Finding and Finessing Static Islands in Dynamically Scheduled Circuits

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

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

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

ABSTRACT

In high-level synthesis, scheduling is the process that determines the start time of each operation in hardware. A hardware design can be scheduled either at compile time (static), run time (dynamic), or both. Recent research has shown that combining dynamic and static scheduling can achieve high performance and small area. However, there is still a challenge to determine which part to schedule statically and which part dynamically. An inappropriate choice can lead to suboptimal design quality. This paper proposes a heuristic-driven approach to automatically determine 'static islands' – *i.e.*, code regions that are amenable for static scheduling. Over a set of benchmarks where our approach is applicable, we show that our tool can achieve on average a 3.8-fold reduction in area combined with a 13% performance boost through automatic identification and synthesis of static islands from fully dynamically scheduled circuits. The performance of the resulting hardware is close to optimum (as determined by an exhaustive enumeration of all possible static islands).

ACM Reference Format:
Jianyi Cheng, John Wickerson, and George A. Constantinides. 2022. Finding and Finessing Static Islands in Dynamically Scheduled Circuits. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22), February 27-March 1, 2022, Virtual Event, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3490422.3502362

1 INTRODUCTION

High-level synthesis (HLS) is a process that automatically transforms programs in a high-level language like C/C++ into hardware descriptions in a low-level language like Verilog/VHDL. HLS tools ideally allow software engineers without hardware background to program custom hardware to achieve high performance. As a promising trend in recent years, various HLS tools have been developed in both academia, such as LegUp from the University of Toronto [5], Dynamatic from EPFL [31], and industry, such as Xilinx Vivado/Vitis HLS [47], Intel's HLS Compiler [27], Catapult HLS from Mentor [10] and Stratus HLS from Cadence [40].

Still, it remains the case that automatically synthesising high performance and area-efficient hardware from arbitrary high-level programs is challenging. In order to tackle this problem, techniques have been proposed to produce optimised hardware architectures for specific domains, such as image processing [41, 48], deep learning [38, 46, 50] and other matrix computation-based applications [21]. However, users of these tools are still required to have hardware background to write efficient code for optimal performance. The problem remains unsolved for general applications due to the presence of complex control flow that can only be partially parallelised [33].

Scheduling is one of the main steps in HLS. It determines the start time of each operation in the input code. Static scheduling enables high area efficiency, and dynamic scheduling enables high performance. Recent work known as 'DASS' by Cheng *et al.* [12] combines the best of both worlds by supporting statically scheduled (SS) hardware that forms internal components, named *static islands*, within dynamically scheduled (DS) hardware. However, this work left it to the user to determine which pieces of code in an arbitrary program should form the static islands and which parts should be dynamically scheduled. The primary contribution of this paper, then, is a heuristic-driven technique for finding good static islands in arbitrary code.

Our secondary contribution has to do with how these static islands, once identified, are interfaced with the surrounding DS circuit. Prior work assumes that when a static island has multiple inputs, all inputs are required at the start of computation [12]. This assumption can lead to observable inefficiency in the final circuit, because if some inputs are valid before the others, they must be held until the others are valid too, even if some of the static island's computation could proceed using only those inputs that are valid.

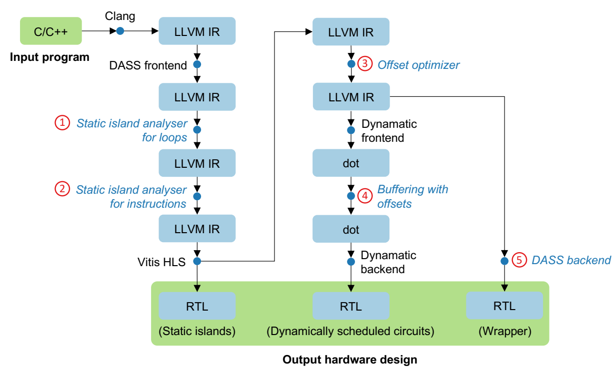


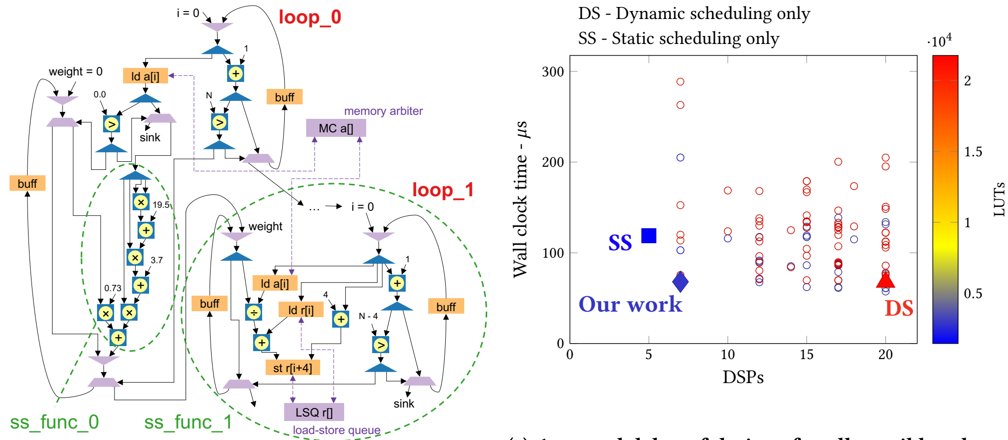
Figure 1: Our work integrated into the open source DASS tool from [12]. The steps numbered ① to ⑤ are contributions of this paper. Section 4.4 explains the details.

FPGA '22, February 27-March 1, 2022, Virtual Event, CA, USA

ACM ISBN 978-1-4503-9149-8/22/02…$15.00
https://doi.org/10.1145/3490422.3502362

```
1  float a[N], r[N];
2  void vecNormTrans() {
3    float weight = 0.0f;
4    loop_0:
5    for (int i=0; i<N; i++) {
6      float d = a[i];
7      if (d < 1.0f)
8        // weight = ss_func_0(
9        // d, weight);
10       weight = ((d*d+19.5)*d
11       +3.7)*d+0.73*weight;
12   }
13   // ss_func_1(weight);
14   loop_1:
15   for (int i=0; i<N-4; i++)
16     r[i+4] = r[i]+a[i]/w;
17 }
```

(a) Source



(b) Data flow graph

(c) Area and delay of designs for all possible selection of static islands

Figure 2: A normalised transformation kernel as a motivating example. There are 102 possible selections of static islands, and most of them lie at the top right of the figure (big & slow). The polynomial expression on line 10 and the loop named `loop_1` should be synthesised as two static islands. Automatically determining the optimal selection is challenging.

We remove this assumption made in [12], and we demonstrate in our evaluation that doing so can lead to a 2.6× performance improvement. As an analogy, we can imagine a timing diagram with time on the vertical axis and resource on the horizontal axis. In such a picture, the static islands introduced in [12] can be thought of as *rectangles* (where all the inputs are aligned on the top edge). Meanwhile, the static islands in this work have a less regular shape. We refer to the process of 'derectangularising' the islands as *finessing* them, because it is our method for squeezing better performance out of them.

