests that contain dynamically-bound inner loops. Dai *et al.* [14, 15] 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



Component	Description
	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.

Table 1: Dataflow components in DS circuits.

adds existing hardware optimisation techniques into dynamically scheduled circuits and supports arbitrary code input.

Carloni [4] describes 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.

4 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 explain the selection of II for the SS hardware. Finally, we 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.

4.1 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 3.3. 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.

4.2 II Selection for SS Hardware

When using SS, there are often several options for the II of a function or a loop. In this subsection, we discuss some principles that guide how to select an appropriate II for the SS portion of a DSS circuit.

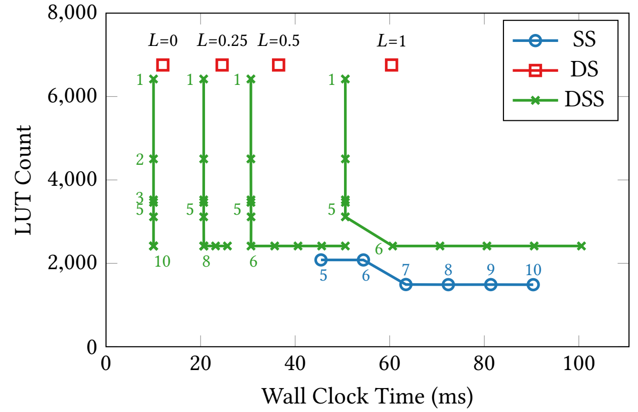


Figure 3: LUT usage of different scheduling approaches over the performance for the example from Fig. 2. Each data point on DSS is labelled with the II of function g. Each data point on SS is labelled with the loop II. L indicates the fractions of long latency operations.

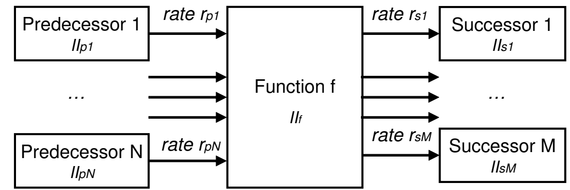


Figure 4: Rate balancing of components in a DS circuit by selecting proper II for function `f`.

Let us consider the case when the entire circuit is generated using SS first. 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. 3 (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 DSS 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 of `g` is at least 5 cycles (when `g` is called from two consecutive loop iterations). The time between calls of `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 define the fraction of the loop iterations where `d >= 0` as to be L, since these have long latency through function `g` and the summation to `s`, and

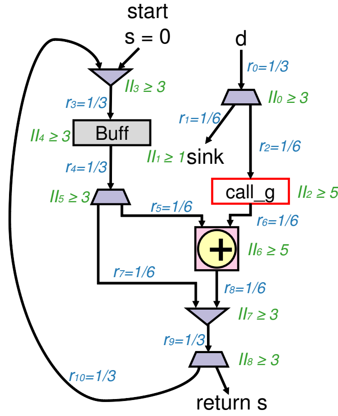


Figure 5: Rate analysis for the motivating example.

the fraction of the iterations where `d < 0` as 1 - L. The data distribution affects L over all the loop iterations. Fig. 3 shows the LUT usage and wall clock time of the hardware by three scheduling approaches over several input data distributions. We assume the circuit is ideally buffered so the data is fed in a uniform distribution. In the figure, it can be seen that the best II for function `g` varies in terms of the input data distribution. For example, in Section 2, where only the odd loop iterations are long and the rest are short, L 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 *DSS optimal II* (II_{opt}). It depends on two constraints, the maximum producing rate allowed from its predecessors, $1/II_p$, and the maximum consuming rate allowed by its successors, $1/II_s$.

