Dynamic Inter-Block Scheduling for HLS

Jianyi Cheng,[1] Lana Josipović,[2] George A. Constantinides[1] and John Wickerson[1]
[1]Imperial College London, UK [2]ETH Zurich, Switzerland
Email: {jianyi.cheng17, g.constantinides, j.wickerson}@imperial.ac.uk, ljosipovic@ethz.ch

***Abstract*—A recent theme in HLS research is the production of *dynamically scheduled* circuits, which are made up of components that use handshaking to schedule themselves at run time, as opposed to following a schedule determined statically at compile time. Dynamically scheduled circuits promise superior performance on 'irregular' source programs, such as those whose control flow depends on input data, at the cost of additional area.**

Current dynamic scheduling techniques are well able to exploit parallelism among instructions *within* each basic block (BB) of the source program, but parallelism *between* BBs is underexplored. Although current tools allow the operations of different BBs to overlap, they require the BBs to start in strict program order, thus limiting the achievable parallelism and overall performance.

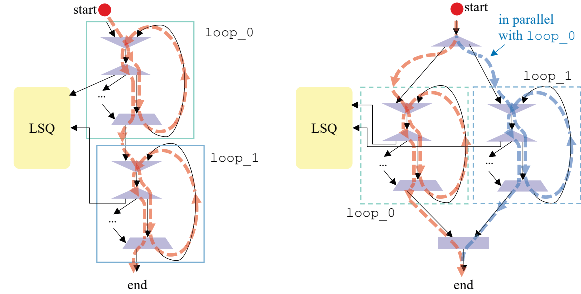
We seek to lift this restriction. Doing so involves developing a toolflow that tackles the following challenges: (1) finding consecutive subgraphs in the control-flow graph and using static analysis to identify those subgraphs that can be safely parallelised, and (2) adapting the circuit so that those subgraphs are executed in parallel while ensuring deterministic circuit behaviour and correct usage of memory interfaces.

Using two benchmark sets from related works, we compare our proposed toolflow against a state-of-the-art dynamically scheduled HLS tool called Dynamatic. Our results show that after standard loop unrolling is applied, our toolflow yields a 4× average speedup, with a negligible area overhead. This increases to a 7.3× average speedup when our toolflow is further combined with C-slow pipelining.

I. Introduction

A central step of the HLS process is scheduling, which maps each operation in the input program to a clock cycle. This mapping can be decided either at compile time (statically) or at run time (dynamically). There has been recent interest in dynamic scheduling because it enables the hardware to adapt its behaviour at run time to particular input values, memory access patterns, and control-flow decisions. Therefore, it potentially achieves better performance compared to the static schedule produced by conservative analysis at compile time.

Dynamically scheduled HLS tools, such as Dynamatic [1], transform a sequential program into a circuit made up of components that are connected by handshaking signals. Each component can start as soon as all of its inputs are ready. Although these tools aim to allow out-of-order execution as much as possible, they must take care to respect dependences in the source program. There are two kinds of dependences: memory dependences (*i.e.* dependence via a memory location) and data dependences (*i.e.* dependence via a program variable). There are also two scopes of dependence: between instructions in the same BB, and between instructions in different BBs. This leads to four cases to consider:



(a) Default CFG. (b) Parallelised CFG.

Fig. 1: Transformation of control flow graph (CFG) for the example in Fig. 2. Our toolflow enables multiple BBs to start simultaneously to achieve the schedule in Fig. 2c.

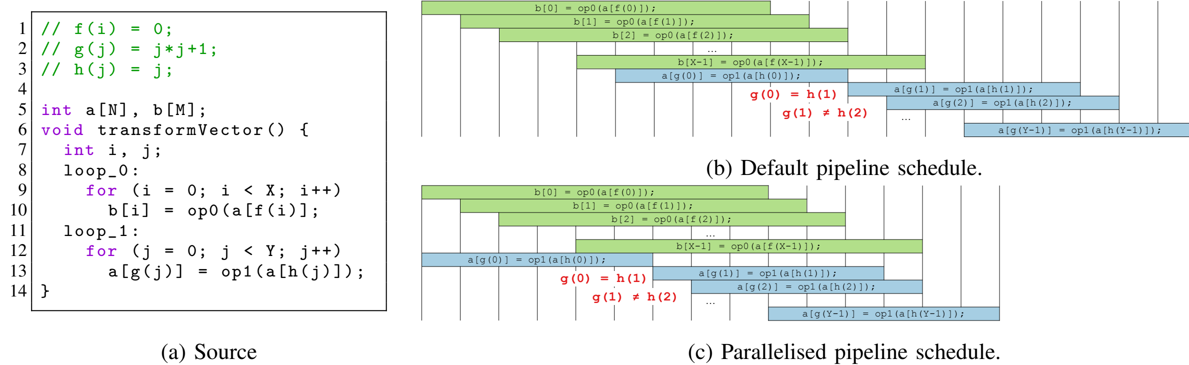
1) Intra-BB data dependences: these can be respected by placing handshaking connections between the corresponding hardware operations in the circuit.
2) Intra-BB memory dependences: these can be kept in the original program order using *load-store queues* (LSQs) [2]. An LSQ is a hardware component that schedules memory operations at run time.
3) Inter-BB data dependences: these can be respected using handshaking connections, as in (1), and additionally by starting BBs in strict program order, so that the inputs of each BB are accepted in program order [3].
4) Inter-BB memory dependences: these can be respected by starting BBs in strict program order and using an LSQ.

In all cases, existing dynamically scheduled HLS tools allow out-of-order execution *within* a BB, but require different BBs to start in-order, even when some BBs are independent and could start in parallel. This, naturally, leads to missed opportunities for performance improvements.

In this work, we focus on cases (3) and (4) above. We find BBs that can be started out-of-order (or even simultaneously), and use static analysis (powered by the Microsoft Boogie verification engine [4]) to ensure that inter-BB dependences are still respected. We tackle two problems: 1) How to automatically identify BBs that can safely start in parallel? 2) How to synthesise efficient hardware that can start BBs in parallel? Our main contributions include:

- a technique that automatically identifies sequences of consecutive subgraphs from the control-flow graph (CFG) of a sequential program and reschedules these subgraphs for parallelism using the Microsoft Boogie verifier;

1946-1488/22/$31.00 ©2022 IEEE
DOI 10.1109/FPL57034.2022.00045



(a) Source

(b) Default pipeline schedule.