Taken together, our two contributions extend the state of the art in HLS by automating the choice between static and dynamic scheduling and efficiently synthesising hardware that combines both scheduling approaches. In summary, this paper presents:

- a technique that finds an optimised allocation of static islands by analysing the features of each code region and evaluating the hardware performance by different scheduling strategies (① and ② in Figure 1),
- an automated HLS pass that calls the commercial tool Xilinx Vitis HLS to efficiently synthesise static islands with high performance by supporting non-simultaneous inputs (③, ④ and ⑤ in Figure 1), and
- analysis and results showing that the proposed approach achieves on average a 3.8-fold reduction in area combined with a 13% performance boost through automatic identification and synthesis of static islands compared to fully dynamically scheduled circuits. Compared to the fastest designs discovered via exhaustive search, our approach generates designs that are within 2% of the performance.

The rest of our paper is organised as follows. Section 2 gives a motivating example to illustrate the challenge in selecting scheduling approaches. Section 3 presents background in scheduling in HLS and related works. Section 4 formalises the problem and presents a tool to automatically find good static islands. Section 5 evaluates the effectiveness of our tool on a set of benchmarks.

Table 1: Comparison between our approach and baselines using static scheduling or dynamic scheduling only.

	LUTs	DSPs	Cycles	Fmax (MHz)	Time (μs)
Vitis HLS	1614	5	12327	120	102
Dynamatic	21290	20	6079	90	68
Our work	2860	7	6703	106	63

2 MOTIVATING EXAMPLE

In this section, we use a motivating example to demonstrate the challenge in selecting code regions for static scheduling within dynamic surroundings. Figure 2a shows a code example to be scheduled and synthesised. In the code, a loop named `loop_0` loads an array element `a[i]` and adds a Horner-style polynomial evaluated at `a[i]` onto a variable named `weight` if the array element is less than 1. The value of `weight` is then used in a subsequent loop named `loop_1` for the transformation of array `r`.

Figure 2b shows the data flow graph of the code in Figure 2a assuming it is fully dynamically scheduled using the approach presented in [31]. Recent work [12] proposes to statically schedule part of the data flow graph as individual static islands, indicated as dotted circles, for resource sharing inside each island. Combining dynamic and static scheduling techniques onto a single circuit can achieve both high performance and area efficiency. Dynamic scheduling is beneficial for input-dependent control flows, such as the `if` condition in `loop_0`, while static scheduling is beneficial for predictable data flow, such as the polynomial expression on line 10. The goal of our work is to determine all the best possible static islands under given performance constraints and efficiently synthesise them.

Automatically determining static islands is challenging. We enumerate all the possible sets of (non-overlapping) subgraphs (each of which must have at least two instructions to be worth statically scheduling) from the graph in Figure 2b, resulting in 102 designs in Figure 2c. The design that only uses dynamic scheduling is at the bottom right of the figure; it has low latency but large area. The design that only uses static scheduling is at the left of the figure; it has higher latency but smaller area. The other designs use both approaches, with different code regions using different approaches. Some designs perform much worse than either fully static or fully dynamic scheduling. For instance, the design at the top right of the figure has high latency and large area. This motivates us to formalise and automate the search for code features that are amenable for static scheduling as part of a fully-automated tool flow.

For this example, our tool suggests to statically schedule the polynomial expression on line 10 and the entire `loop_1` as shown in Figure 2b. This results in a design with similar performance to the DS design but similar area to the SS design, as indicated in Figure 2c. The absolute results are shown in Table 1.

The selection suggested by our tool results in a high-quality design for three reasons. First, the synthesised design still computes at a high throughput because the input-dependent `if` statement remains dynamically scheduled. Second, DSPs are significantly reduced due to resource sharing in the polynomial expression. Finally, instead of using area-expensive load-store queues (LSQs) to handle inter-iteration memory dependence for dynamic scheduling, statically scheduling `loop_1` saves many LUTs and still achieves the same throughput. In the rest of this paper, we will explain in detail how these features are modelled and optimised by our tool.

3 BACKGROUND

This section first reviews static scheduling and dynamic scheduling in HLS. Then we compare existing works on combining static and dynamic scheduling with our work. Finally, related works on module selection and optimisation are also reviewed.

3.1 Static Scheduling

Static scheduling is a process to determine the start time of each operation in the input program at compile time [26]. In static scheduling, the input code is translated into a control- and data-flow graph (CDFG) [18]. A CDFG has two levels. At the higher level, the graph illustrates control flows of a program. Each vertex represents a basic block (BB) in the code, and each edge represents a control transition between two BBs. Each of these vertices corresponds to a subgraph at the lower level. Each subgraph vertex represents an operation, and each edge between these vertices represents a data dependence between two operations. Given a CDFG, a 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 commonly used static scheduling techniques is to formulate and solve the problem as a system of difference constraints (SDC) [16]. This approach is used in popular HLS tools such as Vitis HLS [47] and LegUp [5]. This approach to scheduling has also been extended to pipelining to achieve high throughput loops and functions [4, 49].

A static scheduler can achieve efficient resource optimisation since the whole schedule is predictable at compile time. However, this methodology does not suit programs with input-dependent control flow. In this case, the scheduler has to assume the worst case to ensure correct results at the price of performance compared to dynamic scheduling.

3.2 Dynamic Scheduling

Dynamic scheduling is a process that schedules operations at runtime. Initial work on synthesising DS hardware from a high-level language proposed a framework for automatically mapping an occam program into a synchronous hardware netlist [25]. This work was later extended to a commercial language named Handel-C [11]. Venkataramani *et al.* [44] propose a framework that automatically transforms a C program into an asynchronous circuit. They implement each node in a data flow graph of Pegasus [3] into a pipeline stage. Recent work [31] proposes an HLS tool named Dynamatic that generates synchronous dataflow circuits from C code. Dynamatic takes arbitrary input code, automatically exploits the parallelism of the hardware and uses handshaking signals for dynamic scheduling to achieve high throughput. This approach is also realised in an MLIR-based HLS flow named CIRCT [17].

As formalised by Carloni *et al.* [7], dynamic scheduling extends the CDFG of the input code into a hardware-like data flow graph. In the data flow graph, each vertex represents a pre-defined hardware component, and each edge represents a handshake connection between two components. Apart from the operations that are directly mapped from the code, a set of elastic operations are placed in the data flow graph to achieve parallelism.

In this paper, we use Dynamatic HLS to generate DS hardware. Here we introduce a few elastic operations in Dynamatic that are related to our analysis:

(1) A *merge* receives an item of input data from one of its multiple predecessors, with a pre-defined priority order, and forwards it to its single successor.
(2) A *mux* is similar to a merge but the input data is chosen based on the select bit.
(3) A *branch* passes a piece of data to one of its two successors, depending on the input condition.

One difficulty in dynamic scheduling is scheduling memory accesses. CIRCT forces the memory accesses to be carried out in the same order as the program order, at the price of no memory parallelism. Dynamatic uses generic LSQs [30, 32] to monitor and schedule memory accesses at run-time.

Dynamic scheduling enables earliest computation of an operation based on the presence of its input data, which can achieve higher performance than static scheduling. However, resource optimisation is challenging since the states of hardware are unknown at compile time, resulting in lower area efficiency compared to static scheduling.

3.3 Combining Dynamic & Static Scheduling

Combining dynamic and static scheduling can balance the trade-off between performance and area. There are works that extend static scheduling to support dynamic mechanisms for custom features.

Alle *et al.* [2] and Liu *et al.* [35] propose a source-to-source transformation technique to enable run-time selection among multiple schedules based on certain values. Tan *et al.* [43] propose a tool flow named ElasticFlow to optimise pipelining of irregular loop nests that have dynamically-bound inner loops. Dai *et al.* [19, 20] propose pipeline flushing for high throughput of the pipeline and dynamic hazard detection circuitry for speculation in specific applications. These works are still restricted by the conservatism of static scheduling and the hardware performance is still limited for general cases, such as complex memory accesses or control flows.