Fig. 4 shows function `f` as a component in a DS circuit like Fig. 2(e). It has N predecessors to produce data to the function and M successors to take data from the function. Let the actual data rate between function `g` and its ith predecessor be r_{pi}, the actual data rate between function `g` and its ith successor be r_{si}, and the II of function `f` be II_f. Then we can see from the figure that:

$$\forall i \in [1, N],\ r_{pi} \leq \min(1/II_{pi}, 1/II_f) \tag{1}$$
$$\forall j \in [1, M],\ r_{sj} \leq \min(1/II_{sj}, 1/II_f) \tag{2}$$
$$\forall i \in [1, N],\ r_{\text{overall}} \leq r_{pi} \tag{3}$$
$$\forall j \in [1, M],\ r_{\text{overall}} \leq r_{sj} \tag{4}$$

Assuming no other component in the system limits the overall rate of function `f`, shown as r_{overall}, this rate can be expressed as the minimum value of all r_p and r_s. To maximise hardware performance, we want r_{overall} to be high. The external II coefficient II_{ext} refers to the maximum II among all the IIs in the graph except function `f`:

$$II_{\text{ext}} = \max(II_{p1}, \ldots, II_{pN}, II_{s1}, \ldots, II_{sM}) \tag{5}$$

When $1/II_f > 1/II_{\text{ext}}$, function `f` is overperforming and it consumes unnecessary hardware resources. In contrast, when $1/II_f < II_{\text{ext}}$, although function `f` has small area, it also becomes the performance bottleneck, limiting the overall rate. Knowing such a trade-off between area and performance, the *DSS optimal II* of function `f`, $II_{\text{opt}} = 1/II_{\text{ext}}$. This results in the actual overall rate $r_{\text{overall}} = 1/II_{\text{opt}}$ for the example in Fig. 4. Similar analysis can be extended to the whole dataflow graph.

The processing rates of dataflow components depend on the topology of the circuit and the input data distribution. Other works have investigated these effects [17, 24] and their formal definition is therefore out of the scope of this work. Here, we only give an example of the rate analysis for the motivating example in Fig. 2.

The rate analysis of a portion of the dataflow circuit from Fig. 2(c) is shown in Fig. 5. The green labels show the II constraints of each component and the blue labels represent the actual rate along each edge between two components. The rate changes when the data goes through the dataflow components shown in Section 3.3, as detailed below:

$$\text{Merge:} \quad r_{\text{out}} = r_{\text{in1}} + r_{\text{in2}} \tag{6}$$
$$\text{Fork:} \quad r_{\text{in}} = r_{\text{out1}} = r_{\text{out2}} \tag{7}$$
$$\text{Join:} \quad r_{\text{out}} = r_{\text{in1}} = r_{\text{in2}} \tag{8}$$
$$\text{Branch:} \quad r_{\text{in1}} = r_{\text{in2}} = r_{\text{out1}} + r_{\text{out2}} \tag{9}$$

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 that is no greater than 5 does not affect the performance. 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 II of a *branch* component can be determined by:

$$II_{\text{branch}} \geq \max(II_{\text{in}},\ II_{\text{out1}} \times p_1 + II_{\text{out2}} \times (1 - p_1)) \tag{10}$$

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. 2(c) 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(e) 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:

$$r_{\text{out1}} = p_1 \times r_{\text{in}}, \tag{11}$$
$$r_{\text{out2}} = (1 - p_1) \times r_{\text{in}} \tag{12}$$

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. 5. In conclusion, the rate to the function `g` is 1/6 at the highest overall rate, and the *DSS 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. 3, 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 *DSS optimal II*, shown as the elbows in the DSS lines, can have better performance than the DS hardware by improving the maximum clock frequency with SS implementation. In addition, the DSS 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 *DSS optimal II* of function g equal to $II_{\text{opt2}} = 1/L + 4$:

$$\text{Knowing } II_d = 1,\ II_0 = (1 - L) \times II_1 + L \times II_2 \quad (13)$$

$$\text{Knowing } II_1 \geq 1 \text{ and } II_2 \geq 5,\ II_0 \quad (14)$$

$$\text{For best performance, } II_0 = 1 + 4L \quad (15)$$

$$r_0 = 1/II_0, r_2 = r_0 * L, r_2 = 1/II_{\text{2opt}} \Rightarrow II_{\text{opt2}} = 1/L + 4 \quad (16)$$

For instance, at $L = 1$, the *DSS optimal II* is 5, the same as the minimum loop II from SS. When $L = 0$, 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 *DSS optimal II* is infinity, *i.e.* function g is no longer needed.