(c) Parallelised pipeline schedule.

Fig. 2: Motivating example. Assume no dependence between two loops. The dynamically scheduled hardware from the original Dynamatic [1] computes in the schedule in (b). Our work achieves an optimised schedule in (c).

- a transformation pass that efficiently parallelises these subgraphs in hardware; and
- results and analysis showing that our approach, compared to original Dynamatic, achieves a 4× average speedup (and a 7.3× speedup when combined with our recent work on C-slow pipelining [5]), and almost the same circuit area.

The rest of our paper is organised as follows: Sec. II provides a motivating example of our work. Sec. III introduces existing works on dynamically scheduled HLS and parallelising CFGs for HLS. Sec. IV explains our approach in detail. Sec. V evaluates the effectiveness of our toolflow.

II. Motivating Example

This section illustrates a motivating example of parallelising two sequential loops in dynamically scheduled hardware. Fig. 2a shows an example of two sequential loops, `loop_0` and `loop_1`. In each iteration of `loop_0`, an element at index `f(i)` of array `a` is loaded and processed by a function `op0`. The result is stored to an element at index `i` of array `b`. In each iteration of `loop_1`, an element at index `h(j)` of array `a` is loaded and processed by a function `op1`. The result is stored back to array `a` at index `g(j)`. For simplicity, let `f(i) = 0`, `g(j) = j*j+1` and `h(j) = j`. Hence, there is no memory dependence between two loops, that is, $\forall_{0\leq i<\mathtt{X}}.\forall_{0\leq j<\mathtt{Y}}.\,\mathtt{f}(i)\neq\mathtt{g}(j)$.

Dynamatic [1], the state-of-the-art dynamic scheduled HLS tool, synthesises hardware that computes in a schedule shown in Fig. 2b. The green bars represent the pipeline schedule of `loop_0`, and the blue bars represent the pipeline schedule of `loop_1`. In `loop_1`, the interval between the starts of consecutive iterations, known as the *initiation interval* (II), is variable because of the dynamic inter-iteration dependence between loading from `a[h(j)]` and storing to `a[g(j)]`. For instance, if we suppose that `g` and `h` are defined such that $\mathtt{g}(0)=\mathtt{h}(1)$, then the first two iterations must be executed sequentially, and if we further suppose that $\mathtt{g}(1)\neq\mathtt{h}(2)$, then the second and third iterations are pipelined with an II of 1.

However, `loop_1` is stalled until all the iterations in `loop_0` have started even though it has no dependence on `loop_0`.

TABLE I: Elastic components for dynamically scheduled HLS.

Component	Description
Merge:	takes the input data from an arbitrary predecessor and propagates it to its single successor.
Fork:	takes the input data from its single predecessor and replicates it to each of its multiple successors.
Join:	triggers its single successor only when the input data of its all predecessors is available.
Branch:	takes the data from its data predecessor and propagates it to one of its multiple successors based on the select value from its control predecessor.

The reason is that Dynamatic forces all the BBs to start sequentially to preserve any potential inter-BB dependence, such as the inter-iteration memory dependence in `loop_1`. For this example, each loop iteration is a single BB, and at most one loop iteration starts in each clock cycle.

An optimised schedule is shown in Fig. 2c. In the figure, both loops start from the first cycle and iterate in parallel, resulting in better performance. Existing approaches cannot achieve the optimised schedule: static scheduling can start `loop_0` and `loop_1` simultaneously such as using multi-threading in LegUp HLS [6], but `loop_1` is sequential as the static scheduler assumes the worst case of dependence and timing; dynamic scheduling has a better throughput of `loop_1`, but cannot start it simultaneously with `loop_0`.

Besides, determining the absence of dependence between these two loops for complex `f(i)`, `g(j)` and `h(j)` is challenging. In this paper, our toolflow 1) generates a Boogie program to formally prove that starting `loop_0` and `loop_1` simultaneously cannot break memory dependence and 2) parallelises these loops in dynamically scheduled hardware if they are proved independent. The Boogie program generated for this example is explained later (in Fig. 3).

The transformation for the example in Fig. 2 is demonstrated in Fig. 1. Fig. 1a shows the CFG generated by the original Dynamatic. The CFG consists of a set of pre-defined components, as listed in Tab. I. As indicated by the red arrows, a control token enters the upper block and triggers all the operations in the first iteration of `loop_0`. It circulates within the upper

block for X cycles and then enters the lower block to start loop_1. Fig. 1b shows a parallelised CFG by our toolflow. Initially, a control token is forked into two tokens. These two tokens simultaneously trigger loop_0 and loop_1. A join is used to synchronise the two tokens when they exit these loops. Both designs use the same hardware; yet, Fig. 1b uses these resources in a more efficient way by allowing the two loops to be used in parallel, reducing the overall execution time. The rest of the paper explains the details of our approach.

III. Background

This section first reviews related works in existing HLS tools that use dynamic scheduling. We also compare related works on parallelising CFGs for HLS with our work.

A. Dynamically Scheduled High-Level Synthesis

Most HLS tools such as Xilinx Vivado HLS [7] and Dynamatic [1] translate an input program into an intermediate representation (IR) such as LLVM IR, and then transform the IR into a control data flow graph (CDFG) for scheduling [8]. A CDFG is a two-level directed graph that contains a set of vertices connected by edges. The top level is a control-flow graph (CFG), where each vertex represents a basic block (BB) in the IR, and each edge represents the control flow. At the lower level, each vertex is also a data-flow graph (DFG), where each sub-vertex inside the DFG represents an operation in the BB and each sub-edge represents a data dependence. The CDFG is used as part of the dependence constraints in both static and dynamic scheduling [1, 9].