Extending dynamic scheduling to support SS components is also popular. Carloni [6] describes how to encapsulate static modules into a latency-insensitive system. This approach is realised in an HLS tool named DASS that supports SS circuits inside a DS circuit [12]. DASS still requires manual selection of the code regions as static components. In CIRCT [17], an MLIR dialect named StaticLogic is implemented with a pass as an initial step that automatically extracts the non-elastic operations from the code. However, that MLIR pass only analyses the code at instruction level and cannot recognise whether a loop should be statically scheduled. Also, the hardware synthesis of StaticLogic is not supported. We automate the process of finding these SS components at both instruction level and loop level, and synthesise them into efficient hardware. In this paper, we extend the open-source DASS tool, but our approach can be equally applied to other tool flows such as CIRCT.

3.4 Module Selection & Optimisation in HLS

Module selection is a process to select an optimal module design among a set of choices with the same functionality to improve performance or area. Module selection in HLS has been widely studied. Ishikawa and Micheli [28] propose a module selection algorithm that optimises the schedules of hardware with a finite set of predefined components. Ahmad *et al.* [1] present a problem-space genetic algorithm for static scheduling. Ito *et al.* [29] propose an ILP-based model for optimising the schedule of data flow architecture. Sun *et al.* [42] combine the module selection and resource sharing in design exploration. Cong *et al.* [15] propose an ILP-based scheduling including module selection for streaming applications. However, these approaches all target SS hardware only. The behaviour of DS surroundings can be unpredictable, and these methods cannot be applied without assuming the worst-case computation.

In dynamic scheduling, latency-insensitive system graphs (lisgraphs) are used for hardware optimisation, such as loop pipelining, re-timing and buffering [8, 9, 14, 39]. This is extended to marked graphs in HLS tools like Dynamatic [31]. Cheng *et al.* [13] propose a Petri net-based technique to optimise the initiation interval (II) of each SS component in a DS circuit. In pipelining, an II is defined as the number of clock cycles between the start times of two consecutive iterations. However, none of these works optimise the offsets of component ports. Our work optimises the offsets of static islands to achieve better performance (explained in Section 4.3).

4 METHODOLOGY

This section shows how to determine good static islands for minimal performance loss and maximal resource sharing. Resources in a static island can only be shared inside the static island due to dynamic behaviour in its surroundings. The scheduler can determine the states of a static island once the island starts to compute, but it cannot determine when it starts in relation to other static islands. Therefore, a larger static island contains more resources to share and can achieve higher area efficiency. However, if a large static island contains data-dependent operations, the conservatism in static scheduling may cause performance loss and violate the performance constraints.

In order to automatically determine these optimally sized static islands, we first summarise the fundamental features of code that are amenable for static scheduling. Based on these features, we then show how our tool extracts static islands. Next, we illustrate how to optimise the interface between static islands and their DS surroundings for high performance. Finally, we demonstrate the complete tool flow integrated into DASS.

4.1 Features Amenable for Static Scheduling

In general, static scheduling is optimal if the code behaviour is fully predictable, otherwise the scheduler always assumes the worst case in time. A code region that is amenable for static scheduling should satisfy following conditions:

(1) Code should have no data-dependent control flow, or only have data-dependent control flow where all control flow paths have the same throughput.
(2) If code contains loops, each loop should have no inter-iteration dependence, or only have inter-iteration dependences with constant dependence distances.

For these code regions, the timing of the synthesised hardware is predictable in clock cycles, that is, the SS design can achieve the same throughput as the DS design. The features that are amenable for static scheduling by Vitis HLS are explained later.

4.2 Constructing Static Islands

Constructing static islands is a two-step process, one targeting high-level operations in loops and the other targeting low-level operations in instructions. For each function, if the function contains loops, our analyser first checks whether each loop can be statically scheduled. If it can, our tool identifies the whole loop as a static island. Otherwise, our analyser checks instructions in the function/loop body and constructs static islands in the form of groups of instructions. Each group of instructions forms a subgraph in the data flow graph of the input program as illustrated in Figure 2b.

Step 1: Constructing Static Islands from Loops. Here we introduce four pre-conditions for determining whether a loop should be statically or dynamically scheduled. The first two pre-conditions are restricted by the tools we use, and the other two pre-conditions are restricted by the input code.

Condition 1: Vitis HLS Loop Restrictions. Apart from the features in Section 4.1, Vitis HLS, the tool we use for static scheduling, has additional restrictions for loops to achieve efficient loop pipelining [36, 45]. First, loop nests that cannot be merged into a single loop cannot be optimally pipelined by Vitis HLS. If a loop nest cannot be merged, Vitis HLS either pipelines the top-level loop with all the inner loops fully unrolled, or only pipelines the innermost loops with the rest of the code sequential. Additionally, support

Table 2: Reference table for the scheduling approach that needs to taken for a `for` loop. DS means dynamic scheduling. SS means static scheduling. DS/SS means depending on the performance model.

Iteration latency	Dependence distance		
	No dependence	Constant	Variable
Constant	SS	SS	DS
Variable	SS/DS	SS/DS	DS

for estimating loop trip count from variable loop bounds is limited in Vitis HLS. The scheduler prefers that all the loop bounds and step of a loop to be constant. Finally, all the array indices should be in affine form as the analyser only supports affine analysis. We formalise these features into following constraints:

- L_{single}: the loop is a single loop [1] or the loop is a loop nest that can be merged into a single loop.
- B_{cst}: all the loop bounds and steps are constant.
- A_{affine}: all the array indices are in affine form.

A pre-condition of a loop to be scheduled by Vitis HLS is:

$$C_{\text{VHLS}} = L_{\text{single}} \land B_{\text{cst}} \land A_{\text{affine}}$$

Condition 2: DASS Shared Memory Restrictions. DASS, the tool we use as an HLS tool, also has restrictions on shared memory architecture between static islands and DS hardware. Static islands are treated as black boxes by the DS hardware, and memory dependence between SS and DS hardware cannot be handled by DASS [12]. That is, a loop should not have memory dependence with external code. In the data flow graph, the memory dependence is controlled by LSQs. Here we restrict that loops that share an LSQ with other loops cannot be statically scheduled. We then define the following:

- $L = \{l_0, l_1, ...\}$: the set of all the loops in the program.
- Q: the set of all the LSQs in the data flow graph of a program if the program is dynamically scheduled.
- Q_l: the set of all the LSQs connected to the data flow graph of loop l if the loop is dynamically scheduled. $Q_l \subseteq Q$

The restriction by DASS for a loop l is then:

$$C_{\text{DASS}} = (\forall l' \in L \setminus \{l\}.\, Q_l \cap Q_{l'} = \emptyset)$$

Condition 3: Loop with No Branches. Now we discuss the conditions restricted by the code. A minimum II for an loop iteration is the number of cycles that the iteration must wait after its last iteration starts. Dynamic scheduling supports different minimum IIs across loop iterations, while static scheduling allows a constant II value for all the iterations. If the minimum II is the same across all the loop iteration, the loop should be statically scheduled. Otherwise, all the IIs are relaxed to the maximum value among all the minimum IIs, which causes reduced throughput.