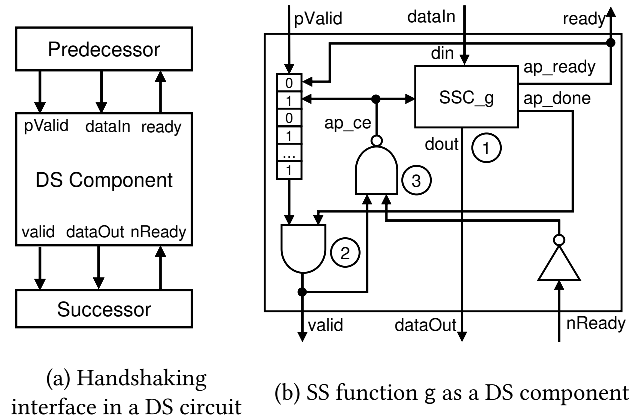
In this work, we let users manually determine the optimal II for the SS function in the DSS hardware. In general, finding the optimal II for an SS function can be difficult as it depends on both the topology of the circuit and the input data distribution. However, even if users have only some information on the circuit, such as the minimum II achievable due to loop-carried dependencies, the hardware optimisation is still promising. In the figure, the DSS 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 $II = 2$.

4.3 Integrating SS Hardware into DS Hardware

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, indicated in Fig. 2(c). In this subsection, 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 Section 2 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. 6(a). Each DS component uses the bundled data protocol [8] for communication, where each data connection has request and acknowledgement signals. For instance, the following is the control interface of a component from Dynamatic [22]:

- *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.



(a) Handshaking interface in a DS circuit

(b) SS function g as a DS component

Figure 6: The statically scheduled (SS) circuit of function g is wrapped with additional control circuitry for interfacing to the DS circuit.

On the other hand, the traditional SS hardware has a different interface only to monitor and control the states of the centralised FSM. An example of an HLS tool that generates SS hardware is Vivado HLS [28]. For a function named g 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_done:* The done signal indicates the output from SS hardware at the current clock cycle is valid.

The interface synthesis 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. A *valid* signal is designed 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, a *ready* signal is designed 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.

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. 6(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_done* 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 allow 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 apply similar methodology from the dataflow circuit [22] to synchronise data with the help of *join* components. A *join* component is used for preserving the valid inputs to the SS circuit until all the inputs are valid. This is similar to a DS component with multiple inputs.

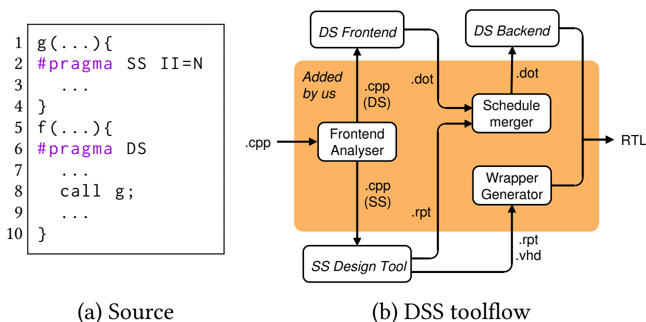
For multiple outputs in the SS function, each component has its own handshaking signals. The output handshaking signals are implemented in two parts, the *valid* and *nReady* signals. Firstly, each data has its own *valid* signal. The SS circuit with multiple outputs has the same number of *ap_vld* signals, where each *ap_vld* signal indicates whether the corresponding data is valid. Instead of forwarding the *ap_done* signal to the *AND* gate at the bottom left of Fig. 6(b), each *ap_vld* signal is *AND*ed with the oldest bit of the shift register and forwarded to the corresponding *valid* signal. The number of *nReady* signals is the same as the number of successors. All the *nReady* signals of the wrapper are inversed and connected to the *NAND* gate in Fig. 6(b), such that the backpressure from any successor can stall the SS hardware.