In the world of dynamic scheduled HLS, initial work was studied by Page and Luk [10], which maps occam programs into hardware and has been extended to support a commercial language named Handel-C [11]. The idea of mapping a C program into a netlist of pre-defined hardware components has been studied in both asynchronous [12] and synchronous [1] worlds. Sayuri and Nagisa [13] propose a method that synthesises single-level loops into dynamically scheduled circuits. Josipović *et al.* [1] propose an open-sourced HLS tool named 'Dynamatic' that automatically translates a program into a dynamically pipelined hardware. Dynamatic uses a set of pre-defined components with handshake connections formalised by Carloni *et al.* [14]. Each edge in the CDFG of the input program is translated to a handshake connection between components. This allows a component to execute at the earliest time when all its inputs are valid. The memory dependence is controlled by load-store queues (LSQs). An LSQ exploits out-of-order memory accesses by checking memory dependence in program order at run time [2] and early executing those independent memory accesses.

Dynamatic parallelises DFGs within and across BBs for high performance, but the CFG still starts BBs sequentially. Sequentially starting BBs is required to respect inter-BB dependences at run time. An unverified BB schedule may cause an error. Our toolflow uses Boogie to formally prove that the transformed BB schedule cannot break any dependence, such that the synthesised hardware is still correct.

B. Parallelising CFG for HLS

Automatically parallelising a CFG of a sequential program has been well-studied in the software compiler world [15]. Traditional approaches exploit BB parallelism using polyhedral analysers such as Pluto [16] and Polly [17]. These tools automatically parallelise code that contains affine memory accesses [18, 19] and have been widely used in HLS to parallelise hardware kernels [20, 21, 22, 23]. However, polyhedral analysis is not applicable when analysing irregular memory patterns such as non-affine memory accesses, which are commonly seen in applications amenable for dynamic scheduling, such as tumour detection [24] and video rendering [25].

Recently, there are works that use formal verification to prove the absence of dependence to exploit hardware parallelism. Zhou *et al.* [26] propose a satisfiability-modulo theory (SMT)-based [27] approach to verify absence of memory contention in banked memory among parallel kernels. Cheng *et al.* propose a Boogie-based approach for simplifying memory arbitration for multi-threaded hardware [28]. Microsoft Boogie [4] is an automated program verifier on top of SMT solvers. It uses its own intermediate verification language to describe the behaviour of a program to be verified, which can be automatically decoded into SMT queries. An SMT solver under Boogie then reasons the program behaviour, including the values that its variables may take. Our work also uses Boogie but for parallelising BBs in dynamically scheduled hardware.

Mapping a parallel BB schedule into hardware has also been widely studied. Initial work by Cabrera *et al.* [29] proposes an OpenMP extension to off-load computation to an FPGA. Leow *et al.* [30] propose a framework that maps OpenMP code in Handel-C [11] to VHDL programs. Choi *et al.* [31] propose a plugin that synthesises both OpenMP and Pthreads C programs into multi-threaded hardware, used in an open-sourced HLS tool named LegUp [6]. Gupta *et al.* propose an HLS tool named SPARK that parallelises control flow with speculation [32]. These works either require user annotation or only use static scheduling, while our approach only uses automated dynamic scheduling.

Finally, there are works on simultaneously starting BB in dynamically scheduled HLS. Cheng *et al.* [33] propose an HLS tool named DASS that allows each statically scheduled component to act as a static island in a dynamically scheduled circuit. Each island is still statically scheduled, while our toolflow only uses dynamic scheduling. The closest piece of work to this paper proposes C-slow pipelining for dynamically scheduled hardware [5]. This work parallelises BBs inside a nested loop to achieve better throughput of the innermost loop. However, it only works for nested loops and cannot optimise code such as sequential loops in Fig. 2, while our approach can parallelise these BBs.

IV. Method

In this section, we first formalise the problem of sequentially starting BB execution. We then introduce an approach that statically constructs subgraphs from the CFG of a program

```
1  procedure pickOneMemoryAccess() returns (valid: bool,
2  addr: Index, array: Array, subgraphID: Index,
3  type: MemoryType) {
4    loop_0: for (i = 0; i < X; i++) {
5      // b[i] = op0(a[f(i)]);
6      if (*) { valid := true; addr := f(i); array := a;
7               subgraphID := 0; type := LOAD; return; }
8      if (*) { valid := true; addr := i; array := b;
9               subgraphID := 0; type := STORE; return; }
       }
10   loop_1: for (j = 0; j < Y; j++) {
11     // a[g(i)] = op1(a[h(j)]);
12     if (*) { valid := true; addr := h(j); array := a;
13              subgraphID := 1; type := LOAD; return; }
14     if (*) { valid := true; addr := g(j); array := a;
15              subgraphID := 1; type := STORE; return; }
       }
16   valid := false; return; }
```

(a) Procedure that arbitrarily picks a memory access.

```
1  procedure main() {
2    // assume that all the arrays have arbitrary values
3    havoc a, b;
4    // valid: whether the returned memory access is valid
5    // addr: which address the memory access touches
6    // array: which array the memory access touches
7    // subgraphID: which subgraph the memory access is in
8    // type: the type of memory access, either load/store
9    call valid_0, addr_0, array_0, subgraphID_0, type_0
       := pickOneMemoryAccess();
10   call valid_1, addr_1, array_1, subgraphID_1, type_1
       := pickOneMemoryAccess();
11   assert !valid_0 || !valid_1 ||
12          subgraphID_0 == subgraphID_1 ||
13          array_0 != array_1 ||
14          (type_0 == LOAD && type_1 == LOAD) ||
15          addr_0 != addr_1;
16 }
```

(b) Main procedure that proves the absence of dependence.

Fig. 3: A Boogie program generated for the example in Fig. 2. It tries to prove the absence of memory dependence between two sequential loops `loop_0` and `loop_1`.

and reschedules them. Next, we show how to transform the hardware to achieve a parallel BB starting schedule. Finally, we demonstrate how our work is integrated as a plugin to the open-sourced Dynamatic HLS tool for prototyping.

A. Problem Formulation

Here we formalise our problem of starting BB in parallel. Let $x \prec y$ denote that x begins execution at a time less than the time y begins execution, and let $x \preceq y$ denote that x begins execution at a time no greater than the time y begins execution. In dynamic scheduling, a BB b_{k_1} has inter-BB dependence on b_{k_2}, it must start after the start of b_{k_2}, *i.e.* $b_{k_2} \prec b_{k_1}$.