An II depends on two constraints, iteration latency of the loop and inter-iteration dependence [49]. Assuming that all operators take constant time, the iteration latency is a constant if the loop body does not have branches.[2] The inter-iteration dependence is

[1] A innermost loop of a loop nest can be considered as a single loop.
[2] This assumes that all the functions called in the loop are inlined.

```
float s = 0;
float d1, d2;
for (int i=0; i<N;
  i++) {
  float d = A[i];
  // P(true)=0.5
  if (d < 1)
    // II=9
    s = s*d + d2*(
      d+0.1);
  else
    // II=5
    s = s+d1;
}
```

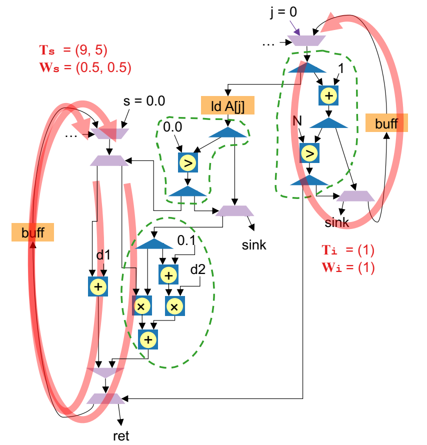


Figure 3: A loop example that conditionally updates a variable `s` based on the value of array element `A[i]`. A data flow graph of the loop is shown at the bottom of the figure. The cycles indicate dependences. The carried dependence on `s` has two possible latencies, illustrating two possible IIs. The extracted static islands are shown as dotted circles based on the given $\mathbf{T}_s$ and $\mathbf{W}_s$ in the figure.

amenable when a loop does not have inter-iteration dependence or its dependence distance is constant. We then define following constraints:

- D_{inter}: the loop has inter-iteration dependence.
- D_{cst}: the distances of all the inter-iteration dependences in the loop are constant.
- B_{br}: the loop body has branches.

Then the loops that satisfy the following condition should be statically scheduled, where iteration latency and dependence distance are both constant, resulting in a constant II:

$$C_0 = \neg B_{\text{br}} \land (D_{\text{cst}} \lor \neg D_{\text{inter}})$$

For instance, the loop below has no branch but inter-iteration dependence on array `A`. Assume `f(i)` is a simple expression of `i`. When `f(i) = i*i`, the dependence distance is a variable, so the loop cannot be statically scheduled. When `f(i) = i+10`, the dependence distance is 10, so the loop can be statically scheduled.

```
for (int i=0; i<N; i++)
  A[f(i)] = A[i];
```

Condition 4: Loop with Branches. A loop that contains branches may have a set of iteration latencies and could lead to a set of IIs. In static scheduling, the maximum value of II among these IIs is used for all the loop iterations. Such approximation could cause throughput loss compared to dynamically scheduling the loop. In order to evaluate the throughput loss caused by the II approximation, we propose a performance model specified by following constraints:

- $V_C = \{v_0, v_1, ...\}$: the set of all the variables in the loop that have carried dependences.
- $\mathbf{P}_v$: the vector of all the cycles for a variable that has carried dependences. $v \in V_C$.
- $\mathbf{T}_v$: the vector of the latencies of these cycles in clock cycles.

- $\mathbf{W}_v$: the vector of the probabilities of computing these cycles. $\mathbf{W}_v \cdot \mathbf{1} = 1$.
- λ_0: affordable loss factor.

In order to perform throughput analysis, a data flow graph is generated from the source of the loop. In the graph, each carried dependence on a variable is represented as a set of cycles. For instance, the left side of Figure 3 shows a loop example that contains a branch. The loop checks whether the value of an array element `A[i]` is less than 1. If it is, the variable `s` is updated by `s*d+d2*(d+0.1)`, otherwise `s` accumulates `d1`.

The data flow graph of the loop example is on the right of Figure 3. There are two variables that have carried dependences, $V_C = \{\texttt{s}, \texttt{i}\}$. In the figure, the cycle on on the top right represents the carried dependence on `i`. The value of `i` always increments by 1 in each iteration, where $|\mathbf{P}_\texttt{i}| = 1$. The cycles on the left represent the carried dependences on `s`. The value of `s` depends on the `if` condition, so there are two cycles in graph representing the results from true and false branches respectively, where $|\mathbf{P}_\texttt{s}| = 2$.

Each cycle has a latency and a probability, formalised as elements in $\mathbf{T_v}$ and $\mathbf{W_v}$ respectively. The latency indicates the minimum time in clock cycles required to update the variable with carried dependence, also known as the iteration latency of the cycle. $\mathbf{W_v}$ are obtained by profiling. In Figure 3, the cycle for `i` has a fixed latency of 1. Therefore, $\mathbf{T}_\texttt{i} = (1)$ and $\mathbf{W}_\texttt{i} = (1)$. Assuming that the probability of `if` condition being true is 0.5, $\mathbf{W}_\texttt{s} = (0.5, 0.5)$, and assuming that the latency of a floating point adder is always 5 cycles and the latency of a floating point multiplier is always 4 cycles, the cycle for the true branch has a latency of 9, and the cycle for the false branch has a latency of 5, that is, $\mathbf{T}_\texttt{s} = (9, 5)$.

The performance of the loop can be estimated using the latencies and probabilities of cycles. Dynamic scheduling supports a variable iteration latency, and the average iteration latency for a variable v is represented as $T_{v,\text{dynamic}}$. Static scheduling supports a constant iteration latency, and the iteration latency for a variable v is then the maximum latency among all the cycles, represented as $T_{v,\text{static}}$.

$$T_{v,\text{dynamic}} = \mathbf{T}_v \cdot \mathbf{W}_v$$
$$T_{v,\text{static}} = \max \mathbf{T}_v$$

We simplify our analysis by restricting that the loop only has no inter-iteration dependence or inter-iteration dependences with constant dependence distance. A loop that has dynamic dependence distances should be dynamically scheduled. The average IIs are:

$$T_{\text{dynamic}} = \max_{v \in V_C} T_{v,\text{dynamic}} \qquad II_{\text{dynamic}} = \frac{T_{\text{dynamic}}}{\text{d}}$$
$$T_{\text{static}} = \max_{v \in V_C} T_{v,\text{static}} \qquad II_{\text{static}} = \frac{T_{\text{static}}}{\text{d}}$$

The satisfied loop then has a constant minimum dependence distance, d. The final II of both SS and DS loops is restricted by the maximum II among all the variables that have carried dependences. In Figure 3, the II is restricted by the variable `s`. The performance loss caused by switching a DS loop to a SS loop can be estimated using a *loss factor* λ.

$$\lambda = \frac{II_{\text{static}} - II_{\text{dynamic}}}{II_{\text{dynamic}}} = \frac{T_{\text{static}} - T_{\text{dynamic}}}{T_{\text{dynamic}}}$$

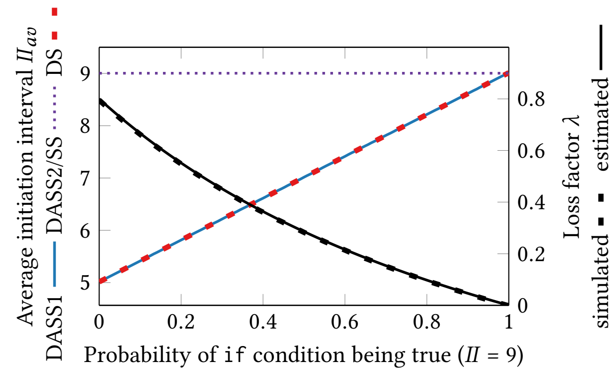


Figure 4: DS = the design without any static island. DASS1 = the design statically scheduling line 9 of Figure 3. DASS2/SS = the design statically scheduling the whole loop. The average IIs of DASS1 design and DS design vary from 5 to 9 because of the dynamically scheduled `if` condition. The II of DASS2/SS design has a constant II of 9. The loss factor of the loop in Figure 3 increases with the probability of `if` condition being true. The maximum error between the estimated loss factor and the actual loss factor by simulation is 0.5%.

