Dynamic C-Slow Pipelining for HLS

Jianyi Cheng, John Wickerson and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London, UK
Email: {jianyi.cheng17, j.wickerson, g.constantinides}@imperial.ac.uk

***Abstract*—In high-level synthesis (HLS), loop pipelining allows multiple iterations of a loop to be executed concurrently. The start time of the operations in each iteration can be determined either at compile time (static pipelining) or at run time (dynamic pipelining). There has been recent interest in dynamic pipelining, as it can overcome the conservatism of static analysis, potentially achieving better performance.**

In order to ensure correctness in the presence of memory dependences, existing state-of-the-art dynamic pipelining algorithms schedule control flow between basic blocks in the original program order even if they allow pipelining of data flow. This allows source code to be compiled compositionally, 'stitching together' the resulting hardware components to produce the final hardware design. However, this approach can result in suboptimal throughput.

In this paper we propose a technique to statically determine a *set* of possible memory-legal control flows for nested loops, together with a scheduler component able to select from that set efficiently at run time, enabling dynamic execution of control as a C-slow pipeline. An empirical evaluation on a range of applications suggests that by using this approach, we can obtain 2.9× speedup with 7% area overhead compared to a dynamic scheduling approach with sequential control flow.

I. Introduction

High-level synthesis (HLS) tools use loop pipelining to enable parallelism between loop iterations. The same operation in two consecutive iterations has a time difference in clock cycles, known as the initiation interval (II). Traditional loop pipelining techniques use static scheduling, which determines the start time of each operation at compile time [1, 2]. This results in a constant II for each synthesized loop. Recently there has been interest in dynamic loop pipelining [3], which determines the start time of each operation at run time. The II of a dynamically pipelined loop can vary depending on the data being computed.

Static pipelining enables efficient resource sharing thanks to its predictable execution timing. However, it also causes conservatism in input-dependent dependence analysis which may result in suboptimal throughput. Dynamic pipelining uses a handshake interface between operations which allows immediate execution when an operation has all its required data. However, this also leads to input-dependent hardware behaviour, making resource sharing challenging.

In dynamic pipelining, each operation is synthesized as a single component. These components are stitched together to form the top-level hardware design. Each data dependence between two operations is presented as a handshake connection between them. The memory dependences, however, are handled either by sequentialising all the memory accesses [4], or by using load-store queues (LSQs) to schedule the memory accesses at run time [5]. The former keeps the conservatism in static analysis, which can lead to poor performance. The latter checks the dependence between every pair of memory operations in the original program order and allows the later memory operation to execute earlier if dependences allow. Using LSQs enables out-of-order memory accesses at run time, achieving better performance.

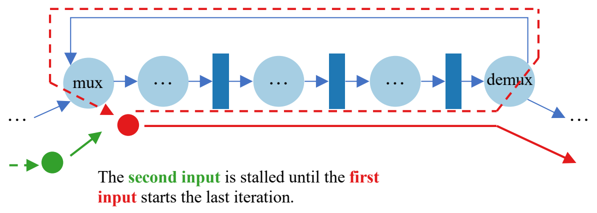
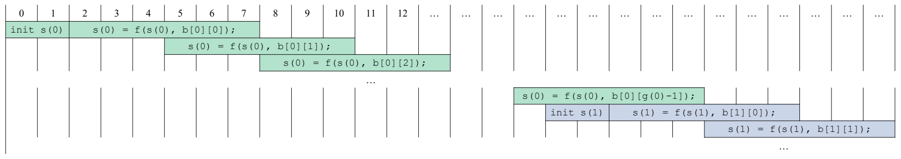

mux
⋯
⋯
⋯
demux
⋯
⋯
The second input is stalled until the first input starts the last iteration.

Fig. 1: An example of a dynamically pipelined loop. The red token is the first input and the green token is the second input. Assume the second input has no dependence with the first input and the loop computes with an II of 3. The restriction of sequential control flow forces the second input to stall until the control flow of the first input finishes. Our approach enables C-slow pipelining and allows the second input to start right after the first input, leading to a doubled throughput.

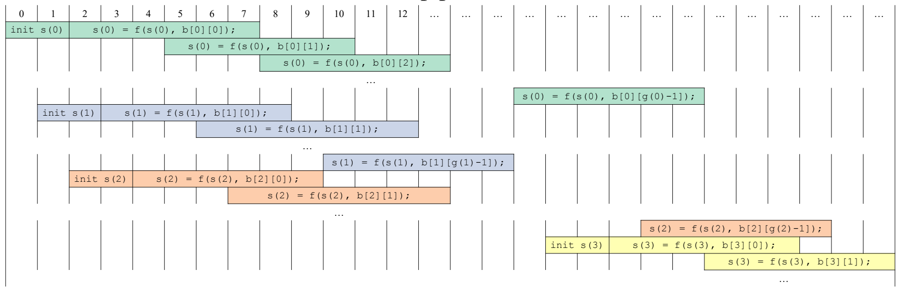
The LSQ used in Dynamatic [3], a leading dynamically scheduled HLS tool, reduces the conservatism in static pipelining. However, it still has a restriction that the hardware must preserve the original program order of memory accesses when it checks dependences at run time. Dynamatic chooses to statically schedule the memory accesses in each basic block (BB), and dynamically allocates these memory accesses into the LSQs when the corresponding BB starts to execute [5]. That means that there is always *at most one* BB *starting* at each clock cycle, even if multiple BBs can be in flight simultaneously. Otherwise, the LSQ cannot determine which memory accesses among multiple simultaneously executing BBs should be allocated first. In order to support LSQs, Dynamatic conservatively keeps control flow sequential.