The search space for BBs that can start in parallel could be huge, and it scales exponentially with the code size. In order to increase scalability, we limit our scope to loops. Each loop forms a subgraph in the CFG for analysis. Parallelising BBs outside any loop adds significant search time but has negligible improvement in latency. We define following terms:

- $G = \{g_1, g_2, g_3, ...\}$: A set of consecutive subgraphs in the CFG of the program.
- O : $g_{1,0} \prec g_{2,0} \prec g_{3,0} \prec g_{1,1} \prec ...$: The order of subgraph execution in the original program order. $g_{i,j}$ represents the jth iteration of subgraph g_i.
- $B_{g_i} = \{b_1, b_2, b_3, ...\}$: The set of all BBs in subgraph g_i .
- O_{g_i} : $b_{1,0} \prec b_{2,0} \prec b_{3,0} \prec b_{1,1} \prec ...$: The original program order of BB execution in subgraph g_i. $b_{k,j}$ represents the jth iteration of BB b_k.

Dynamatic starts BB execution in the order that combines O and O_{g_i} lexicographically. However, for an order O : ... $\prec g_{i_1,j} \prec g_{i_2,j} \prec$ ..., if it is proven that g_{i_1} cannot have dependence with g_{i_2}, then O' : ... $\prec g_{i_2,j} \preceq g_{i_1,j} \prec$ ... is also memory legal. O' exploits parallelism between g_{i_1} and g_{i_2}, which could achieve better performance.

The optimised order O' still respect all the dependences. First, only the BB order is changed, where the intra-BB dependences remain the same. The inter-BB dependences are respected as only the independent BBs are made out-of-order. The following sections explain two main problems solved by our work: 1) How to efficiently determine a large set of G and a highly parallelised order O' for G? 2) How to map the parallelised order O' into efficient hardware?

B. Searching and Scheduling Subgraphs

Here we first show how to construct sets of subgraphs from a sequential program, where subgraphs in the same set may start in parallel. Then we show how to use Boogie to check dependence among these subgraphs, potentially resulting in a parallel BB schedule.

Given an input program, our toolflow analyses sequential loops in each depth and constructs a number of sets of subgraphs. Each set contains several consecutive sequential loops at the same depth, where each loop forms a subgraph. For instance, the example in Fig. 2 has a set of two subgraphs, corresponding to `loop_0` and `loop_1`. Our toolflow then checks the dependence among the subgraphs for each set. Dynamatic translates data dependence into handshake connections in hardware for correctness. Our toolflow does not change these connections so the data dependence is still preserved. For memory dependences, our toolflow generates a Boogie program to prove the absence of dependence among subgraphs. For this example, Boogie proves that the two loops do not conflict on any memory locations and therefore can be safely reordered.

Boogie uses its own language with has its own constructions [4]. Here we list the ones used in this paper:

1) `if (*) {A} else {B}` is a non-deterministic choice. The program arbitrarily does `A` or `B`.
2) `havoc x` assigns an arbitrary values to a variable or an array `x`, used to capture all the possible values of `x`.
3) `assert c` proves the condition `c` for all the values that the variables in `c` may take.

For example, Fig. 3 shows the Boogie program that proves the absence of a dependence between `loop_0` and `loop_1` in Fig. 2. The Boogie program consists of two procedures. First,

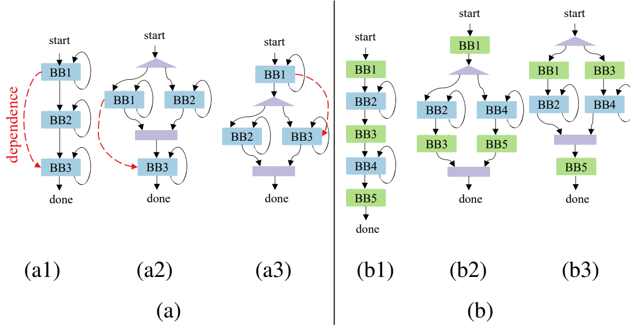


Fig. 4: A CFG may be parallelised differently, depending on (a) parallelising in top-to-bottom/bottom-to-top order, and (b) grouping BBs with the loop before/after. The dashed arrows represent memory dependence.

the procedure in Fig. 3a describes the behaviour of function `transformVector` and arbitrarily picks a memory access during the whole execution. The procedure returns a few parameters for analysis, as listed in lines 4-8 in Fig. 3b. The `for` loop structures are automatically translated using an open-sourced tool named EASY [28]. In the rest of the function, each memory operation is translated to a non-deterministic choice `if(*)`. It arbitrarily returns the parameters of a memory operation or continues the program. If all the memory operations are skipped, the procedure returns an invalid state in line 16. The non-deterministic choices over-approximate the exact memory locations to a set of potential memory locations. For instance, any memory location accessed by the code in Fig. 2a is reachable by the procedure in Fig. 3a. The assertions in Fig. 3b must hold for any possible memory location returned by the procedure in Fig. 3a to pass verification.

Second, Fig. 3b shows the main procedure. In line 3, the verifier assumes both arrays hold arbitrary values, making the verification input-independent. Then, the verifier arbitrarily picks two memory accesses in lines 9-10. Each memory access can capture any memory access during the whole execution of `transformVector`. The assertion describes the dependence constraint to be proved that for any two valid memory accesses (line 11), if they are in different subgraphs (line 12), they must be independent. Lines 13-15 describe the independence, where they either touch different arrays or different indices, or they are both load. If the assertion always holds, then it is safe to parallelise `loop_0` and `loop_1`.

Our toolflow generates $\frac{k(k-1)}{2}$ assertions for k subgraphs, because that is the number of ways of picking 2 subgraphs from k. The subgraphs are rescheduled based on the verification results. If a subgraph is independent of any its preceding subgraphs within a distance of n, it can simultaneously start with its $(m-n)$th last subgraph. For case of two or more consecutive subgraphs that are all mutually independent, it is straightforward to schedule them all in parallel. However, a sequence of subgraphs that are neither completely independent nor completely dependent may result in several possible solutions. For instance, the CFG in Fig. 4a1 contains three consecutive loops, BB1, BB2, and BB3. BB1 and BB2 can be parallelised, as can BB2 and BB3, but BB1 and BB3 cannot. We therefore have to choose between parallelising them as in Fig. 4a2 or in Fig. 4a3. Our current approach greedily parallelises BBs in top-to-bottom order, so yields Fig. 4a2 by default, but this order can be overridden via a user option. It may be profitable in future work to consider Fig. 4a3 as an alternative if BB2 and BB3 have more closely matched latencies.