Handling memory accesses. The SS hardware has its scheduled memory accesses and can directly interact with the memory, while the DS hardware requires an LSQ [21] to schedule the memory accesses at run-time before accessing the data. In DSS hardware, a combination of two memory architectures needs to be handled. An LSQ is beneficial for programs that have irregular memory accesses and can have high performance using dynamic scheduling as explained in Section 3.3. It does not bring performance improvements when implementing regular computation in the form of an SS circuit. Therefore, in this work, we only allow conflict-free memory accesses between the SS part and the DS part; that is, LSQs are not needed, or are only responsible for the DS part, while the SS part has its own memory ports.

Summary. We identify code that is amenable for DSS, where the design quality of the resulting hardware can be improved. In addition, we show that rate analysis can further improve area efficiency and performance of the DSS hardware. With optimal II of the internal SS function, the DSS hardware can not only have comparable performance to the DS hardware but comparable area efficiency to the SS hardware. With our wrapper, the SS hardware can work correctly in a DS circuit.

5 IMPLEMENTATION

Our approach is generic and can be used with various SS and DS HLS tools. For our work, we choose Vivado HLS [28] and Dynamatic [22] to synthesise the SS and DS hardware respectively. Our toolflow is shown in Fig. 7. The user-defined scheduling constraints are configured using *pragmas*. Our tool takes the input C++ code and splits the functions into two groups, representing the SS and the DS functions, as specified by the user. We synthesise a function without



(a) Source (b) DSS toolflow

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

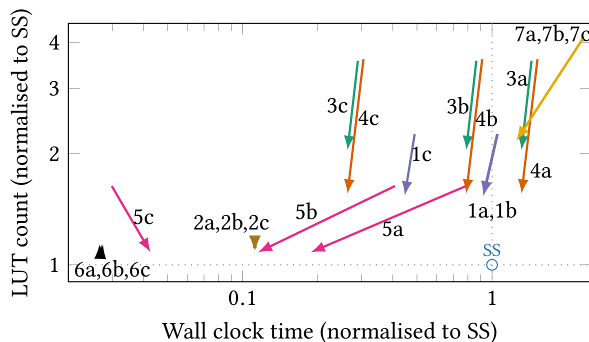


Figure 8: The overall effects of our approach over seven benchmarks in three certain data distributions: all long (a), half long-half short (b) and all short (c). The arrows show the improvements by switching from DS to DSS. Each label corresponds to a benchmark in Tab. 2 and its distribution.

any scheduling constraints to DS hardware by default. Our tool supports the integration of multiple SS functions into a DS function. The SS functions are synthesised by Vivado HLS with a user-defined II. Any SS function without an II specification is synthesised with the optimal II determined by Vivado HLS. The resultant SS hardware is then automatically wrapped up to ensure compatibility with the DS circuit interface, as described in Section 4.3. Each input variable or output variable of the function is constructed as a data port with a set of handshaking signals. Any memory port for array accesses from the SS function is directly forwarded to the memory block.

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 DSS hardware of the top function.

6 EXPERIMENTS

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

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 DSS 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

6.1 Benchmarks

We apply our approach to seven benchmarks:

(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) *gSum* 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) *gSumIf* 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 [16] and a polynomial function.
(6) *getTanh(double)* is similar but uses an array of doubles.
(7) *BNNKernel* is a small binarised neural network [27].