But sequential control flow can cause suboptimal performance. Fig. 1 shows an example of a dynamically pipelined loop. The LSQ needs to allocate all the memory accesses triggered by the first input before starting executing the second input. For this example, the allocation for the first input completes when the first input starts its last iteration. The second input can only enter the loop and start its first iteration



(b) Default pipeline schedule.

```
float a[N], b[N][M];
void triangleVecAccum() {
  int i, j;
  loop_0:
    for (i = 0; i < N; i++) {
      int s = a[f(i)];
      loop_1:
        for (j = 0; j < N-i; j++)
          s = g(s, b[i][j]);
      a[h(i)] = s;
    }
}
```



(a) Source

(c) Proposed pipeline schedule.

Fig. 2: A motivating example of computing a triangle matrix. Assume there is no inter-iteration dependence in `loop_0`, and each instance of `loop_1` has a minimum II of 3. The default pipeline schedule only starts the second iteration of `loop_0` after the last iteration of `loop_1` in the first iteration of `loop_0` starts. Our approach inserts the following iterations of `loop_0` into the empty slots of its first iteration.

after that time, even when there are no dependences between the inputs to the loop.

In this paper, we show how to lift this restriction and enable C-slow pipelining in HLS for a better throughput. C-Slow pipelining was first proposed by Leiserson *et al.* in 1983 [6]. It is a technique that enables multiple sets of data to be computed in the same hardware in the form of multiple threads. Compared to traditional pipelining, C-slow pipelined hardware executes in an out-of-order control flow.

We tackle two problems in dynamically pipelining nested loops: 1) How can the restriction to sequential control flow be lifted to enable C-slow pipelining? 2) How can the memory correctness of C-slow pipelined hardware be preserved? Our main contributions are:

- a technique that statically determines a set of possible parallel memory-legal control flows for nested loops;
- a transformation pass that enables C-slow pipelining, which executes loop iterations early by inserting their schedules into empty pipeline slots of previous iterations; and
- analysis and results showing that over a set of benchmarks from Cheng *et al.* [7], our approach on average achieves a 2.9× speedup with only a 7% area overhead.

The rest of the paper is organised as follows: Sec. II provides a motivating example of C-slow pipelining. Sec. III introduces existing works in dynamically scheduled HLS, C-slow pipelining and static loop analysis for HLS. Sec. IV explains our formulation and proposed approach in detail. Sec. V evaluates the effectiveness of our tool on a set of applicable benchmarks.

II. Motivating Example

In this section, we use a motivating example to demonstrate the problem of pipelining a nested loop. In Fig. 2a, a loop nest updates the elements in an array `a`. The outer loop `loop_0` loads an element at address `f(i)`. The inner loop `loop_1`, bounded by `N-i`, computes `s` with a row in an array `b`, shown as function `g`. The result is then stored back to array `a` at address `h(i)` at the end of outer-loop iteration.

For simplicity, assume there is no inter-iteration dependence in the outer loop `loop_0`. Assume the latency of function `g` is 3 cycles. An inter-iteration dependence of the inner loop `loop_1` on `s` causes a minimum II of 3. The pipeline schedule of the hardware from vanilla Dynamatic is shown in Fig. 2b. The first iteration of `loop_0` is optimally pipelined with an II of 3 shown as the green bars. However, the second iteration, shown as blue bars, can only start after the the last iteration of `loop_1` in the first iteration of `loop_0` starts.

The schedule shown in Fig. 2c is also correct and achieves better performance. Since the loop II of `loop_0` is 3, the two empty slots between every two consecutive iterations allows the next two iterations of `loop_0` start earlier. The second iteration of `loop_0` now starts one cycle after the start of the first iteration of `loop_0`, followed by the third iteration shown as orange bars. After a last iteration of `loop_1` starts, the current inner-loop instance leaves new empty pipeline slots spare. This triggers the start of the fourth iteration, shown as yellow bars, filling into the new empty slot.

The reason that Dynamatic cannot achieve the schedule in Fig. 2c is that the control flow in the latter schedule is out-of-order. The LSQ cannot retain the original program order of memory accesses and cannot verify the correctness of memory order from the out-of-order control flow, which may lead to wrong results. In this paper, we use static analysis to prove such control flow will still maintain a legal memory access order for a given program, so the LSQ still works correctly for this new program order.

This is an example for which traditional techniques such as loop interchange and loop unrolling do not help, because they only work under stringent constraints. In this example, loop interchanging cannot be applied because the bound of the inner loop depends on its outer loop. Also, loop unrolling does not change the control flow and cannot improve the performance. In this paper, we propose a general approach that works for arbitrary nested loops.

III. Background

This section first reviews related work on existing HLS tools that use dynamic scheduling. Next, we review related work on C-slow pipelining on FPGAs. Finally, we compare existing works on static analysis for HLS loops with our work.

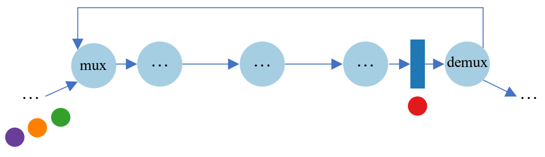
A. Dynamically Scheduled High-Level Synthesis

Dynamically scheduled hardware synthesis from a high-level language was initially proposed as a framework that automatically maps a occam program into a synchronous hardware netlist [8]. This framework was later extended to a commercial language named Handel-C [9]. However, it still requires user effort for hardware optimisation such as pipelining and parallelism. Venkataramani *et al.* [10] propose a dynamically scheduled hardware synthesis framework that takes a C program as input and translates it into an asynchronous circuit. In the framework, the C program is translated into a data flow graph as an intermediate representation. Each node in the data flow represents a pre-defined hardware component that contains its own controlling trigger. Josipović *et al.* [3] bring this dataflow design methodology into synchronous designs. They propose an HLS tool named ‘Dynamatic’ that automatically generates synchronous dataflow circuits from a C program. Dynamatic automatically exploits the parallelism of the hardware by translating each data dependence into a handshake interface. The handshake interface enables immediate computation when the required data is valid at run time, which can achieve high throughput. This approach was later adopted by an HLS tool named CIRCT [4].