Second, the BBs between sequential loops can be included in a subgraph of either loop, resulting in several solutions. For instance, Fig. 4b1 can be parallelised to Fig. 4b2 or to Fig. 4b3. In Fig. 4b2, BB2 is grouped with its succeeding loop BB3, and so is BB4. In Fig. 4b3, the BBs are grouped with their preceding loops. This may result in different verification results which affect whether the subgraphs can be parallelised. For instance, if BB3 depends on BB2, then Fig. 4b2 is memory-legal and Fig. 4b3 is invalid (our toolflow will keep the CFG as in Fig. 4b1). This grouping can be controlled via a user option.

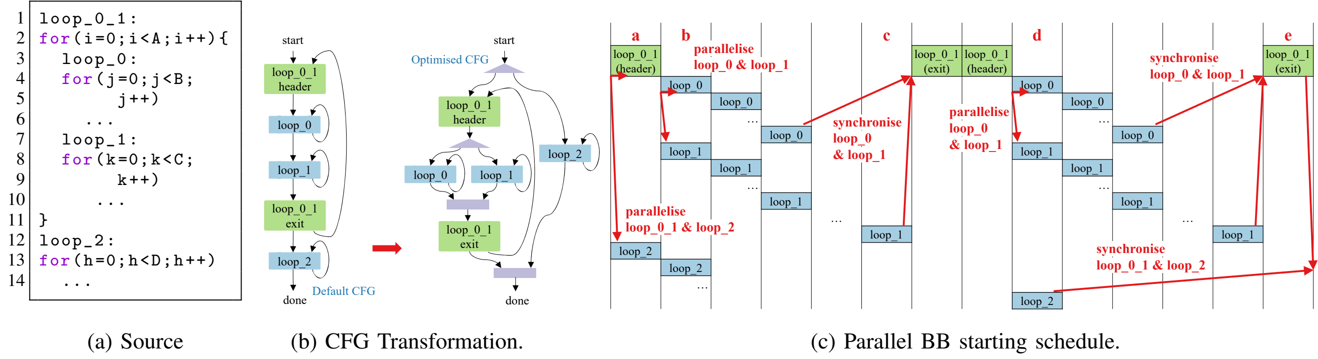
C. Parallelising Hardware

We here explain how to construct dynamically scheduled hardware in which BBs can start simultaneously. First, we illustrate how to insert additional components to enable BB parallelism. Second, we show how to simplify the data flow to avoid unnecessary stalls for subgraphs.

1) Inserting Components for Parallelism: With given sets of subgraphs that start simultaneously, our toolflow inserts additional components into the dynamically scheduled hardware to enable parallelism. For each set, our toolflow first finds the start of the first subgraph and the exit of the last subgraph in the program order. The trigger of the first subgraph is forked to trigger the other subgraphs in the set. The exit of the last subgraphs is joined with the exits of the other subgraphs and then triggers its succeeding BB. For the example in Fig. 1b, the start of the function is forked to trigger both `loop_0` and `loop_1`. A join is used to synchronise the BB starting siganls in `loop_0` and `loop_1`. The join waits for all the BBs in both loops to start and then starts the succeeding BB of the loops.

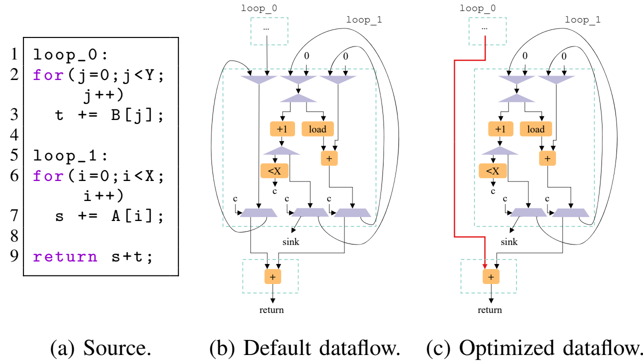
The BB starting order O' is now out-of-order, but the computed data must be in-order. The transformation above ensures the order of data does not affect the correctness. Since we only target loops, only the muxes at the header of the loops are affected. Outside of the loops to be parallelised, the order remains unmodified. When each parallelised loop starts, a token enters the loop and circulates through the loop exactly as the program order. The parallelised loop outputs are synchronised by the join, thus, everything that happens later remains in order. Only the BB orders among these parallelised loops are out-of-order, which have been proven independent.

An advantage of such transformation is that the execution of parallelised subgraphs and their succeeding BB are in parallel, although they still start in order. The memory dependences between these subgraphs and the succeeding BB are still respected at run time as they start in order. This



(a) Source (b) CFG Transformation. (c) Parallel BB starting schedule.

Fig. 5: An example of parallelising BB starting schedule by CFG transformation. There are two sets of sequential loops in different depths. Assuming all the loops are independent, each set of sequential loops start simultaneously after the transformation in (b). (c) only shows the time when a BB starts, where a BB may take multiple cycles to execute.



(a) Source. (b) Default dataflow. (c) Optimized dataflow.

Fig. 6: An example of simplify data flow for live variables. `t` is a live variable in line 3, but not used in `loop_1`. `t` circulates in `loop_1` to preserve liveness but is seen as data dependences, stalling `loop_1` before `t` is valid. Our toolflow identifies and removes these cycles, such that `loop_1` can start earlier.

effect qualitatively corresponds to what standard dynamically scheduled hardware exhibits yet, in that case, only on a single BB at a time. Compared to traditional static scheduling, which only starts the succeeding BB when all the subgraphs finish execution, our design can achieve better performance.

Fig. 5 shows an example of parallelising nested parallel subgraphs. The code contains two sequential loops, `loop_0_1` and `loop_2`. Loop `loop_0_1` is a nested loop that contains two sequential loops, `loop_0` and `loop_1`. For simplicity, assume that there is no dependence between any two loops.