The benchmark programs are selected based on the features where DS can bring the performance benefits, indicated in Section 4.1. 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.* [22]. The two *getTanh* benchmarks apply the existing approximation algorithms on sparse data arrays. These benchmarks are all made publicly available [11].

6.2 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. 2. In the figure, we show three arrows for each benchmark: the best case (all inputs

Table 2: Evaluation of design quality of DSS over seven benchmarks. Assuming the data distribution is unknown, the II of the static function in DSS is selected as the II in the worst case (L = 1).

Benchmarks		DSS II	LUT			DSP			Registers			Total Cycles			Fmax/MHz			Wall Clock Time/us		
			SS	DS	DSS	SS	DS	DSS	SS	DS	DSS	SS	DS	DSS	SS	DS	DSS	SS	DS	DSS
1●	sparseMatrixPower	299	206	465	317	3	6	3	191	489	198	306-30706	141-30005	141-29704	64.9	60.2	67.8	4.7-473.1	2.3-498.3	2.1-437.8
2●	histogram	1	902	1002	990	3	3	3	639	637	809	9008	1008-1015	1008-1013	111.4	111.4	111.4	80.9	9.0-9.1	9.0-9.1
3●	gSum	5	2209	7874	4514	17	79	23	2592	5552	3960	5075	1017-5067	1017-5079	111.4	77.3	84.9	45.6	13.2-65.6	12-59.8
4●	gSumIf	5	3352	12068	5222	31	152	37	3903	9440	5188	5075	1017-5069	1017-5081	111.4	73.2	85.0	45.6	13.9-69.3	12-59.8
5●	getTanh	11	3768	6154	4072	6	12	6	2172	6418	2422	55007	2508-65989	2508-10993	42.4	64.7	45.2	1298.7	38.8-1020.4	55.5-243.2
6●	getTanh(double)	1	2272	2453	2579	50	50	50	2236	2154	2797	38007	1010-1035	1010-1042	111.4	111.4	111.4	341.2	9.1-9.3	9.1-9.4
7●	BNNKernel	303 & 402	306	1250	664	3	9	3	142	1606	519	30908	30407	30412	143.3	61.1	112.8	215.7	498	270
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**

take the short path), the worst case (all long), and a middle case (half short, half long)[2]. 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 DSS hardware is at the end of the arrow. The II of the SS function in the DSS hardware is chosen only considering the worst case of the execution patterns, where all the iterations are long, that is $L = 1$, 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 DSS designs sit at the top left of the corresponding SS hardware. In addition, for the same benchmark, most DSS hardware designs are on the bottom left of the DS hardware. It shows that the DSS hardware can be smaller than the DS hardware and have better performance.

Detailed results of these benchmarks are shown in Tab. 2, considering all the cases of possible input distribution (from all long to all short). In general, 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 scheduler. However, in the case of *sparseMatrixPower*, 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 DSS 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 DSS hardware may also be different from the corresponding DS hardware, which also affects the critical path (like function g in Section 2). In some of the benchmarks like *gSum*, 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 *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 DSS. 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 *getTanh*, the DSS design benefits in cycle count by introducing the fully pipelined SS function. The benchmark *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. If the input data distribution is known, the design quality of the DSS hardware for all these benchmarks can be further improved (as in Section 4.2).

7 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 DSS, our approach on average saves 43% of area in comparison to the corresponding dynamically scheduled design, and results in 1.75× 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. Our future work will explore the automated recognition of such code and these pragmas.

ACKNOWLEDGEMENTS

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.

[2]In the figure, 2{a, b, c} are overlapped and shown as one arrow, so are 6{a, b, c} and 7{a, b, c}; also, 1a and 1b are overlapped as the SS hardware is unpipelined.

REFERENCES

[1] Mythri Alle, Antoine Morvan, and Steven Derrien. 2013. Runtime dependency analysis for loop pipelining in High-Level Synthesis. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, Austin, TX, 51:1–51:10.