The constant d is then cancelled. If the throughput loss caused by approximating the II is affordable for a user-defined threshold λ_0, the loop is then statically scheduled.

$$C_\lambda = (\lambda_0 - \lambda \geq 0)$$

A curve of the loss factor for the loop in Figure 3 over the distribution of `if` condition being true is shown in Figure 4. Our estimated loss factor is close to the one by simulation. For the example in Figure 3, $\lambda = 0.28$. Assuming $\lambda_0 = 0.05$, our tool suggests to dynamically schedule the loop. The loop still has SS parts in the loop body, resulting in the DASS1 design by step 2, which are explained later in this paper. Based on the performance model, loops that satisfy the following condition should be statically scheduled:

$$C_1 = B_{\text{br}} \wedge (D_{\text{cst}} \vee \neg D_{\text{inter}}) \wedge C_\lambda$$

Summarised Condition. The summarised condition that indicates whether a loop should be statically scheduled is shown in Equation 1. Table 2 summarises Equation 1 based on the source patterns. Each loop to be statically scheduled is considered as a single static island. Any two adjacent SS loops are further merged into a static island in step 2.

$$C_{\text{static loop}} = C_{\text{VHLS}} \wedge C_{\text{DASS}} \wedge (C_0 \vee C_1) \tag{1}$$

Step 2: Constructing Static Islands from Instructions. If a loop is not amenable for static scheduling, our tool extracts static islands from the loop body at instruction level. We consider each extracted SS loop or each DS operation as a single node in the data flow graph of the function. We define following terms:

- N: the set of all the nodes in the data flow graph.
- $E \longrightarrow N \times N$: the set of all the edges in the graph.
- $M \longrightarrow \{0, 1\}$: whether these nodes can be merged.
- $S \longrightarrow \mathbb{N}$: the ID of static island to which a node belongs.

We specify the merging rule as follows:

(1) Each node is considered as a static island.
(2) Node that does not have the same throughput at all its input and output ports cannot be merged, *e.g. Merge*, *Branch* and *Mux*. This avoids performance loss in pipelining data-dependent operations.
(3) Nodes that connects to a LSQ cannot be merged for the same reason explained in Condition 2.

Based on the rules above, our tool iteratively merges nodes into larger static islands. For instance, the dotted circles in Figure 3 represent the merged static islands in the graph. Each island can be synthesised into a single component. The following condition holds after extracting static islands.

$$\forall(n_0, n_1) \in E, S(n_0) = S(n_1) \vee \neg(M(n_0) \wedge M(n_1))$$

The main benefit of static scheduling is enabling resource sharing in a static island. As an optimisation process, our tool evaluates the size of each island by counting the number of operations. If an island is too small that has no space for resource sharing, then it remains dynamically scheduled. For the example in Figure 3, only the island at the bottom left is amenable for static scheduling, while the other two islands are ignored. Table 3 shows the number of static islands found over a set of benchmarks.

Table 3: Static islands found over a set of benchmarks.

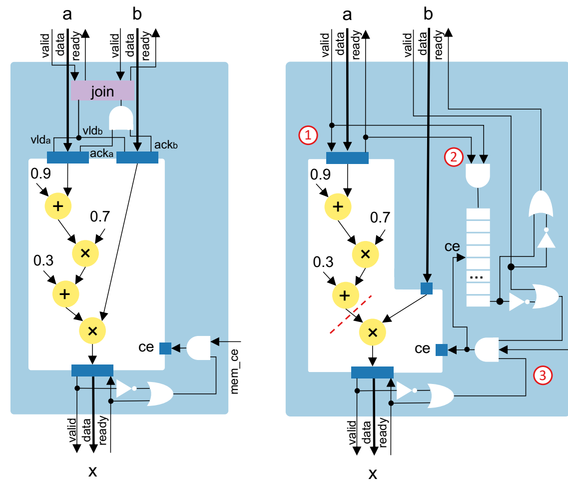
Benchmarks	# Islands	Benchmarks	# Islands
vecNormTrans	2	getTanh	1
doitgenTriple	2	covariance	3
correlation	5	syr2k	1
levmarq	2	gemver	3
gramSchmidt	7	gesummv	2

4.3 Optimising Static Islands using Offsets

Once a static islands is determined, it is synthesised and placed in a wrapper to communicate with its DS surroundings. The wrapper monitors and controls the computation of the static island based on its input and output states.

An offset of an input means the number of cycles between the start time of the computation and the time when this input is firstly used. An offset of an output means the number of cycles between the end time of the computation and the time when this output starts to be valid. This section introduces how to use the offsets to optimise the wrapper to improve the overall throughput. We first compare the throughput of the original wrapper in DASS and our proposed wrapper. Then we show how the proposed wrapper is implemented and its additional constraints.

4.3.1 Throughput of Original DASS Wrapper Design. The offset can be used to determine when an input/output is required, which can affect the overall performance. The original wrapper assumes zero offsets for all the inputs, which requires all the inputs to be valid to start the computation [12]. When a static island is in a loop and has carried dependence, the latency of the static island affects the overall throughput. For a static island, let M be the set of its inputs



(a) Original wrapper **(b) Proposed wrapper**

Figure 5: An example of a wrapper with offset constraints. a is consumed immediately when the component starts to compute (offset = 0), and b is only required 13 cycles after a is consumed (offset = 13). The original wrapper uses a *join* to synchronise all the inputs. A shift register is implemented in (b) for b to count the offset and control its handshake interface. The red dashed line illustrates the state where b is required by the component. The implementation of the proposed wrapper contains three parts: 1) Interface for ports with zero offsets, the same as (a); 2) Interface for ports with positive offsets; 3) Interface for backpressure. ce represents the clock enable signal.

and N be the set of its outputs. The carried dependence set among these inputs and outputs over all the iterations is D, where:

$$D \subseteq M \times N \times \mathbb{N} \times \mathbb{N} : (m, n, k_1, k_2) \in D$$

k_1 and k_2 are the iteration indices, where $k_1 > k_2$. For instance, (m, n, k_1, k_2) means the input m in iteration k_1 depends on the output n in iteration k_2.

Now we formalise the time constraints, and all the times are in clock cycles. Let $t_{m,k}$ be the time when input m in iteration k is *consumed* and $t_{n,k}$ be the time when output n in iteration k *becomes* valid [3]. The dependence constraint can be formulated as follows:

$$\forall(m, n, k_1, k_2) \in D, t_{m,k_1} \geq t_{n,k_2} \tag{2}$$

The DS surroundings use handshake interface to ensure that Equation 2 *always* holds. Here we use Equation 2 as a pre-condition for the following analysis in the static island.

Assume that a static island is pipelined with a constant II and always have the same latency. Let α_m be the offset of input m and L_n be the latency of output n. In an original wrapper design, all the inputs in the same iteration are synchronised:

$$\forall m \in M, \alpha_m = 0 \tag{3}$$

$$\forall m, m' \in M, k > 0, t_{m,k} = t_{m',k} \tag{4}$$

[3] The output value may become valid before the whole computation finishes, where the output offset is the time difference.