Our toolflow constructs two sets of subgraphs in two depths, allowing more parallelism in the CFGs. One set contains `loop_0_1` and `loop_2`, and the other set contains `loop_0` and `loop_1`. The transformation of CFG is illutrated in Fig. 5b. `loop_0_1` and `loop_2` are parallelised at the start the program, and `loop_0` and `loop_1` are further parallelised inside `loop_0_1`. The corresponding BB starting schedule is demonstrated in Fig. 5c, which only shows the time when each BB starts. A BB may have a long latency and execute in parallel with other BBs.

2) Forwarding Variables in Data Flow: The second step is to simplify the data flow of live variables for parallelising sequential loops. Dynamatic directly translates the CFDG of an input program into a hardware dataflow graph. In the data flow graph, each vertex represents a hardware operation, and each edge represents a data dependence between two operations.

The data flow of a loop uses cycles for each variable that has carried dependence. The data circulates in the cycle and updates its value in each iteration. However, such approach also maintain all the live variables in these cycles while executing a loop, even when they are not used inside the loop. The edges of these cycles are seen as data dependences in the hardware, where the edges for unused live variables could cause unnecessary pipeline stalls.

For example, the loops in Fig. 6a can be parallelised. `loop_0` accumulates array `B` onto `t`, and `loop_1` accumulates array `A` onto `s`. The sum of `s` and `t` is returned. The dataflow graph of `loop_1` is shown in Fig. 6b. The loop iterator `i` and the variable `s` have carried dependence in `loop_1`. They are kept and updated in the middle and right cycles. The result of `loop_0`, `t`, is still live and required by addition in line 9. `t` is kept in the left cycle, circulating with `i` and `s`.

`loop_1` is stalled by the absence of `t` even when parallelised with `loop_0`, but `t` is not needed by `loop_1`. In order to remove these unnecessary cycles, our toolflow checks whether a live variable is used in the loop. If it is not, our toolflow removes the corresponding cycle and directly forwards the variable to its next used BB. Fig. 6c illustrates the transformed dataflow graph. `t` is now directly forwarded to the final adder, enabling two loops to start simultaneously.

3) Handling LSQs: The parallel BB schedule also affects the LSQs. First, the original Dynamatic starts BB sequentially, where the LSQ expects sequential BB allocation. Our parallelised schedule allows multiple BBs to start simultaneously; therefore, we place a round-robin arbiter for the LSQ to serialise the allocations. The out-of-order allocation still preserves correctness as the simultaneous BB requests have been statically proven independent by our approach in Sec. IV-B.

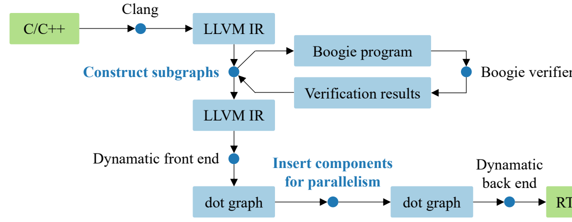


Fig. 7: Our work integrated into the open source Dynamatic tool [1]. Our contribution is highlighted in bold text.

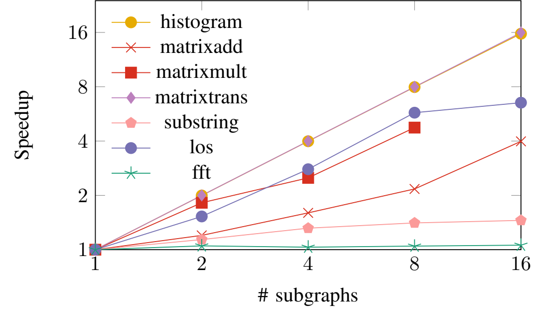


Fig. 8: Speedup, compared to original Dynamatic, as more subgraphs in the CFG are parallelised.

Second, the arbiter may cause deadlock if the LSQ depth is not sufficient to consume and reorder all memory acceses (*e.g.* a later access may be stuck in an LSQ waiting for a token from an earlier access, but the earlier access cannot enter the LSQ if it is full, thus never supplying the token). This issue has been extensively explored in the context of shared resources in dataflow circuits [34]; similarly to what is suggested in this work, the appropriate LSQ size could be determined based on the number of overlapping loop iterations and their IIs. Although systematically determining the minimal allowed LSQ depth is out of the scope of this paper, we here assume a conservative LSQ size that ensures that deadlock never occurs in the benchmarks we consider. We note that minimising the LSQ is orthogonal to our contribution and could only positively impact our results (by reducing circuit area and improving its critical path).

D. Tool Flow

Our toolflow is implemented as a set of LLVM passes and integrated into the open-sourced HLS tool Dynamatic for prototyping. As illustrated in Fig. 7, the input C program is first lowered into LLVM IR and analysed by our subgraph constructor. It generates Boogie assertions and calls Boogie verifier to automatically verify the absence of dependence between any two subgraphs. Then the constructor constructs sets of subgraphs and reschedules them. The front end of Dynamatic translates the LLVM IR into a dot graph that represents the hardware netlist. Our back-end toolflow inserts additional components and simplifies the unnecessary cycles for the live variables, resulting in a new hardware design in the form of a dot graph. Finally, the back end of Dynamatic transforms the new dot graph to RTL code, representing the final hardware design.

V. Experiments

We compare our work with Xilinx Vivado HLS [7], the original Dynamatic [1], and Dynamatic with C-slow pipelining [5]. To make the comparison as controlled as possible, all the approaches only use scheduling, pipelining and array partitioning. We use two benchmark sets to evaluate the designs in terms of total circuit area and wall-clock time. Cycle counts were obtained using the Vivado XSIM simulator, and area results were obtained from the post-Place & Synthesis report in Vivado. We used UltraScale+ family of FPGA devices for experiments, and the version of Xilinx software is 2019.2.