[2] Mihai Budiu and Seth Copen Goldstein. 2002. *Pegasus: An Efficient Intermediate Representation*. Technical Report CMU-CS-02-107. Carnegie Mellon University. 20 pages.

[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, Monterey, CA, USA, 33–36.

[4] Luca P. Carloni. 2015. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proc. IEEE* 103, 11 (Nov 2015), 2133–2151. https://doi.org/10.1109/JPROC.2015.2480849

[5] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (Sep. 2001), 1059–1076. https://doi.org/10.1109/43.945302

[6] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. 2014. High-level synthesis of memory bound and irregular parallel applications with Bambu. In *2014 IEEE Hot Chips 26 Symposium (HCS)*. IEEE, Cupertino, CA, 1–1.

[7] Celoxica. 2005. Handel-C. http://www.celoxica.com

[8] Charles Seitz. 1980. *System Timing*.

[9] Jianyi Cheng. 2019. Datasets for Combining Dynamic & Static Scheduling in High-level Synthesis. http://doi.org/10.5281/zenodo.3406553

[10] Jianyi Cheng. 2019. DSS: Combining Dynamic & Static Scheduling in High-level Synthesis. https://github.com/JianyiCheng/DSS

[11] Jianyi Cheng. 2019. HLS-benchmarks. https://doi.org/10.5281/zenodo.3561115

[12] Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *2006 43rd ACM/IEEE Design Automation Conference*. IEEE, San Francisco, CA, 433–438.

[13] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. 2009. An Introduction to High-Level Synthesis. *IEEE Design Test of Computers* 26, 4 (July 2009), 8–17.

[14] Steve Dai, Mingxing Tan, Kecheng Hao, and Zhiru Zhang. 2014. Flushing-enabled loop pipelining for high-level synthesis. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, 1–6.

[15] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. 2017. 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 (FPGA '17)*. ACM, Monterey, CA, 189–194.

[16] Jean Duprat and Jean-Michel Muller. 1993. The CORDIC Algorithm: New Results for Fast VLSI Implementation. *IEEE Trans. Comput.* 42, 2 (Feb. 1993), 168–178.

[17] Amir H. Ghamarian, Marc C. W. Geilen, Sander Stuijk, Twan Basten, Bart D. Theelen, Mohammad R. Mousavi, Arno J. M. Moonen, and Marco J. G. Bekooij. 2006. Throughput Analysis of Synchronous Data Flow Graphs. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*. IEEE, Turku, Finland, 25–36.

[18] Ian Page and Wayne Luk. 1991. Compiling occam into Field-Programmable Gate Arrays. In *FPGAs, W. Moore and W. Luk, Eds., Abingdon EE&CS Books*.

[19] Vincent John Mooney III and Giovanni De Micheli. 2000. Hardware/Software Co-Design of Run-Time Schedulers for Real-Time Systems. *Design Automation for Embedded Systems* 6, 1 (01 Sep 2000), 89–144.

[20] Intel HLS Compiler. 2017. https://www.altera.com/

[21] Lana Josipović, Philip Brisk, and Paolo Ienne. 2017. An Out-of-Order Load-Store Queue for Spatial Computing. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 125 (Sept. 2017), 19 pages.

[22] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. ACM, Monterey, CA, 127–136.

[23] Junyi Liu, Samuel Bayliss, and George A. Constantinides. 2015. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Vancouver, BC, 159–162.

[24] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, Samos, Greece, 404–411.

[25] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. 2015. ElasticFlow: A complexity-effective approach for pipelining irregular loop nests. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Austin, TX, 78–85.

[26] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. 2004. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow. In *IEEE 13th International Workshop on Logic Synthesis (IWLS)*. IEEE, Temecula, CA.

[27] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. 2019. LUTNet: Rethinking Inference in FPGA Soft Logic. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, San Diego, CA, 26–34.

[28] Xilinx Vivado HLS. 2017. https://www.xilinx.com/

[29] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, San Jose, CA, 211–218.