The time constraint for the output n is then:

$$t_{n,k} \geq t_{m,k} + L_n \tag{5}$$

Substituting Equation 5 into Equation 2:

$$t_{m,k_1} - t_{m,k_2} \geq L_n \tag{6}$$

The upper bound of II at m only depends on the latency of the static island and dependence distance in iterations.

4.3.2 Throughput of Our Proposed Wrapper Design. We propose a new wrapper that does not require all the input to be valid before the computation. The component can start earlier with some missing inputs as long as these inputs are not required in the next clock cycle, that is, these inputs have non-zero offsets. If an input is required but not valid, the wrapper stalls the whole component until the input becomes valid.

For the new wrapper with positive offsets, Equation 4 no longer holds. The output constraint in Equation 5 now becomes:

$$t_{n,k} \geq t_{m,k} + L_n - \alpha_m \tag{7}$$

Substituting it into Equation 2:

$$t_{m,k_1} - t_{m,k_2} \geq L_n - \alpha_m$$

The upper bound of II at m now also depends on its offset and is lower than Equation 6 when the offset is non-zero. Our proposed wrapper can avoid significant throughput loss when the offset and latency are both large.

4.3.3 Implementation of Proposed Wrapper. An example of static island with two inputs and one output is illustrated in Figure 5. The component takes two inputs `a` and `b` and returns a result `x` as `(((0.9+a)*0.7)+0.3)*b`. When the static island starts to compute, only `a` is required to compute `(((0.9+a)*0.7)+0.3)` before multiplying with `b`. Assuming each adder has a latency of 4 cycles and each multiplier has a latency of 5 cycles, `a` has an offset of 0 and `b` has an offset of 13.

Figure 5a is the original wrapper and Figure 5b is our proposed wrapper. Our proposed wrapper is implemented mainly in three parts. First, the interface of an input with zero offset, such as `a`, is implemented by the the same handshake interface as the original wrapper. It checks the valid signal in a constant time interval specified by the II. The component takes *bubble* if the valid is not valid yet, where the data inside the component is *still* being processed [12]. Second, the interface of of an input with positive offset, such as `b`, is controlled by an additional shift register. The shift register is synchronised with the pipeline state of the component. The time when `b` is required is indicated by a certain bit of the shift register being set, *i.e.* the 13th bit for this example. When the bit is set but `b` is not valid, the whole component is stalled waiting for `b`, where the data inside the component is all stalled.

Finally, the clock enable signal of the component is used to stall the component. Similar to the original wrapper, it is controlled by the back pressure and memory arbiter. The major difference from the original wrapper is that the absence of `b` can also stall the component.

4.3.4 II Constraints of Proposed Wrapper. The component now can be stalled by both the inputs and the outputs (back pressure). An inappropriate II for this wrapper could cause deadlock. Here we show how to formalise the deadlock problem. Since the inputs may not be synchronised, the condition that always holds for the new wrapper design is:

$$\forall m, m' \in M, \alpha_m \geq \alpha_{m'}, t_{m',k} - \alpha_{m'} \leq t_{m,k} - \alpha_m \tag{8}$$

This means that an input with a larger offset α_m is always consumed later than another input with a smaller offset $\alpha_{m'}$ by at least $\alpha_m - \alpha_{m'}$ cycles. Then deadlock happens when an executing output is being required by an input:

$$\exists (m, n, k_1, k_2) \in D, m' \in M, t_{m',k_1} + \alpha_m - \alpha_{m'} < t_{n,k_2} \tag{9}$$

For instance, when the input m' in iteration k_1 has propagated to the point where the input m is required. The output n in the iteration k_2 is still in the component, and the input m is not valid because of Equation 2. The component is forever stalled, waiting for the output being stalled.

This never happens in the original wrapper because Equation 9 never holds under Equation 3. In order to avoid the deadlock in the new wrapper, an additional constraint is required to avoid Equation 9:

$$\forall (m, n, k_1, k_2) \in D, m' \in M, t_{m',k_1} + \alpha_m - \alpha_{m'} \geq t_{n,k_2} \tag{10}$$

This always holds when $\alpha_m \leq \alpha_{m'}$. For $\alpha_m > \alpha_{m'}$, substituting Equation 7 into the equation above:

$$\begin{aligned} t_{m',k_1} + \alpha_m - \alpha_{m'} &\geq t_{m',k_2} + L_n - \alpha_{m'} \\ t_{m',k_1} &\geq t_{m',k_2} + L_n - \alpha_m \end{aligned}$$

Assume that a static island is synthesised and pipelined with a constant II of P. Since the input m' is not synchronised with the input m, the actual II of m' is still restricted by P:

$$t_{m',k_1} \geq t_{m',k_2} + P(k_1 - k_2)$$

Then constraint then becomes:

$$P(k_1 - k_2) \geq +L_n - \alpha_m$$

The following must hold for a wrapper design without deadlock:

$$\forall (m, n, k_1, k_2) \in D, \{m' \in M | \alpha_m > \alpha_{m'}\} = \emptyset \vee P \geq \frac{L_n - \alpha_m}{k_1 - k_2} \tag{11}$$

In summary, the original wrapper assumes no offsets and may have suboptimal performance. We propose a new wrapper design that has a improved throughput upper bound with II constraints. Our tool checks if the condition holds. If it does not hold and D is data-dependent, the static island is split into multiple static islands with zero offsets. For instance, the static island in Figure 5b can be split into two island annotated by the red dotted line. If it does not hold and D is known, the tool relaxes the II of the component until Equation 11 holds to avoid deadlock. Even P is restricted, the lower bound of P is still no greater than the optimal II, so the optimal throughput is still reachable. Table 4 evaluates the performance improvement by replacing original wrappers with our wrappers. The area change of a wrapper is negligible compared to the total area of the designs.

Table 4: Comparison of total cycles between the designs using the original wrapper and our new wrapper. When there is a dependence between the input and output of a static island, the throughput of hardware using original wrapper is significantly worse, otherwise only the latency is affected.

Benchmarks	Original wrapper	Our wrapper
vecNormTrans	17439	6703
doitgenTriple	329016	263435
levmarq	54315	54296

4.4 Tool Flow

DASS internally relies on Vitis HLS and Dynamatic: 1) Vitis HLS generates the static components, 2) Dynamatic generates the dynamically scheduled circuits, and 3) DASS itself generates the wrappers around static components so that they can communicate with their dynamically scheduled surroundings. DASS then puts the three design files together as the final design in RTL. Our work is a new extension in the DASS flow.

Figure 1 illustrates the complete tool flow with our work integrated. The input code is analysed by our static island analyser in two steps to determine static islands in loops and instructions. These static islands are then synthesised by Vitis HLS through its LLVM front end [47]. The offset constraints are extracted from the scheduling reports of static islands. The DS code region is transformed into a dot graph buffered with offset constraints, and then translated into a hardware netlist of DS components. Finally, the wrappers with offset optimisation are generated as part of design.

5 EXPERIMENTS

We evaluate our work on a set of realistic benchmarks, comparing with the corresponding SS and DS designs in total circuit area and wall clock time. The IIs of static islands are automatically chosen by the II analyser in DASS [13]. We obtain the total clock cycles from Vivado XSIM simulator and the area results from Post Synthesis report in Vivado. The FPGA family we used for result measurements is xc7z020clg484 and the version of Xilinx software is 2020.2.

5.1 Benchmarks

The benchmarks are chosen based on whether they lack the features mentioned in Section 4.1, that is, fully static scheduling is suboptimal. Finding suitable benchmarks is a perennial problem for papers that push the limits of HLS, in part because existing benchmark sets such as Polybench [37] and CHStone [23] tend to be tailored to what HLS tools can already comfortably handle. Therefore, we list ten realistic benchmarks, where most of them are modified from Polybench. [4] In the first six benchmarks, regular computations are translated into sparse computations to improve the performance. The last four benchmarks have complex loop nests.

vecNormTrans is the motivating example in Figure 2a.
doitgenTriple is a weighted version of Multi-resolution analysis kernel (MADNESS).
correlation computes the correlation matrix.

[4] Open-sourced at [24].

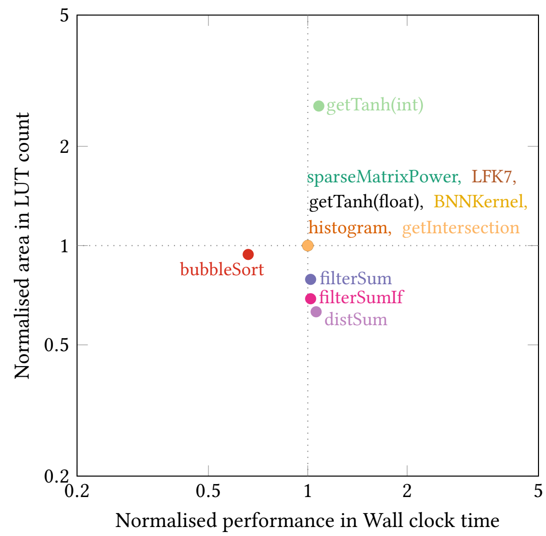


Figure 6: Comparison with the manual DASS designs in [12]. The manual DASS designs are at (1, 1) and the points represent the designs by our approach. For most benchmarks, our tool achieves the same hardware as the hardware manually designed by experts. For certain benchmarks, our tool even finds better static islands than those in the manual designs, resulting in smaller area or better performance. The small difference in performance is mainly caused by different maximum frequencies of the designs. We expect higher performance in `getTanh(int)` but the result is restricted by the pipelining capabilities of Dynamatic for complex loops.

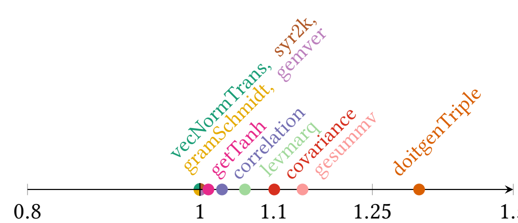


Figure 7: Comparison with the designs with the best performance. The best performant design is normalised at 1, and the points represent the relative performance of our designs for evaluated benchmarks.

levmarq is an implementation of the Levenberg-Marquardt algorithm to solve least-squares problems [34].
gramSchmidt is an optimised version Gram-Schmidt decomposition, which conditionally computes matrix based on the 2-norm of the rows in a matrix [22].
getTanh transforms a vector using a *tanh* function, where the *tanh* function has different latencies for inputs in different regions.
covariance computes the covariance matrix.
syr2k is a symmetric rank-2k update for two matrices.
gemver is vector multiplication and matrix addition.
gesummv is scalar, vector and matrix multiplication.

Table 5: Evaluation of our approach on a set of benchmarks. The first part of table evaluates the benchmarks that are amenable to our approach; and the second part of the table evaluates the benchmarks that do not have data-dependent operations. SS = the designs directly synthesised by Vitis HLS. DS = the designs directly synthesised by Dynamatic. DASS = the designs synthesised by DASS using our proposed approach. ADP = area-delay product.

Benchmarks	IIs	LUTs			DSPs			Cycles			Fmax (MHz)			Wall clock time (s)			Norm. ADP		
		SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS
vecNormTrans	9, 2	1.61k	21.3k	2.86k	5	20	7	12.3k	6.08k	6.70k	121	89.7	106	102μ	63.1μ	67.8μ	1	2.7	0.87
doitgenTriple	5, 5	768	21.8k	20.4k	10	20	10	399k	198k	263k	121	87.6	87.6	3.3m	2.26m	3.01m	1	1.4	0.91
correlation	4, 4, 10, 1, 4	2.13k	14.3k	6.72k	5	36	22	79.9k	66.6k	70.0k	121	80.5	103	661μ	827μ	680μ	1	9	4.5
levmarq	59, 6	2.42k	22.4k	6.74k	15	153	22	204k	51.2k	54.3k	121	120	120	1.69m	427μ	452μ	1	2.6	0.39
gramSchmidt	1, 34, 1, 1, 1, 15, 2	3.27k	15.1k	15.9k	5	49	31	375k	99.0k	61.0k	121	117	117	3.11m	844μ	519μ	1	2.7	1
getTanh	4	892	3.79k	1.32k	16	16	5	1.03k	1.04k	1.03k	121	120	120	8.5μ	8.7μ	8.6μ	1	1	0.31
Norm. median		**1×**	**7.99×**	**2.97×**	**1×**	**5.6×**	**1.43×**	**1×**	**0.50×**	**0.60×**	**1×**	**0.86×**	**0.93×**	**1×**	**0.68×**	**0.76×**	**1×**	**2.6×**	**0.89×**
covariance	4, 6, 4	2.33k	7.91k	4.24k	5	9	9	92.3k	72.9k	84.0k	121	86.1	99.8	764μ	847μ	841μ	1	2	2
syr2k	4	829	4.04k	3.52k	5	19	8	81.4k	66.5k	78.7k	121	98.1	118	673μ	678μ	665μ	1	3.8	1.6
gemver	6, 4, 4	1.44k	8.32k	3.24k	10	28	17	599k	611k	532k	115	106	106	5.23m	5.76m	5.01m	1	3.1	1.6
gesummv	1, 10	2.11k	3.37k	2.75k	8	18	14	71.2k	262k	68.8k	121	113	102	589μ	2.31m	673μ	1	8.8	2
Norm. median		**1×**	**4.14×**	**2.03×**	**1×**	**2.53×**	**1.73×**	**1×**	**0.92×**	**0.94×**	**1×**	**0.87×**	**0.89×**	**1×**	**1.10×**	**1.04×**	**1×**	**3.5×**	**1.8×**

5.2 Results

Compared with the manual designs by experts over the benchmarks in the original DASS paper [12], most of the designs generated by our tool have comparable performance and area as shown in Figure 6. Compared with the fastest designs discovered via exhaustive search over the selected ten benchmarks, our approach generates designs that are within 2% of the performance as shown in Fig. 7. Details are shown in Table 5. The first part of the table shows the detailed results of the six benchmarks where dynamic scheduling should significantly improve the throughput. First, as shown in Section 2, `vecNormTrans` has high throughput because the input dependent computation remains in DS part, and our tool enables resource sharing and LSQ removing. `doitgenTriple` has two loops in sequence that have memory dependence scheduled by a LSQ. Since static islands cannot share LSQs, these loops remain dynamically scheduled. The loop bodies of two loops can still be synthesised as static islands, resulting in significantly reduced DSPs. A big difference between the DASS hardware and DS hardware is because the difference of the latencies of floating-point adders in two tools, which affects the throughput. This also occurs in `levmarq`.

`correlation` has complex loop nests and conditional computation on the standard deviation to avoid zero-divide. Dynamic scheduling pipelines the top-level loop, while Vitis HLS only pipelines innermost loops. This is more significant for large benchmarks `levmarq` and `gramSchmidt`. The DASS hardware and DS hardware avoid that and achieve higher throughput. There is a significant performance improvement in `gramSchmidt` when switch DS to DASS, because Vitis HLS uses advanced `facc` ops for floating-point accumulation which has a latency of 1 cycle, while Dynamatic uses normal `fadd` ops which has a latency of 4 cycles.

`getTanh` is a special case, where DASS hardware has close performance as SS hardware but smaller area. In static scheduling, the scheduler assumes all the elements in the input vector are in the linear region, resulting in an II of 1 for the high-precision computation. In DASS, the code region that performs high-precision computation is synthesised as a static island. Knowing the probability of the input that is in the saturation region by profiling, the II of the static island can be relaxed. Overall, the average area-delay product of our designs is better than both SS and DS designs.

The second part of Table 5 shows area advantages over naive dynamic scheduling which is separated from those benchmarks where dynamic scheduling provides an advantage. They are more classic benchmarks where static scheduling would be an obvious approach. Overall, the latencies of designs by three approaches is close but the areas are significantly different. The reduced cycles from DS to DASS is caused by the use of `facc` ops. Our approach is able to reduce the overhead of using dynamic scheduling. However, DASS cannot share resource between two static islands as stated in Section 4. This results in larger area-delay product compared to SS designs.

6 CONCLUSIONS

Existing HLS tools require user to manually specify their scheduling constraints to achieve high performance and area efficient hardware designs. In this work, we present a rule-based technique to automatically select the scheduling strategies for the input programs. Our tool also optimises the interface between code regions using different scheduling strategies to achieve high performance.

We show how to use our approach to automatically determine the static islands in a DS circuit. Across a range of benchmarks that are amenable to our approach, our approach on average achieves a 3.8-fold reduction in area combined with a 13% performance boost through automatic identification and synthesis of static islands from fully dynamically scheduled circuits. The performance of the resulting hardware is close to optimum. Our future work will explore the fundamental limits of this approach, both theoretically and practically.

ACKNOWLEDGEMENTS

This work is supported by the EPSRC (EP/P010040/1, EP/R006865/1).

REFERENCES

[1] I. Ahmad, M. K. Dhodhi, and C. Y. R. Chen. 1995. Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis. *IEE Proceedings - Computers and Digital Techniques* 142, 1 (1995), 65–71. https://doi.org/10.1049/ip-cdt:19951516

[2] 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.

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

[4] A. Canis, S. D. Brown, and J. H. Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2014.6927490

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 24 (Sept. 2013), 27 pages.

[6] 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

[7] 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

[8] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. 2000. Performance Analysis and Optimization of Latency Insensitive Systems. In *Proceedings of the 37th Annual Design Automation Conference* (Los Angeles, California, USA) *(DAC '00)*. Association for Computing Machinery, New York, NY, USA, 361–367. https://doi.org/10.1145/337292.337441

[9] Mario R. Casu and Luca Macchiarulo. 2004. A New Approach to Latency Insensitive Design. In *Proceedings of the 41st Annual Design Automation Conference* (San Diego, CA, USA) *(DAC '04)*. Association for Computing Machinery, New York, NY, USA, 576–581. https://doi.org/10.1145/996566.996725

[10] Catapult High-Level Synthesis. 2020. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

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

[12] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson. 2021. DASS: Combining Dynamic and Static Scheduling in High-level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), 1–1. https://doi.org/10.1109/TCAD.2021.3065902

[13] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2021. Probabilistic Scheduling in High-Level Synthesis. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 195–203. https://doi.org/10.1109/FCCM51124.2021.00031

[14] R. L. Collins and L. P. Carloni. 2008. Topology-Based Performance Analysis and Optimization of Latency-Insensitive Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 12 (2008), 2277–2290.

[15] Jason Cong, Muhuan Huang, Bin Liu, Peng Zhang, and Yi Zou. 2012. Combining module selection and replication for throughput-driven streaming programs. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1018–1023. https://doi.org/10.1109/DATE.2012.6176645

[16] 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.

[17] CIRCT contributors. 2021. CIRCT: Circuit IR Compilers and Tools. https://github.com/llvm/circt/tree/main/.

[18] 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.

[19] 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.

[20] 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.

[21] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. 2020. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 244–254. https://doi.org/10.1145/3373087.3375296

[22] gram-schmidt. 2021. https://github.com/chrundle/gram-schmidt

[23] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*. 1192–1195. https://doi.org/10.1109/ISCAS.2008.4541637

[24] HLS Benchmarks. 2021. https://github.com/JianyiCheng/HLS-benchmarks/tree/master/StaticIslands

[25] 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*.

[26] 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.

[27] Intel HLS Compiler. 2021. https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/hls-compiler.html

[28] M. Ishikawa and G. De Micheli. 1991. A module selection algorithm for high-level synthesis. In *1991., IEEE International Sympoisum on Circuits and Systems*. 1777–1780 vol.3. https://doi.org/10.1109/ISCAS.1991.176748

[29] K. Ito, L. E. Lucke, and K. K. Parhi. 1998. ILP-based cost-optimal DSP synthesis with module selection and data format conversion. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6, 4 (1998), 582–594. https://doi.org/10.1109/92.736132

[30] 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.

[31] 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.

[32] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne. 2019. Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 197–205.

[33] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 65–76. https://doi.org/10.1109/ISPASS.2009.4919639

[34] levenberg-maquardt-example. 2021. https://github.com/leechwort/levenberg-maquardt-example

[35] 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.

[36] Optimization Techniques in Vitis HLS. 2021. https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/vitis_hls_optimization_techniques.html

[37] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012).

[38] Zhiqiang Que, Erwei Wang, Umar Marikar, Eric Moreno, Jennifer Ngadiuba, Hamza Javed, Bartłomiej Borzyszkowski, Thea Aarrestad, Vladimir Loncar, Sioni Summers, Maurizio Pierini, Peter Y Cheung, and Wayne Luk. 2021. Accelerating Recurrent Neural Networks for Gravitational Wave Experiments. In *32th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE.

[39] M. Singh and M. Theobald. 2004. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 2. 1008–1013 Vol.2.

[40] Stratus High-Level Synthesis. 2021. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[41] Qiuyue Sun, Amir Taherin, Yawo Siatitse, and Yuhao Zhu. 2020. Energy-Efficient 360-Degree Video Rendering on FPGA via Algorithm-Architecture Co-Design. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 97–103. https://doi.org/10.1145/3373087.3375317

[42] W. Sun, M. J. Wirthlin, and S. Neuendorffer. 2007. FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2007), 254–265. https://doi.org/10.1109/TCAD.2006.887923

[43] 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.

[44] 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.

[45] Vitis HLS Coding Styles. 2021. https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_coding_styles.html

[46] 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)*. 26–34. https://doi.org/10.1109/FCCM.2019.00014

[47] Xilinx Vitis HLS. 2021. https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/index.html
[48] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. 2020. Using OpenCL to Enable Software-like Development of an FPGA-Accelerated Biophotonic Cancer Treatment Simulator. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 86–96. https://doi.org/10.1145/3373087.3375300
[49] Z. Zhang and B. Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 211–218. https://doi.org/10.1109/ICCAD.2013.6691121
[50] Ruizhe Zhao, Ho-Cheung Ng, Wayne Luk, and Xinyu Niu. 2018. Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 147–1477. https://doi.org/10.1109/FPL.2018.00033