A. Benchmarks

We use two open-sourced benchmark sets for evaluation. One is the LegUp benchmark set by Chen and Anderson [35] for evaluating multi-threaded HLS. The LegUp benchmark set manually specifies the threads using Pthreads [36]. We inlined all the threads to a sequential program. The other benchmark set the C-slow pipelining benchmark set by Cheng *et al.* [5] for evaluating dynamically scheduled HLS. We only include the benchmarks where our approach is applicable. The other benchmarks will have the same results as the original Dynamatic. The second benchmark set aims to evaluate dynamic loop pipelining and contains few loop kernels. In order to create more opportunities for our optimisation to be applied, we unrolled the outermost loops by a factor of 8. This is the largest factor that still led to the designs fitting on our target FPGA. We also partitioned the memory in block scheme to increase memory bandwidth. The benchmarks that we used are listed as follows:

- `histogram` constructs a histogram from an integer array,
- `matrixadd` sums a float array,
- `matrixmult` multiplies two float matrices,
- `matrixtrans` transposes a single matrix,
- `substring` searches for a pattern in an input string,
- `los` checks for obstacles on a map,
- `fft` performs the fast Fourier transformation,
- `trVecAccum` transforms a triangular matrix,
- `covariance` computes the covariance matrix,
- `syr2k` is a symmetric rank-2k matrix update, and
- `gesummv` is scalar, vector and matrix multiplication.

B. Results

Figure 8 assesses the extent to which more parallelisation of subgraphs leads to more speedups compared to the original Dynamatic, using the seven LegUp benchmarks. We see that all the lines except `matrixadd` indicate speedup factors above 1. Placing more subgraphs in parallel leads to more speedup, with `histogram` and `matrixtrans` achieving optimal speedups. In the `matrixadd` benchmark, two reasons for the lack of speedup are: 1) that there are other parts of the CFG that have to be started sequentially, and 2) that the memory is naively partitioned in a block scheme, so the

TABLE II: Evaluation of our work on two benchmark sets. For each benchmark, we highlight the **best** results in each dimension.

(a) Evaluation on the LegUp benchmarks. vhls = Vivado HLS; dhls = original Dynamatic; cslow = C-slow pipelining by Cheng *et al.* [5]; ours = our work; both = our work + C-slow pipelining. The code size lists the number of loops, BBs, instructions and extracted subgraphs.

Benchmarks	Code size				LUTs (1000s)					DSPs					Cycles (1000s)					Fmax (MHz)					Relative area-delay product				
	loops	bbs	insts	graphs	vhls	dhls	ours	cslow	both	vhls	dhls	ours	cslow	both	vhls	dhls	ours	cslow	both	vhls	dhls	ours	cslow	both	vhls	dhls	ours	cslow	both
histogram	9	91	384	9	**1.67**	156	156	156	156	0	0	0	0	0	197	317	**39.8**	317	**39.8**	**464**	57.7	58.3	57.7	58.3	**1**	1212	150.9	1212	150.9
matrixadd	8	17	112	8	**1.11**	9.15	9.17	9.15	9.17	**2**	30	30	30	30	262	106	**48.9**	106	**48.9**	**159**	110	110	110	110	**1**	4.8	2.2	4.8	2.2
matrixmult	72	145	1521	72	**6.87**	79.4	79.8	101	100	**5**	320	320	320	320	4195	1090	229	**164**	**164**	**155**	123	104	83.3	72.3	1	3.8	**0.9**	1.1	1.2
matrixtrans	8	17	81	8	**0.103**	2.67	2.69	2.67	2.69	0	0	0	0	0	65.6	65.6	**8.2**	65.6	**8.2**	**562**	227	210	227	210	**1**	64	8.7	64	8.7
substring	16	54	255	8	**0.938**	14.4	14.6	14.4	14.6	0	0	0	0	0	**98.3**	217	154	217	154	**470**	126	129	126	129	**1**	126	88.9	126	88.9
los	24	89	513	8	**2.68**	46.5	46.1	46.5	46.1	0	0	0	0	0	48.8	114	**19.9**	114	**19.9**	281	272	**282**	272	**282**	**1**	41.8	7	41.8	7
fft	24	65	457	8	**2.65**	351	351	351	351	**16**	192	192	192	192	86	5.39	**5.15**	5.39	**5.15**	**155**	81.8	103	81.8	103	**1**	15.7	12	15.7	12
Norm. median						1	**1**	1	**1**		1	1	1	1		1	**0.21**	1	**0.17**		1	1.01	1	1.01	**1**	41.8	8.7	41.8	8.7

(b) Evaluation on the C-slow pipelining benchmarks. unroll = original Dynamatic taking the program where all outermost loops are unrolled by a factor of 8; ours = unroll + our work; both = unroll + our work + C-slow pipelining.

Benchmarks	Code size (unrolled)				LUTs (1000s)					DSPs					Cycles (1000s)					Fmax (MHz)					Relative area-delay product				
	loops	bbs	insts	graphs	dhls	unroll	ours	cslow	both	dhls	unroll	ours	cslow	both	dhls	unroll	ours	cslow	both	dhls	unroll	ours	cslow	both	dhls	unroll	ours	cslow	both
trVecAccum	16	49	249	8	**18.9**	149	150	19.8	153	**5**	40	40	**5**	40	389	393	161	256	**33.2**	**188**	132	121	157	117	1	11.27	5.05	**0.83**	1.11
covariance	48	113	593	24	27.1	56.6	55.5	**26.8**	65.7	**9**	72	72	**9**	72	698	605	77.1	263	**33.6**	72.1	**132**	102	89.7	102	1	0.99	0.16	0.30	**0.08**
syr2k	24	49	385	8	**4.14**	30.5	30.7	4.57	34.4	**19**	152	152	**19**	152	674	602	84.1	255	**33.9**	125	126	98.9	**130**	128	1	6.53	1.16	**0.40**	0.41
gesummv	16	33	297	8	**3.96**	26.9	26.9	4.56	30.2	**18**	144	144	**18**	144	797	787	327	524	**66.6**	128	162	**163**	119	120	1	5.33	2.20	0.82	**0.68**
Norm. median					1	7.10	**7.09**	1.08	7.85	1	8	8	1	8	1	1	**0.27**	0.52	**0.07**	1	1.13	1.03	0.99	0.98	1	5.93	1.68	0.61	**0.54**

memory bandwidth is limited and there is serious contention between BBs for the LSQs.