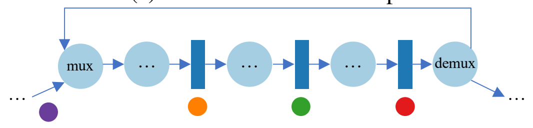
Dynamatic uses a set of components with handshake interface formalised by Carloni *et al.* [11]. These components can be ‘stitched together’ to form a netlist that represents arbitrary programs. A challenge in dynamically scheduled HLS tools is scheduling memory accesses. CIRCT forces memory accesses to execute in the original program order regardless of their dependences, which may lead to significant performance overhead. Dynamatic uses LSQs [5] to monitor and re-order memory accesses at run time. Using LSQs brings significant performance boost at the price of circuit area. This paper tackles the problem of sequential control flow caused by LSQs to achieve better performance.

B. C-Slow Pipelining

C-slow pipelining is a technique that replaces each register in the circuit with C registers to construct C independent threads [12]. The circuit then operates as C-thread hardware while keeping one copy of resources. For instance, a stream of data enters a pipelined loop in Fig. 3a. The loop computes with an II of 1, as illustrated by the presence of one register in the cycle. Assume the loop trip count is N, then the latency of the loop is approximately N cycles for a large N. The overall throughput of the hardware is 1/N and the critical path is the delay of the cycle. Assume that each set of data is independent from other sets. Fig. 3b demonstrates a 3-slowed loop that is functionally equivalent to the one in Fig. 3a. There are three registers in the cycle, evenly distributed in the path. This increases the latency of the hardware to 3N cycles. The loop can iterate with three sets of data in the cycle concurrently. Then the overall throughput of the hardware is approximately 3/(3N) = 1/N, and the critical path is nearly 1/3 of the one in Fig. 3a. A C-slowed loop can either have a better throughput or the better maximum clock frequency to achieve approximately C times speedup.



(a) Initial version of a loop.



(b) A 3-slowed loop.

Fig. 3: An example of 3-slowing a loop. The 3-slowed loop has tripled latency, the same throughput and one-third critical path compared to the initial version.

C-slow pipelining was firstly proposed by Leiserson *et al.* for optimising the critical path of synchronous circuits [6]. Markovskiy and Patel [12] propose a C-slow based-approach to improve the throughput of a microprocessor. Weaver *et al.* [13] propose an automated tool that applies C-slow retiming on a class of applications for certain FPGA families. Our work brings the idea of C-slow pipelining into the dynamic HLS world. We analyse nested loops at code level to determine C for each loop by checking the dependence between inputs to the loop and then apply hardware transformations to achieve C-slow pipelining.

C. Static Loop Analysis in HLS

Analysis for loop pipelining in HLS has been well-studied in the past decades. Zhang and Liu [1] propose a memory-dependence and resource model in their scheduling algorithms. This is further extended by Canis *et al.* [2] by restructuring recurrence for reducing IIs. Polyhedral analysis is also popular for dependence analysis [14] for affine memory accesses. Initial work by Liu *et al.* uses polyhedral analysis for memory optimization [15]. Morvan *et al.* use polyhedral analysis and propose a method to improve pipelining of nested loops [14].

Their work first flattens the nested loop and insert pipeline stalls to resolve the memory conflicts. Their work also requires no inter-iteration dependence in the innermost loop. There are also other polyhedral analysis-based techniques used for optimizing static loop pipelines [16, 17, 18] or adding dynamic mechanisms into static loop pipelining for better performance [3, 19, 20, 21].

Formal verification has been commonly used in both software and hardware. There has been recent interest in using formal verification for loop optimisation. Cheng *et al.* propose a Boogie-based approach to capture the correlation between the dependence distance and iteration latency to reduce the II of a loop [22]. Boogie [23] is an automated program verifier on top of satisfiability modulo theories (SMT) solvers. has its own intermediate verification language to describe the behaviour of a program to be verified, which can be automatically translated into SMT queries. An SMT solver then reasons about program behaviour including the values that its variables may take.

All these works use static analysis to optimise static loop pipelining, while our approach optimises dynamic loop pipelining. Unlike static pipelining, our analysis does not include resource constraints. We also neglect the iteration latency of the operations as they are well-handled by the dynamically scheduled hardware.

IV. Methodology

In this section, we first formalise the restriction caused by LSQs in dynamic pipelining. Second, we show how to generate a Boogie program for safely parallelizing the control flow of a nested loop. Then we illustrate the design of our proposed loop scheduler. Finally, we demonstrate our tool flow integrated into the open-sourced HLS tool Dynamatic as a prototype.

A. Problem Formulation

We first formalise the behaviour of a sequential control flow. Let $x \prec y$ denote that x begins execution at a time less than the time y begins execution. An intermediate representation (IR) of a program, such as LLVM IR, usually contains data flow and control flow. A data flow graph indicates the dependences between operations. A control flow graph indicates the dependences between BBs. Since we only consider control flow, our analysis targets the execution of BBs. For a given program and its inputs, we define the following terms:

- $B = \{b_1, b_2, b_3, ...\}$: The set of all the BBs in the program.
- $O : b_{1,1} \prec b_{2,1} \prec b_{1,2} \prec ...$: The original program order of BB execution. $b_{i,j}$ represents the jth iteration of b_i.
- $M_b = \{m_{b,1}, m_{b,2}, ...\}$: The set of all the memory operations within a BB b.
- $o_b : m_{b,1} \prec m_{b,2} \prec m_{b,3} \prec ...$: The original program order of memory operation execution within a BB b.

Combining these O and o_b lexicographically, we can obtain the original program order of memory operations.

If a memory operation is executed outside the program order, it must not break a dependence. A LSQ retains the original order of BB execution for the *start time* of each BB while allowing statements inside the BBs to execute concurrently once started, while reordering memory operations by checking dependences with preceding operations within any currently executing BB at run time.

Let $x \preceq y$ denote that x begins execution at a time no greater than the time y begins execution. For a $O : ... \prec b \prec b' \prec ...$, if it is proven that any memory access in a BB execution b cannot have dependence with any memory access in its successor BB execution b', then the order $O' : ... \prec b' \preceq b \prec ...$ is memory-legal. This means that the LSQ can allocate either b or b' first without breaking any dependence, and b' does not need to be stalled by b to satisfy O.

Let T be the set of all the memory-legal orders of BBs, where the original program order $O \in T$. The hardware generated by vanilla Dynamatic always executes in O. The problems solved in this paper are: 1) how to determine a set of memory-legal orders of BBs, $F \subseteq T$, that execute with high performance, and 2) how to efficiently synthesize hardware that execute in an dynamic order of BBs, O', where $O' \in F$.

B. Determining C-Slow Depth

C-slow pipelining is amenable for improving the throughput when 1) a hardware design that has an II of greater than 1; and 2) it allows out-of-order execution of control flow. To simplify the problem, we restrict the scope of our work to nested loops. First, IIs of greater than 1 are commonly seen in HLS loops. Second, the behaviour of the outer loop of an inner loop helps static dependence analysis between inputs to the inner loop.

For example, a nested loop has a outer loop and an inner loop. Let the execution of the jth iteration of its inner loop in the ith iteration of its outer loop be $e_{i,j}$. Let trip count of the inner loop and outer loop be N_i and M, where N_i may vary in each outer-loop iteration. The original program order of the iterations in the nested loop is shown as follows:

$$E : e_{0,0} \prec e_{0,1} \prec e_{0,2} \prec ... \prec e_{0,N_i-1} \prec e_{1,0} \prec ... \qquad (1)$$

The execution order of a C-slowed loop starts as follows:

$$E' : e_{0,0} \prec e_{1,0} \prec ... \prec e_{C-1,0} \prec e_{0,1} \prec ... \qquad (2)$$

$$C = \min(P, M) \qquad (3)$$

P is the maximum number of outer-loop iterations that can execute in parallel while satisfying both resource constraints and dependence constraints. The constraint for a C-slowed nested loop is that there are always *at most* C outer-loop iterations executing concurrently.

The dependence constraint must not break under the C-slow condition. At the same loop level, E' still holds the same dependence constraints as E:

$$\forall_{0 \le i < M-1} . \forall_{0 \le j < N_i} . e_{i,j} \prec e_{i+1,j} \qquad (4)$$

$$\forall_{0 \le i < M} . \forall_{0 \le j < N_i - 1} . e_{i,j} \prec e_{i,j+1} \qquad (5)$$

The above means that each level of the inner loop still has its iterations starting in strict program order. This means that the memory dependences inside each inner loop instance are still captured by the LSQ. Let $D(e_{i,j_0}, e_{i+c,j_1})$ be the dependence

```
1  procedure pickOneMemoryAccessFromLoop() returns (
2  valid: bool, stmt: int, addr: Index, array: Array,
3  iteration:Index, type: MemoryType) {
4    loop_0: for (int i = 0; i < N; i++) {
5      // s = a[f(i)];
6      if (*) {
7        valid := true; stmt := 0; addr := (f(i));
8        array := a; iteration := (i); type := LOAD;
9        return; }
10     loop_1: for (int j = 0; j < g(i); j++)
11       // s = f(s, b[i][j]);
12       if (*) {
13         valid := true; stmt := 1; addr := (i, j);
14         array := b; iteration := (i); type := LOAD;
15         return; }
16     // a[h(i)] = s;
17     if (*) {
18       valid := true; stmt := 2; addr := (h(i));
19       array := a; iteration := (i); type := STORE;
20       return; }
21   }
22   valid := false;
23   return;
24 }
```

(a) Procedure that arbitrarily picks a memory access.

```
1  // C : Given dependence distance
2  procedure main(C: int) {
3    // assume that all the arrays have arbitrary values
4    havoc a, b;
5
6    // valid: whether the returned memory access is valid
7    // stmt: which statement that executes the memory access
8    // addr: which address the memory access touches
9    // array: which array the memory access touches
10   // iteration: the iteration index of the current outer-loops
11   // type: the type of the memory access, either load or store
12   call valid_0, stmt_0, addr_0, array_0, iteration_0, type_0 :=
         pickOneMemoryAccessFromLoop();
13   call valid_1, stmt_1, addr_1, array_1, iteration_1, type_1 :=
         pickOneMemoryAccessFromLoop();
14
15   assert !valid_0 || !valid_1 ||
16          array_0 != array_1 ||
17          stmt_0 == stmt_1 ||
18          (type_0 == LOAD && type_1 == LOAD) ||
19          iteration_0 >= iteration_1 ||
20          getDistance(iteration_0, iteration_1) >= C ||
21          addr_0 != addr_1;
22 }
```

(b) Main procedure that describes absent dependence for a given `C`.

Fig. 4: A Boogie program generated for the example in Fig. 2. It tries to prove the absence of memory dependence between any two outer-loop iterations with a distance less than `C`.

set between two loop iterations. The dependence condition that needs to be proved is that:

$$\forall_{0\leq i<M-C}.\forall_{0\leq j_0,j_1<N_i}.\forall_{0<c<C}.D(e_{i,j_0},e_{i+c,j_1})=\emptyset \quad (6)$$

In C-slow pipelining, any two concurrently executing inputs of the innermost loop must be independent. With a constraint C, that means any two outer-loop iterations that have a distance less than C must have no dependence.

The dependence constraint above is equivalent to that the minimum dependence distance of the outer loop is less than C. A loop-carried data dependence always has a dependence distance of 1, therefore, we only need to analyse memory dependences. Our tool automatically generates a Boogie program to describe the memory behaviour of the nested loop and calls Boogie verifier to prove the absence of memory dependence within a given distance. Boogie is a verification language, which has its own structs [23]. The ones used in this paper are as follows:

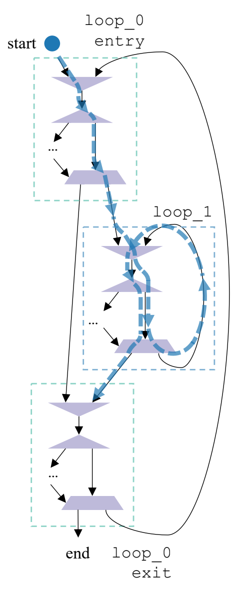
1) `if (*) {A} else {B}` arbitrarily does `A` or `B`.
2) `havoc x` assigns an arbitrary values to a variable or an array `x`. It tells the verifier to prove the condition under all the possible values of `x`.
3) `assert c` proves the condition `c` for all the values that the variables in `c` may take.

For example, Fig. 4 illustrates the Boogie program generated for the motivating example in Fig. 2. It tries to prove the absence of memory dependence between any two outer-loop iterations with a distance less than C, which mainly includes two parts. The Boogie procedure in Fig. 4a arbitrarily picks a memory access from the nested loop during the whole execution and returns its parameters. The returned parameters include the label of the statement being executed, the array and address of the accessed memory, the iteration index of the outer loop and the type of the memory access. Detailed definitions of these parameters are listed at line 6-11 in Fig. 4b.

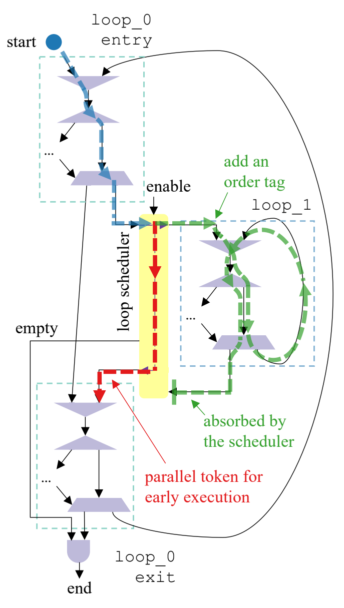
In Fig. 4a, the `for` loop structures in Boogie are directly generated by the automated tool named EASY [24]. In the loop body, each memory access is replaced with an `if(*)` statement. The `if(*)` statement arbitrarily chooses to return the parameters of the current memory access or continue. The procedure is then able to capture *all* the memory access that may execute during the whole execution. If all these memory accesses are skipped, the procedure exits at line 23 with a false `valid` bit, indicating that the returned parameters are invalid.

Fig. 4b describes the main Boogie procedure for dependence analysis. It takes a given C as an input. At line 4, it assumes that arrays `a` and `b` hold arbitrary values. This makes the verification results independent from the program inputs. Lines 12 and 13 arbitrarily pick two memory accesses from the nested loop using the procedure in Fig. 4a.

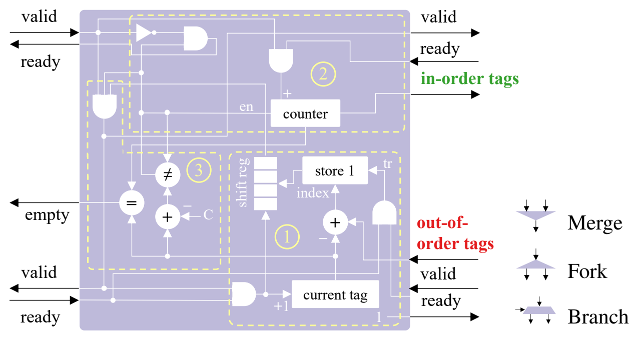
The assertion at line 15 proves the dependence constraint as shown in Eq. 6 for a given C. First, the picked two memory accesses from line 12 and 13 must hold valid parameters (line 15). Second, two accesses touching different arrays cannot have dependence (line 16). Two accesses executed by the same statement is safe (line 17), where the dependence is captured by the LSQ as proved in Eq. 4 and Eq. 5. Two loads cannot have dependence (line 18). Two returned memory accesses are arbitrary and have no difference. Here we assume the memory access with index 0 executes in an earlier outer-loop iteration than the one with index 1 (line 19). In the C-slow pipelining formulation, only the outer-loop iterations with an iteration distance less than C can execute concurrently (line 20). Any two memory accesses that exclude the cases at line 15-20 cannot access the same address. The assertion must hold for *any* two memory accesses for any input values. The Boogie



(a) Default CFG.



(b) Transformed CFG.



(c) Loop Scheduler

Fig. 5: Our tool considers each instance of the innermost loop as a thread and achieves the schedule in Fig. 2c. The dashed arrows represent the token transition in control flow. The scheduler tags the control tokens in the innermost loop to reorder them at the output after out-of-order execution.

verifier automatically verifies whether the assertion always holds for a given C. If the assertion always holds, then it is safe to parallelize C iterations of the outer loop.

Our tool checks C from 2 and increments C if the assertion holds. For nested loops with a larger depth, our tool starts from the innermost loop, taking the second innermost loop as the outer loop, and repeats for the upper level of the loop until the outermost loop is reached. Our experiment results show that only C-slowing the innermost loop with a C less than 10 is effective enough for improving the throughput, where most of empty pipeline slots can be filled. Our tool sets the maximum C to 10 and only analyses each innermost loop by default, providing these parameters as user options. Overall the search time is neglectable compared to the total compilation time of the whole HLS tool flow.

Besides the dependence constraint, a resource constraint of C-slow pipelining is that each path must be able to hold at least C sets of data. The hardware transformation pass in our tool inserts a FIFO with a depth of C in each cycle of the inner loop to enable holding at least C tokens.

C. Inserting Loop Scheduler

Once the value of C for a loop is determined, our tool inserts a component named *loop scheduler* between the entry and exit of each loop. Each C is used as a parameter of the corresponding loop scheduler. The loop scheduler dynamically schedules the control flow and ensures that at most C iterations can execute concurrently. Any outermost loop or unverified loop has its loop scheduler holding $C = 1$ and executes control flow sequentially.

Fig. 5 shows the proposed loop scheduler integrated into a dynamically scheduled control flow graph. Dynamatic uses three kinds of components for control flow, as listed at the bottom right of Fig. 5: a merge arbitrarily takes one of its inputs and sends to the output; a fork replicates its single input and sends to all the outputs; and a branch sends the data input to one of its outputs based on its select input.

For example, the control flow graph of the code in Fig. 2 from vanilla Dynamatic is shown in Fig. 5a. Each dotted block represents a BB. The top BB represents the entry control of the outer loop, which starts the outer-iteration and decides whether to execute the inner loop. The middle BB represents the control of the inner loop. The bottom BB represents the exit control of the outer loop, which decides whether to exit the outer loop.

In each BB, a merge is used to accept a control token that triggers the start of current BB execution. Then a fork is used to produce other tokens to trigger all the data operation inside this BB, hidden in the ellipsis. The control token flows through the fork to a branch. The branch decides the next BB to trigger based on the BB condition. The control flow in Fig. 2 follows the following steps:

1) A control token enters the top BB to start;
2) The token goes through the top BB and enters the middle BB to start the inner loop.
3) The token circulates in the middle BB through the back edge until the exit condition met.
4) The token exits the middle BB and enters the bottom BB. It either goes back to the top BB to repeat 2) or exit, depending on the exit condition.

The control flow is sequential as there is always at most one control token in the control path. Fig. 5b shows the control flow graph with the proposed loop scheduler integrated into the inner loop. The loop scheduler for the outer loop has C of 1 and is neglected for simplicity. The control flow is then:

1) A control token t_1 enters the top BB to start.
2) t_1 goes through the top BB and enters the middle BB with a tag added by the loop scheduler. The loop scheduler checks if there are fewer than C(=3 in this example) tokens in the inner loop. If yes, it immediately produces another token t_2 and sends it to the bottom BB to execute the control flow early (indicated as the red dashed arrow).

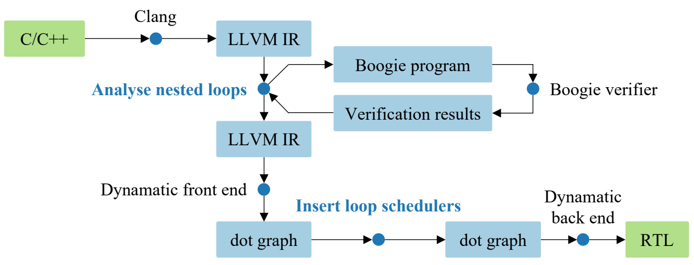


Fig. 6: Our work integrated into Dynamatic. Our contributions are highlighted in bold blue text.

3) t_1 circulates in the middle BB. t_2 goes to the top BB and enters the middle BB with another tag. The loop scheduler produces another token t_3 and sends it to the bottom BB.
4) The above repeats and t_4 is produced. t_1, t_2 and t_3 are all circulating in the middle BB.
5) t_4 reaches the branch in the top BB but *blocked* by the loop scheduler until one token exits the middle BB and consumed by the loop scheduler.
6) The above repeats until the last token exits the bottom BB. An AND gate is inserted at the exit of the bottom BB to synchronise the control flow. It requires a token from the exit of the nested loop and there is no token remained in the inner loop.

The design of the loop scheduler is shown in Fig. 5c. It guards the entry and exit of the inner loop. In the scheduler, a counter is used to count the number of executing control tokens in the inner loop. Based on the value of the counter and the specified C, it decides whether to accept the token from the outer loop and replicates a token to the output to the outer loop for early execution of the next outer-loop iteration. The input token from the exit of the inner loop decrements the counter value by one, allowing the next token from the outer loop to enter the inner loop. An empty bit is used to indicate whether there is no control token in the inner loop.

D. Tool Flow

We integrate our work into Dynamatic for prototyping, as shown in Fig. 6. First, the input C/C++ program is lowered into LLVM IR. Our analysis in LLVM passes searches for C for each innermost loop using Boogie verifier. A control data flow graph is then generated in the form of a dot graph by the front-end of Dynamatic. Our transformation pass inserts loop schedulers and FIFOs into the dot graph. Finally, the transformed graph is translated to RTL code by the back-end of Dynamatic as the final design. Our work can also be integrated into other HLS tools, such as CIRCT [4].

V. Experiments

We evaluate our work on a set of benchmarks, comparing with the designs using Xilinx Vivado HLS and Dynamatic in total circuit area and wall clock time. The total clock cycles were obtained using Vivado XSIM simulator, and the

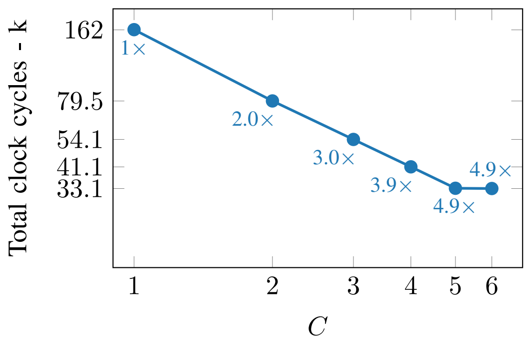


Fig. 7: Speedup by varying C. $C > 6$ breaks the memory dependence in the outer loop. Increasing C initial improves the throughput. However, once all the empty slots are filled, further increasing C has less affect on the throughput.

area results were obtained from the post P&R report in Vivado. The FPGA device we used for result measurements is xc7z020clg484. The version of Xilinx software is 2019.2.

A. Benchmarks

Most HLS benchmarks such as Polybench [25] and CHStone [26], tend to be tailored to what HLS tools already comfortably handle. In order to show how our work pushes the limits of HLS, we use an open-sourced benchmark set from [7], which have been used to evaluate dynamic scheduled HLS. We use our motivating example and select seven benchmarks in the benchmark set that are applicable to our approach for experiments, listed as follows.

triangleVecAccum is the motivating example which uses `f(i) = i`, `g(a, b) = a+b` and `h(i) = i*i+7`.
doitgenTriple is a weighted version of multi-resolution analysis kernel (MADNESS).
correlation computes the correlation matrix.
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.
gramSchmidt is a Gram-Schmidt decomposition.

B. Results

We first take the motivating example as a case study and then discuss the overall results for all the benchmarks. Fig. 7 shows the total clock cycles of the hardware for the motivating example with different C. Only $C < 6$ for this example does not break memory dependence. When C increases initially, more outer-loop iterations are parallelized, significantly improving the throughput. However, when C reaches 5, further increasing C does not affect the throughput. That is because almost all the empty pipeline slots have been filled. The overhead caused by $C = 6$ only adds additional one depth into the FIFOs. With a small C, the area overhead caused by the FIFO depths is neglectable for most of applications.

The speedups of our designs compared to the baselines over all the benchmarks are shown in Fig. 8, and the detailed

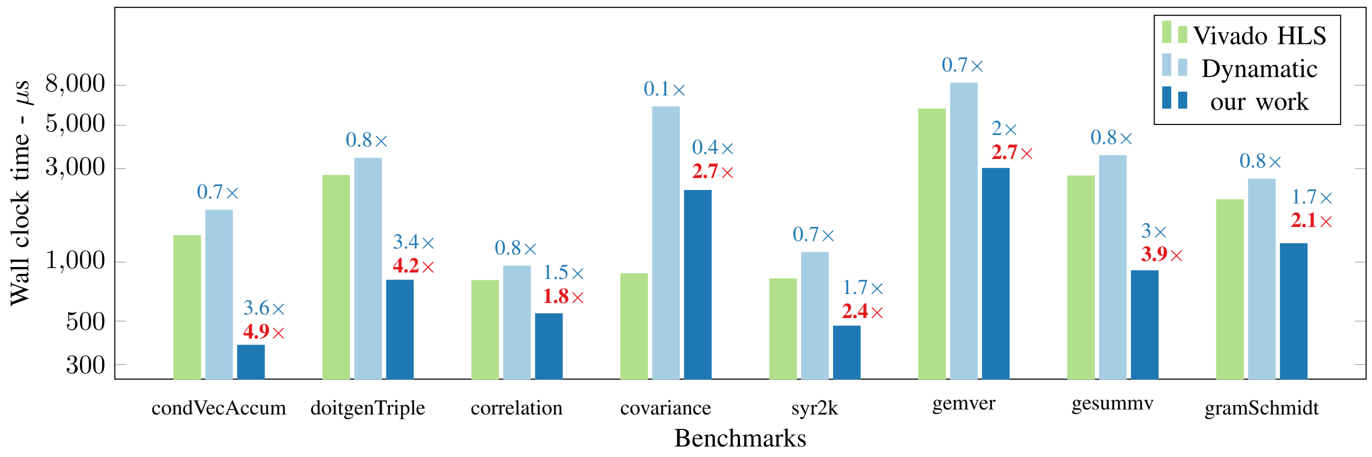


Fig. 8: Performance comparison of the designs from our tool with the designs from Vivado HLS and Dynamatic. The speedups shown in blue and red take the design from Vivado HLS and Dynamatic as the baselines respectively.

TABLE I: Evaluation of our approach on a set of benchmarks. vhls = Vivado HLS. base = vanilla Dynamatic. ours = our work.

Benchmarks	LUTs - k				DSPs				Registers - k				Cycles - k				Fmax - MHz				Wall clock time - ms			
	vhls	base	ours	×	vhls	base	ours	×	vhls	base	ours	×	vhls	base	ours	×	vhls	base	ours	×	vhls	base	ours	×
triangleVecAccum	0.311	18.8	19.4	1.03	5	5	5	1	0.295	4.92	5.57	1.13	165	162	33.2	0.20	121	88.2	88.3	1	1.37	1.84	0.376	0.20
doitgenTriple	0.728	21.8	22	1.01	5	20	20	1	0.715	8.18	8.47	1.04	335	297	67.3	0.23	121	87.6	83.3	0.95	2.77	3.39	0.808	0.24
correlation	1.94	14.6	14.8	1.01	5	36	36	1	1.78	11.8	11.8	1	97.2	90.1	41.7	0.46	121	94.4	76.5	0.81	0.805	0.955	0.545	0.57
covariance	1.55	7.44	8.96	1.20	5	9	9	1	1.25	6.83	9.4	1.38	105	89.8	33.6	0.37	121	14.5	14.5	1	0.872	6.2	2.32	0.37
syr2k	0.603	4.04	4.57	1.13	5	19	19	1	0.693	3.97	4.44	1.12	99.3	82.4	34.3	0.42	121	73.6	72.9	0.99	0.822	1.12	0.471	0.42
gemver	1.44	8.33	8.72	1.05	5	28	28	1	1.24	8.4	8.29	0.99	730	719	227	0.32	121	87.7	75.3	0.86	6.04	8.2	3.01	0.37
gesummv	0.612	3.48	3.76	1.08	5	18	18	1	0.701	3.66	4.18	1.14	333	327	81.9	0.25	121	93.5	90.6	0.97	2.75	3.5	0.903	0.26
gramSchmidt	1.66	47.4	47.9	1.01	5	30	30	1	1.58	16.3	16.9	1.04	238	142	57.9	0.41	114	53.6	46.6	0.87	2.08	2.65	1.24	0.47
Geom. mean				**1.07**				1				**1.1**				**0.32**				**0.94**				**0.35**

results are shown in Tab. I. Overall, the circuits from vanilla Dynamatic are slightly slower than Vivado HLS even though they have better throughput. It is because that the academic buffering tool in Dynamatic cannot perform retiming as good as the commercialised Vivado HLS. This leads to a lower maximum clock frequency, especially for `covariance` and `gramSchmidt`. C-slow pipelining significantly improves the throughput of dynamically scheduled circuits, leading to the best performance in the figure for most benchmarks.

The actual speedup is less than the expected speedup based on the value C due to resource constraints. For instance, there are multiple memory accesses to the same array in one iteration, and the memory blocks have limited bandwidth. These memory accesses are then stalled and serialised by the internal memory arbiter, resulting in additional pipeline stalls. Such performance overhead could be reduced when array partitioning is applied. Even though, our tool achieves at least 1.75× speedup from vanilla Dynamatic. Additionally, benchmarks such as `correlation` and `gramSchmidt` contain a only small region of code that can be C-slow pipelined, resulting in less performance improvement. The maximum clock frequency decreases slightly after C-slow pipelining because the insertion of non-transparent FIFOs (latency = 0 cycle) increases the critical path of the circuit.

The average area overhead of 7% in LUTs and 10% in registers is caused by the additional loop schedulers and FIFOs. DSPs are not changed as they are used for floating-point operators. The area overhead caused by our approach is relatively small compared to the performance gain.

VI. Conclusions

The existing dynamically scheduled HLS tool that requires sequential execution of control flow to retain memory dependence for high performance out-of-order memory accesses. This causes significant performance overhead when pipelining nested loops that contain deeply pipelined inner loops.

We propose an automated approach to lift the restriction and efficiently improve the performance of the hardware. We show how to statically prove the absence of dependence between iterations of the outer loop, and apply C-slow pipelining to the inner loop. We treat each instance of the inner loop as a thread and efficiently parallelize them at run time. The performance gain is significant with little area overhead. Our future work is to explore C-slow retiming in this framework.

Acknowledgements

This work is supported by the EPSRC (EP/P010040/1, EP/R006865/1). The authors wish to thank Lana Josipović for helpful comments.

References

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

[2] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo sdc scheduling with recurrence minimization in high-level synthesis," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[3] 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.

[4] C. contributors, "Circt: Circuit ir compilers and tools," https://github.com/llvm/circt/tree/main/, 2021.

[5] 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.

[6] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming (preliminary version)," in *Third Caltech conference on very large scale integration*. Springer, 1983, pp. 87–116.

[7] J. Cheng, J. Wickerson, and G. A. Constantinides, "Finding and finessing static islands in dynamically scheduled circuits," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3490422.3502362

[8] 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.

[9] Celoxica, "Handel-C," 2005. [Online]. Available: http://www.celoxica.com

[10] 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.

[11] 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.

[12] Y. Markovskiy and Y. Patel, "Simple symmetric multi-threading in xilinx fpgas," 2002.

[13] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-placement c-slow retiming for the xilinx virtex fpga," in *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 185–194. [Online]. Available: https://doi.org/10.1145/611817.611845

[14] A. Morvan, S. Derrien, and P. Quinton, "Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 3, pp. 339–352, 2013.

[15] 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.

[16] S. Dai, Mingxing Tan, Kecheng Hao, and Z. Zhang, "Flushing-enabled loop pipelining for high-level synthesis," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[17] G. Dimitriou, M. Dossis, and G. Stamoulis, "Operation dependencies in loop pipelining for high-level synthesis," in *2018 South-Eastern European Design Automation, Computer Engineering, Computer Networks and Society Media Conference (SEEDA_CECNSM)*, 2018, pp. 1–6.

[18] M. Smelyanskiy, S. A. Mahlke, E. S. Davidson, and H. . S. Lee, "Predicate-aware scheduling: a technique for reducing resource constraints," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003, pp. 169–178.

[19] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[20] S. Dai, G. Liu, R. Zhao, and Z. Zhang, "Enabling adaptive loop pipelining in high-level synthesis," in *2017 51st Asilomar Conference on Signals, Systems, and Computers*, 2017, pp. 131–135.

[21] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1802–1815, 2018.

[22] J. Cheng, J. Wickerson, and G. A. Constantinides, "Exploiting the correlation between dependence distance and latency in loop pipelining for hls," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 341–346.

[23] 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/

[24] 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. 1–1, 2021.

[25] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, vol. 437, 2012.

[26] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and

K. Ishii, “Chstone: A benchmark program suite for practical c-based high-level synthesis,” in *2008 IEEE International Symposium on Circuits and Systems*, 2008, pp. 1192–1195.