Detailed results for the LegUp benchmarks using eight subgraphs are shown in Tab. IIa. We observe the following:

1) Static scheduling (Vivado HLS) is the clear winner in terms of area (see columns 'LUTs' and 'DSPs'), but in the context of dynamic scheduling, our approach brings only a negligible area overhead because we only insert small components into the hardware.
2) Our approach requires substantially fewer cycles than original Dynamatic thanks to the parallelism it exploits between BBs (see column 'Cycles').
3) The only benchmark where C-slow pipelining wins is the `matrixmult` benchmark (see column 'Cycles'), which contains nested loops. Fortunately, our approach is compatible and complementary to C-slow pipelining, so we can apply both techniques simultaneously to reach the best of both worlds (see the 'both' columns).
4) Our approach achieves a 4× average speedup (see column 'Wall clock time'). We further observe that the area-delay products we obtain are significantly smaller than those of the original Dynamatic.
5) Although Vivado HLS has low performance in cycles, its high clock frequency makes it win for `histogram` and `substring`. Also, it uses if-conversion to simplify BBs (unlike our work), which results in fewer BBs. The BBs in innermost subgraphs still start sequentially, leading to large cycle counts.

For the C-slow pipelining benchmark set (Tab. IIb), we make the following observations:

1) The area-delay product of Dynamatic is significantly worse than Vivado HLS because the version of Dynamatic we used does not support resource sharing leading to significant area overhead (although it is now supported [34]).
2) Unrolling alone is not enough to obtain substantial speedups because the BBs still have to start sequentially (see column 'Cycles → unroll').
3) C-slow pipelining enables a 1.9× average speedup with only 8% area overhead (see column 'Wall clock time'). Our approach achieves a 4.2× average speedup for the unrolled benchmarks. By applying both techniques simultaneously on the unrolled programs, we achieve a 14.3× average speedup with a 10% area overhead. That significant speedup can be attributed in part to the reordering of BBs.

VI. Conclusions

Existing dynamically scheduled HLS tools require all BBs to start in strict program order, in order to respect any inter-BB dependences, regardless of whether dependences are actually present. This leads to missed opportunities for performance improvements by having BBs start simultaneously.

We propose an automated approach to lifting this restriction. We show how to statically identify sequences of consecutive subgraphs in the CFG of a program and reschedule them (with the help of the Microsoft Boogie verifier) to start simultaneously. We then show how to map the optimised schedule into efficient hardware designs. The performance gain is significant (and can be further improved with C-slow pipelining), while the area overhead is negligible. Our plan for future work is to automate the process of optimising subgraph configurations for arbitrary programs in this framework.

Acknowledgment

This work is supported by the EPSRC (EP/P010040/1, EP/R006865/1). For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

References

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

[2] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 125:1–125:19, Sep. 2017.

[3] L. Josipović, A. Guerrieri, and P. Ienne, "From c/c++ code to high-performance dataflow circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.

[4] K. R. M. Leino, "This is boogie 2," June 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[5] J. Cheng, J. Wickerson, and G. A. Constantinides, "Dynamic c-slowing pipelining for hls," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022.

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

[7] Xilinx Vivado HLS, 2022. [Online]. Available: https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html

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

[9] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 433–438. [Online]. Available: https://doi.org/10.1145/1146909.1147025

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

[11] Celoxica, "Handel-C," 2022. [Online]. Available: http://www.celoxica.com

[12] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein, "C to asynchronous dataflow circuits: An end-to-end toolflow," in *IEEE 13th International Workshop on Logic Synthesis (IWLS)*. Temecula, CA: IEEE, Jun 2004.

[13] S. Ota and N. Ishiura, "Synthesis of distributed control circuits for dynamic scheduling across multiple dataflow graphs," in *2019 34th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, 2019, pp. 1–4.

[14] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[15] Intel Compiler, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html#gs.sa60u7

[16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 101–113. [Online]. Available: https://doi.org/10.1145/1375581.1375595

[17] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

[18] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[19] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[20] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 9–18. [Online]. Available: https://doi.org/10.1145/2435264.2435271

[21] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic On-chip Memory Minimization for Data Reuse," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. Napa, CA, USA: IEEE, April 2007, pp. 251–260.

[22] J. Liu, J. Wickerson, and G. A. Constantinides, "Loop splitting for efficient pipelining in high-level synthesis," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 72–79.

[23] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 93–104. [Online]. Available: https://doi.org/10.1145/3431920.3439292

[24] T. Young-Schultz, L. Lilge, S. Brown, and V. Betz, "Using opencl to enable software-like

development of an fpga-accelerated biophotonic cancer treatment simulator," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 86–96. [Online]. Available: https://doi.org/10.1145/3373087.3375300

[25] Q. Sun, A. Taherin, Y. Siatitse, and Y. Zhu, "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*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 97–103. [Online]. Available: https://doi.org/10.1145/3373087.3375317

[26] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A new approach to automatic memory banking using trace-based address mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 179–188. [Online]. Available: https://doi.org/10.1145/3020078.3021734

[27] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[28] J. Cheng, S. T. Fleming, Y. T. Chen, J. Anderson, J. Wickerson, and G. A. Constantinides, "Efficient memory arbitration in high-level synthesis from multi-threaded code," *IEEE Transactions on Computers*, pp.

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

1–1, 2021.

[29] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jiménez-González, "Openmp extensions for fpga accelerators," in *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*. IEEE, 2009, pp. 17–24.

[30] Y. Leow, C. Ng, and W. Wong, "Generating hardware from openmp programs," in *2006 IEEE International Conference on Field Programmable Technology*, 2006, pp. 73–80.

[31] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 270–277.

[32] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: a high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design, 2003. Proceedings.*, 2003, pp. 461–466.

New York, NY, USA: Association for Computing Machinery, 2020, pp. 288–298. [Online]. Available: https://doi.org/10.1145/3373087.3375297

[34] L. Josipović, A. Marmet, A. Guerrieri, and P. Ienne, "Resource sharing in dataflow circuits," in *Proceedings of the 30th IEEE Symposium on Field-Programmable Custom Computing Machines*, New York, May 2022, to appear.

[35] Y. T. Chen and J. H. Anderson, "Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

[36] B. Barney, "POSIX Threads Programming," 2022. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads