Parallelising Control Flow in Dynamic-Scheduling High-Level Synthesis

JIANYI CHENG, Imperial College London, United Kingdom
LANA JOSIPOVIĆ, ETH Zürich, Switzerland
JOHN WICKERSON, Imperial College London, United Kingdom
GEORGE A. CONSTANTINIDES, Imperial College London, United Kingdom

Recently, there is a trend to use high-level synthesis (HLS) tools to generate dynamically scheduled hardware. The generated hardware is made up of components connected using handshake signals. These handshake signals schedule the components at run time when inputs become available. Such approaches promise superior performance on 'irregular' source programs, such as those whose control flow depends on input data. This is 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 under-explored, due to the complexity in run-time control flows and memory dependencies. Existing tools allow some of the operations of different BBs to overlap, but in order to simplify the analysis required at compile time they require the BBs to *start* in strict program order, thus limiting the achievable parallelism and overall performance.

We formulate a general dependency model suitable for comparing the ability of different dynamic scheduling approaches to extract maximal parallelism at run-time. Using this model, we explore a variety of mechanisms for run-time scheduling, incorporating and generalising existing approaches. In particular, we precisely identify the restrictions in existing scheduling implementation and define possible optimisation solutions. We identify two particularly promising examples where the compile-time overhead is small and the area overhead is minimal and yet we are able to significantly speed-up execution time: (1) parallelising consecutive independent loops; and (2) parallelising independent inner-loop instances in a nested loop as individual threads. Using benchmark sets from related works, we compare our proposed toolflow against a state-of-the-art dynamic-scheduling HLS tool called Dynamatic. Our results show that on average, our toolflow yields a 4× speedup from (1) and a 2.9× speedup from (2), with a negligible area overhead. This increases to a 14.3× average speedup when combining (1) and (2).

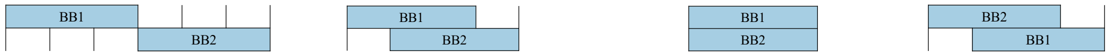
CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; **Modeling and parameter extraction**.

Additional Key Words and Phrases: FPGA, High-Level Synthesis, Dynamic Scheduling, Static Analysis

ACM Reference Format:
Jianyi Cheng, Lana Josipović, John Wickerson, and George A. Constantinides. 2023. Parallelising Control Flow in Dynamic-Scheduling High-Level Synthesis. In . ACM, New York, NY, USA, Article 1, 32 pages. https://doi.org/10.1145/3599973

1 INTRODUCTION

FPGAs are now widely used as a reconfigurable device for custom high-performance computing, such as in datacentres including Microsoft Project Catapult [42] and Amazon EC2 F1 instances [1]. However, users need to understand low-level hardware details for directly programming on FPGAs. In order to lift this restriction for software engineers, high-level synthesis (HLS) tools automatically translate a software program in a high-level software language, such as C, into low-level hardware descriptions. This could also significantly reduce the design effort compared to manual register transfer level (RTL) implementation. Today various HLS tools have been developed in both academia, including LegUp from The University of Toronto [7], Bambu from the Politecnico di Milano [9] and Dynamatic from EPFL [32], and



(a) Sequential execution. (b) Sequentially-started execution. (c) Simultaneous execution. (d) Out-of-order execution.

Fig. 1. Basic block schedules. Assume BB1 executes before BB2 in the original program order. Dynamatic only supports (a) and (b). We show how to support (c) and (d) using analysis in our proposed model.

industry, including Intel HLS compiler [30], Xilinx Vivado HLS [51], Cadence Stratus HLS [46] and Siemens Catapult HLS [10].

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 conservative schedule produced by static analysis.

Dynamic-scheduling HLS tools, such as Dynamatic [32], 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 dependencies in the source program. There are two kinds of dependencies: memory dependencies (*i.e.* dependency via a memory location) and data dependencies (*i.e.* dependency via a program variable). There are also two scopes of dependency: between instructions in the same basic block (BB), and between instructions in different BBs. This leads to four cases to consider:

(1) Intra-BB data dependencies: these can be respected by placing handshaking connections between the corresponding hardware operations in the circuit.
(2) Intra-BB memory dependencies: these can be kept in the original program order using elastic components named *load-store queues* (LSQs) [31]. An LSQ is a hardware component that schedules memory operations at run time.
(3) Inter-BB data dependencies: 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 [33].
(4) Inter-BB memory dependencies: these can be respected by starting BBs in strict program order and using an LSQ, as in (2).

In all cases, existing dynamic-scheduling HLS tools well exploit parallelism for cases (1) and (2) above. This allows out-of-order execution *within* a BB, but requires 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.

The existing dependency model only analyses intra-BB dependencies, which exploits parallelism at the data flow level. The inter-BB dependencies are resolved by maintaining the original program order. The order is preserved by sequentially starting BB execution even though these BBs are executing in parallel, as shown in Fig. 1b. Analysis for inter-BB dependencies is limited. In this article, we propose a dependency model that formalises all four cases above. As a demonstration of applications, we use the dependency model to focus on further exploiting parallelism for cases (3) and (4). We find BBs that can be started out-of-order or simultaneously, as shown in Fig. 1c and Fig. 1d, and use static analysis (powered by the Microsoft Boogie verification engine [35]) to ensure that inter-BB dependencies are still respected. We then achieve a parallel BB schedule by two techniques: 1) parallelising independent consecutive loops (inter-block scheduling); and 2) parallelising independent consecutive inner-loop instances in a nested loop (C-slow pipelining). Our main contributions include:

- a general dependency model that formalises both data flow dependencies and control flow dependencies for dynamic-scheduling high-level synthesis;
- a technique that automatically identifies the absence of dependencies between consecutive loops using the Microsoft Boogie verifier for parallelism;
- a technique that automatically identifies the absence of dependencies between consecutive iterations of a loop using the Microsoft Boogie for C-slow pipelining; and
- results and analysis showing that our techniques, compared to original Dynamatic, achieve on average 14.3× speedup with 10% area overhead.

The rest of our article is organised as follows: Section 2 introduces existing works on dynamic-scheduling HLS, parallelising control-flow graphs (CFGs) for HLS, and C-slow pipelining. Section 3 explains our general dependency model for dynamic-scheduling HLS. Section 4 demonstrates inter-block scheduling as an application of the proposed dependency model. Section 5 demonstrates C-slow pipelining as another application of the proposed dependency model. Section 7 evaluates the effectiveness of these two techniques.

2 BACKGROUND

This section first reviews related work on existing HLS tools that use dynamic scheduling. We then compare existing works on static analysis of memory dependencies for HLS with our work. Finally, we review related works on parallelising CFGs for HLS and C-slow pipelining on FPGAs.

2.1 Dynamic-Scheduling High-Level Synthesis

Most HLS tools such as Xilinx Vivado HLS [51] and Dynamatic [32] 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 [22]. 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 dependency. The CDFG is used as part of the dependency constraints in both static and dynamic scheduling [20, 32].

In dynamic-scheduling HLS, initial work was studied by Page and Luk [44], 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 and synchronous worlds. In the asynchronous world, Venkataramani *et al.* [48] propose a toolflow that maps ANSI-C programs into asynchronous hardware designs. Li *et al.* [38] propose a dynamic-scheduling HLS tool named Fluid that supports the synthesis of complex CFGs into asynchronous hardware designs. In the synchronous world, Sayuri and Nagisa [43] propose a method that synthesises single-level loops into dynamically scheduled circuits. Josipović *et al.* [32] propose an open-sourced HLS tool named 'Dynamatic' that automatically translates a program into a dynamically pipelined hardware.

Dynamatic uses pre-defined components with handshake connections formalised by Carloni *et al.* [8]. 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 dependency is controlled by load-store queues (LSQs). An LSQ exploits out-of-order memory accesses by checking memory dependency in program order at run time [31] 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 dependencies 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 dependency, such that the synthesised hardware is still correct.

2.2 Parallelising Control Flows for HLS

Dependency analysis for parallelising a CFG of a sequential program has been well-studied in the software compiler world [28]. Traditional approaches exploit BB parallelism using polyhedral analysers such as Pluto [3] and Polly [25]. These tools automatically parallelise code that contains affine memory accesses [4, 23] and have been widely used in HLS to parallelise hardware kernels [39, 40, 49, 55]. 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 [52] and video rendering [47].

Recently, there are works that use formal verification to prove the absence of dependency to exploit hardware parallelism. Compared with affine or polyhedral analysis, formal verification can analyse non-affine memory accesses but takes a longer time. Zhou *et al.* [54] propose a satisfiability-modulo theory (SMT)-based [24] approach to verify the 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 [13]. Microsoft Boogie [35] is an automated program verifier on top of SMT solvers. The Boogie verifier does not run a Boogie program but generates a set of specifications for verification. 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. Boogie has its own constructs, and here we list the ones used in this article:

(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.

In this article, we use Boogie to verify the absence of memory dependency between different iterations of the same loop or two consecutive loops. Our approach can generate Boogie programs from arbitrary programs based on the formulation by [19].

Mapping a parallel BB schedule into hardware has also been widely studied. Initial work by Cabrera *et al.* [5] proposes an OpenMP extension to off-load computation to an FPGA. Leow *et al.* [37] propose a framework that maps OpenMP code in Handel-C [11] to VHDL programs. Choi *et al.* [18] propose a plugin that synthesises both OpenMP and Pthreads C programs into multi-threaded hardware, used in an open-sourced HLS tool named LegUp [7]. Gupta *et al.* propose an HLS tool named SPARK that parallelises control flow with speculation [26]. Existing commercialised HLS tools [10, 29, 46, 51] support multi-threaded hardware synthesis using manually annotated directives by users. These works either require user annotation or only use static scheduling, while our approach only uses automated dynamic scheduling.

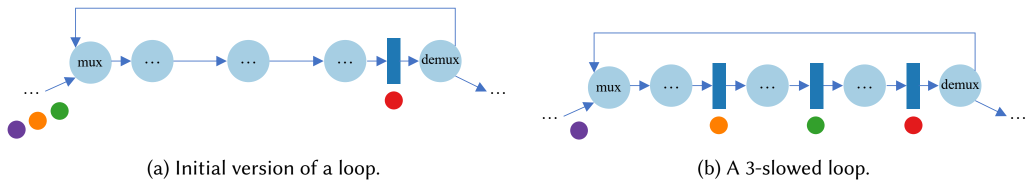


Fig. 2. An example of 3-slowing a loop. The 3-slowed loop has tripled latency, the same throughput and one-third critical path compared to the initial version.

Finally, there are works on simultaneously starting BB in dynamic-scheduling HLS. Cheng *et al.* [14] 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.

2.3 *C*-Slow Pipelining

C-slow pipelining is a technique that replaces each register in the circuit with C registers to construct C independent threads [41]. 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. 2a. We use initiation interval (II) for evaluating the hardware performance. An II of a loop is defined as the time difference in block cycles between the start of the same operation in two consecutive iterations. 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 of other sets. Fig. 2b demonstrates a 3-slowed loop that is functionally equivalent to the one in Fig. 2a. 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. 2a. A C-slowed loop can have a better throughput or clock frequency to achieve approximately C times speedup.

C-slow pipelining was first proposed by Leiserson *et al.* for optimising the critical path of synchronous circuits [36]. Markovskiy and Patel [41] propose a C-slow based-approach to improve the throughput of a microprocessor. Weaver *et al.* [50] 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 the source level to determine C for each loop by checking the dependency between inputs to the loop and then apply hardware transformations to achieve C-slow pipelining.

3 DEPENDENCY MODEL FOR DYNAMIC-SCHEDULING HLS

In this section, we formalise the dependency model for dynamic-scheduling HLS and demonstrate the restriction of the state-of-the-art dynamic-scheduling HLS tool. The dependency formulation for static scheduling has been well-studied [6, 16, 53]. Here we extend the dependency model in [16] to support dynamic scheduling.

3.1 Scheduling Specifications

We first formulate the fundamental specifications of the dependency constraints for scheduling. For two run-time events x and y[1], we introduce the following terms:

- $d(x, y)$ denotes whether executions of x and y have dependencies,
- $x \prec y$ denotes whether x executes before y in strict program order,
- $t(x) \in \mathbb{N}$ denotes the start time of the execution x in clock cycles, and
- $l(x) \in \mathbb{N}$ denotes the latency of the execution x in clock cycles.

The general dependency constraint is listed as follows:

$$\forall x, y.\, d(x, y) \land x \prec y \Rightarrow t(x) + l(x) \leq t(y) \tag{1}$$

where x and y are the operations in the input program. The total latency of the execution is defined as t_T:

$$\forall x.\, t_T \geq t(x) + l(x) \tag{2}$$

The goal is to determine a schedule with a minimum t_T that must not break Constraint 1.

3.2 Dynamatic Implementation

Now we show how the state-of-the-art dynamic-scheduling HLS tool, Dynamatic [32], parallelises the instruction execution dynamically.

3.2.1 Control Flow Graph. We assume that the input program is sequential. As introduced in Section 2.1, the input program is initially lowered from C/C++ to a CFG, where each BB contains instructions in sequential order. A CFG illustrates the control flows of a program. Each vertex represents a BB, and each edge represents a control transition between two BBs. Each of these vertices corresponds to a subgraph at the lower level, known as data flow graph (DFG). Each subgraph vertex represents an operation, and each edge between these vertices represents a data dependency between two operations. For a given program and its inputs, we define the following terms for the input source:

- $B = \{\mathrm{b}_1, \mathrm{b}_2, ...\}$ denotes the set of all the BBs in the program, and
- $I_b = \{\mathrm{i}_1, \mathrm{i}_2, ...\}$ denotes the set of all the instructions within a BB b.

The original execution order of the input program can be defined as follows:

- $E_B \subseteq B \times \mathbb{N}$ denotes the set of execution of BBs, where (b_h, k) denotes execution of BB h (b_h) in its kth iteration,
- $\prec \subseteq E_B \times E_B$ denotes the original program order of BB execution, where $(b_h, k) \prec (b_{h'}, k')$ denotes (b_h, k) executes before $(b_{h'}, k')$ in strict program order,
- $E_I \subseteq \bigcup_{b \in B} I_b \times \mathbb{N}$ denotes the set of execution of instructions, and
- $\prec_b : I_b \times I_b$ denotes the original program order of instruction execution within a BB b.

The execution of BBs $\prec$ can be dynamic, where a BB may have a different number of iterations than another BB. However, inside each BB, the execution of instructions $\prec_b$ is static, where they always have the same number of iterations and execution order inside the BB, as shown in the following constraints.

$$\forall b, i, k.\, (b, k) \in E_B \land i \in I_b \Rightarrow (i, k) \in E_I \tag{3}$$

$$\forall i, i', k.\, (i, k) \in E_I \land (i', k) \in E_I \land i \prec i' \Rightarrow (i, k) \prec (i', k) \tag{4}$$

[1]The run-time event could be the execution of an instruction, a basic block or a loop.

By combining $\prec$ and $\prec_b$ lexicographically, the original program order of the instruction execution can be obtained. This is used as a reference for correctness checks. A schedule being correct is defined as the execution result by this schedule is always the same as the result by sequential execution in the original program order.

3.2.2 Dependency Constraints. There are mainly four types of dependencies to solve as introduced in Section 1. First, the intra-BB data dependencies are represented as edges in DFGs inside BBs and can be directly mapped to *handshake signals* between hardware operations. An operation may have variable latency depending on its inputs. Operations not connected through handshake signals are independent and can execute in parallel or out-of-order. Although an operation may have a variable latency and execute out-of-order, each data path propagates in-order data through the edges as formalised by Carloni *et al.* [8]. Such a preserved data order inside each BB preserves the intra-BB data dependencies.

Second, the intra-BB memory dependencies are dynamically scheduled using LSQs. Dynamatic analyses $\prec_b$ and statically encodes the sequential memory order of each BB into the LSQ. An LSQ allocates the memory operations in a BB into its queue at the start of the corresponding BB execution. It dynamically checks these memory operations following the order of $\prec_b$ and executes a memory operation if it is independent of all its priorly executing operations. The LSQ ensures that the intra-BB memory dependencies are resolved by statically encoding $\prec_b$ into the LSQ for dependency check.

We now need to resolve the inter-BB dependencies. Completely resolving inter-BB dependencies for concurrent execution at run time is still an open question. The most dynamic approach so far is by Dynamatic which enables dynamic scheduling with only one restriction. The restriction requires BB executions must start sequentially in strict program order, even if they can execute in parallel. Let $t : E_I \cup E_B \rightarrow \mathbb{N}$ denote the start times of an instruction execution or a BB execution in clock cycles.

$$\forall i, b, k.\, i \in I_b \land (b, k) \in E_B \Rightarrow t(b, k) \leq t(i, k) \tag{5}$$

$$D : \forall e, e'.\, e \in E_B \land e' \in E_B \land e \prec e' \Rightarrow t(e) < t(e') \tag{6}$$

This preserves the original program order $\prec$ during run-time hardware execution and provides a reference of correctness for dynamic scheduling when combined with $\prec_b$.

Third, with Constraint 6, the inter-BB data dependencies are preserved using muxes. Dynamatic uses a mux to select data input to a BB at the start of the BB by its preceding BB execution. Since the BBs are restricted to start sequentially, there can only be at most one BB receiving at most one starting signal from its multiple preceding BBs. The input data of a BB is selected by the muxes based on the starting signal and accept the correct input data for the computation. The starting signal in the sequential BB execution order ensures that the data sent to each BB is also in strict program order. Such property ensures in-order data flow between BBs, which preserves inter-BB data dependencies.

Finally, the inter-BB memory dependencies are resolved by the LSQ by dynamically allocating memory operations between these BB executions. LSQ dynamically monitor the start signals of BB executions and allocate groups of memory operations in the same BB order, also known as $\prec$. It checks the memory operations in an order that combines $\prec$ and $\prec_b$ lexicographically, the same as the original program order. With the logic that resolves the intra-BB memory dependencies, the LSQ ensures that the out-of-order memory execution always has the same results as the execution in the allocated order *i.e.* the program order. Such an approach of preserving $\prec$ for allocation in the LSQ resolves the inter-BB memory dependencies.

With the implementation above, $\prec_b$ is directly encoded to hardware at compile time, and $\prec$ is recovered at run time. All four kinds of dependencies can be resolved by checking dependencies between operations in strict program order. Let $d : (E_I \cup E_B)^2 \rightarrow \{0, 1\}$ denote whether two executions have dependencies, and let $l : E_I \rightarrow \mathbb{N}$ denote the latencies of instruction execution in clock cycles. Dynamatic ensures the following dependency constraint always holds:

$$\forall e, e'.\ e \in E_I \wedge e' \in E_I \wedge e \prec e' \wedge d(e, e') \Rightarrow t(e) + l(e) \leq t(e') \tag{7}$$

3.2.3 Hardware Restriction. The dependency constraints above ensure the correctness of the schedule with optimised performance. However, the hardware architecture could also bring restrictions on performance.

Here we only consider the case of pipelining. HLS uses hardware pipelining to exploit parallelism among different iterations of a BB using a single hardware instance. A resource restriction is then added to this scheduling model.

$$\forall b, k, k'.\ (b, k) \in E_B \wedge (b, k') \in E_B \wedge k \neq k' \Rightarrow t(b, k) \neq t(b, k') \tag{8}$$

This means that two iterations of the same BB cannot start simultaneously in pipelining, which requires multiple hardware instances. In Dynamatic implementation, Constraint 8 is covered by Constraint 6. We keep it here as Constraint 6 will be relaxed in the later sections. Also, Dynamatic generates a data flow hardware architecture where the input data order is always the same as the output data order for each BB. This could also restrict the execution of each individual operation:

$$\forall i, k, k', b.\ b \in B \wedge i \in I_b \wedge (i, k) \in E_I \wedge (i, k') \in E_I \wedge t(b, k) < t(b, k') \Rightarrow t(i, k) < t(i, k') \tag{9}$$

3.2.4 Summary. In summary, Dynamatic automatically generates efficient dynamically scheduled hardware with minimal static analysis. The generated hardware must satisfy four constraints. First, Constraint 6 ensures the program order is preserved in the hardware design. Second, given Constraint 6, the hardware logic ensures Constraint 7 for dynamically resolving dependencies using the sequential program order. Third, the performance of pipelined architecture is restricted by the resource Constraint 8. Finally, the performance is also restricted by the Constraint 9 introduced from the data flow architecture.

Constraint 6, which restricts BBs to start sequentially, significantly reduces the need for static analysis and simplifies the dynamic scheduling problem. However, this is too conservative for dependency analysis. For instance, when all the BB executions do not have dependencies, Constraint 6 could cause sub-optimal performance. We now demonstrate how Constraint 6 can be relaxed in this model and what analysis can be applied to achieve a schedule with a better performance.

3.3 Possible Relaxations

We seek to relax Constraint 6 and only restrict the order between the BB executions that may have dependencies. If two instruction executions in two BB executions have dependencies, then these two BB executions have dependencies.

$$d((b, k), (b', k')) = (\exists i, i'.\ (i, k) \in E_I \wedge (i', k') \in E_I \Rightarrow d((i, k), (i', k'))) \tag{10}$$

Then Constraint 6 can be relaxed to:

$$D' : \forall e, e'. e \in E_B \wedge e' \in E_B \wedge e \prec e' \wedge d(e, e') \Rightarrow t(e) < t(e') \tag{11}$$

However, analysing the dependency between two BB execution at run time is still challenging.

In order to enable static analysis for dependency analysis, we over-approximate analysing two particular BB executions to all the executions of two particular BBs. This means that each dependency is checked between statements in the source instead of run-time events. Let $d' : B \times B \longrightarrow \{0, 1\}$ denote whether two basic blocks may have dependency during their executions.

$$d'(b, b') = (\exists k, k'. \, d((b, k), (b', k'))) \tag{12}$$

We then relax Constraint 11 to the following:

$$D'' : \forall b, b', k, k'. \, (b, k) \in E_B \wedge (b', k') \in E_B \wedge (b, k) \prec (b', k') \wedge d'(b, b') \Rightarrow t(b, k) < t(b', k') \tag{13}$$

This ensures that only some BBs must start sequentially, and BBs that cannot have dependency during the whole execution can start in parallel or out-of-order. The dependency set of the program from our formulation lies between Constraint 6 and Constraint 11, where $D' \subseteq D'' \subseteq D$. With this new constraint, the dependencies are still respected with existing muxes and LSQs since the dependent BB executions remain starting sequentially.

In the rest of the article, we demonstrate how to apply static analysis for relaxing Constraint 6 towards Constraint 13 and enable two hardware optimisation techniques for dynamic-scheduling HLS.

4 DYNAMIC INTER-BLOCK SCHEDULING

This section demonstrates an application of the proposed dependency model for achieving the simultaneous execution of two independent BBs by parallelising independent sequential loops. It formalises our prior conference paper [15] in the proposed model. Section 4.1 demonstrates a motivating example of parallelising independent sequential loops. Section 4.2 explains how to formulate the problem into the proposed model in Section 3 and shows how to use Microsoft Boogie to automatically determine the absence of dependency between these sequential loops. Section 4.3 illustrates efficient hardware transformation for parallelising sequential loops.

4.1 Motivating Example

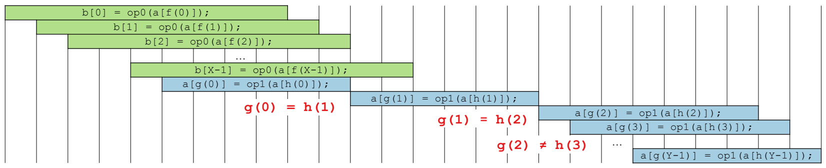
Here we illustrate a motivating example of parallelising two sequential loops in dynamically scheduled hardware. Fig. 3a shows an example of two sequential loops, `loop_0` and `loop_1`. In each iteration of `loop_0`, an element at index `f(i)` of array `a` is loaded and processed by a function `op0`. 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 dependency 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 [32], the state-of-the-art dynamic scheduled HLS tool, synthesises hardware that computes in a schedule shown in Fig. 3b. 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 dependency 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 `g(0)` = `h(1)`, then the first two iterations must be executed sequentially, and if we further suppose that `g(1)` ≠ `h(2)`, then the second and third iterations are pipelined with an II of 1.

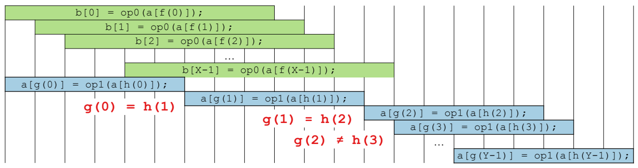
However, `loop_1` is stalled until all the iterations in `loop_0` have started, even though it has no dependency on `loop_0`. The reason is that Dynamatic forces all the BBs to start sequentially to preserve any potential inter-BB

```
// f(i) = 0;
// g(j) = j*j+1;
// h(j) = j;

int a[N], b[M];
void transformVector() {
  loop_0: for (int i = 0; i < X; i++)
    b[i] = op0(a[f(i)]);
  loop_1: for (int j = 0; j < Y; j++)
    a[g(j)] = op1(a[h(j)]);
}
```

(a) Source



(b) Default pipeline schedule.



(c) Parallelised pipeline schedule.

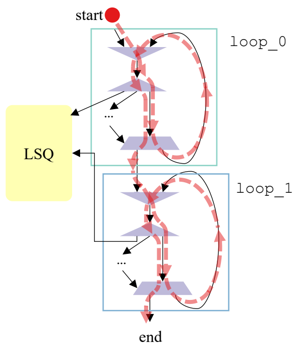
Fig. 3. Motivating example. Assume no dependence between two loops. The dynamically scheduled hardware from the original Dynamatic [32] a schedule in (b). Our work achieves an optimised schedule in (c).

dependency, such as the inter-iteration memory dependency in `loop_1`. For this example, each loop iteration is a single BB, and at most one loop iteration starts in each clock cycle.

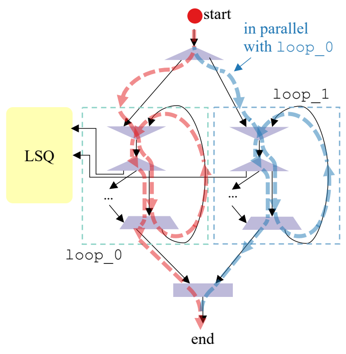
An optimised schedule is shown in Fig. 3c. In the figure, both loops start from the first cycle and iterate in parallel, resulting in better performance. Existing approaches cannot achieve the optimised schedule: static scheduling can start `loop_0` and `loop_1` simultaneously such as using multi-threading in LegUp HLS [7], but `loop_1` is sequential as the static scheduler assumes the worst case of dependency and timing; dynamic scheduling has a better throughput of `loop_1`, but cannot start it simultaneously with `loop_0`.

Besides, determining the absence of dependency between these two loops for complex `f(i)`, `g(j)` and `h(j)` is challenging. In this section, our toolflow 1) generates a Boogie program to formally prove that starting `loop_0` and `loop_1` simultaneously cannot break memory dependency 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. 5).

The transformation for the example in Fig. 3a is demonstrated in Fig. 4a and Fig. 4b. Fig. 4a shows the CFG generated by the original Dynamatic. The CFG consists of a set of pre-defined components, as listed in Table 1. 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. 4b 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



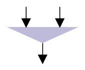
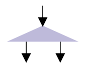
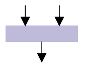
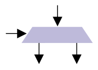
(a) Default CFG.



(b) Parallelised CFG.

Fig. 4. Hardware transformation of the motivating example in Fig. 3.

Table 1. Elastic components for dynamically scheduled HLS.

	**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 same hardware, yet, Fig. 4b 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 Section 4 explains the details of our approach.

4.2 Problem Formulation and Dependency Analysis

Here we first show how to formalise the problem based on the model in Section 3. We then show how to extract sets of subgraphs from a sequential program, where subgraphs in the same set may start in parallel. The absence of dependency between these parallelised subgraphs is formally verified using the generated Boogie program by our tool.

4.2.1 Problem Formulation. 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 the following terms:

- $G = \{g_1, g_2, \ldots\}$ denotes a set of consecutive subgraphs in the CFG of the program,
- $E_G \subseteq G \times \mathbb{N}$ denotes the executions of subgraphs,
- $\prec_G \subseteq E_G \times E_G$ denotes the original program order of subgraph execution, and
- $B_g \subseteq B$ denotes the set of all the BBs in subgraph g.

The subgraph set G must satisfy the following constraints based on the original sequential program order. First, the BB sets of all the subgraphs in G must be disjoint.

$$\forall g, g'. \, g \in G \wedge g' \in G \wedge g \neq g' \Rightarrow B_g \cap B_{g'} = \emptyset \tag{14}$$

Second, no BB outside a subgraph executes during the execution of the subgraph in sequential program order. Let $b_0(e)$ and $b_\mathrm{n}(e)$ denote the first BB execution and the last BB execution in a subgraph execution, where $e \in E_G$.

$$\nexists b, k, g, m. \, (g, m) \in E_G \wedge b \notin B_g \wedge (b, k) \in E_B \Rightarrow b_0(g, m) \prec (b, k) \prec b_\mathrm{n}(g, m) \tag{15}$$

Finally, all the subgraphs in G must consecutively execute, *i.e.* directly connected to at least one subgraph in CFG. That means that they are sequentially executed in each iteration. Let $c(g, g')$ denote whether the execution of a subgraph g' is consecutive after the execution of subgraph g.

$$c(g, g') = (\nexists m, g''. \, (g, m) \in E_G \wedge (g', m) \in E_G \wedge (g'', m) \in E_G \Rightarrow (g, m) \prec (g'', m) \prec (g', m)) \tag{16}$$

$$\nexists g, g', m, b, k. \, (g, m) \in E_G \wedge (g', m) \in E_G \wedge c(g, g') \wedge (b, k) \in E_B \Rightarrow b_\mathrm{n}(g, m) \prec (b, k) \prec b_0(g', m) \tag{17}$$

Now we start to map the execution of subgraphs into a hardware schedule. As explained in Constraint 6 in Section 3, Dynamatic forces BB to start sequentially. This leads to the following constraint regardless any dependency.

$$\forall g, g', m. \, c(g, g') \wedge (g, m) \prec (g', m) \Rightarrow t(b_\mathrm{n}(g, m)) < t(b_0(g', m)) \tag{18}$$

If it is proven that the execution (g, m) cannot have any dependency with the execution (g', m) of its consecutively following subgraph, there is no need to use muxes and LSQs to resolve the dependency between these two subgraphs. Then (g', m) can start execution early, such as $t(b_0(g, m)) = t(b_0(g', m)))$, which leads to a correct schedule with a better performance. Let $d'(g, g')$ denote that two subgraphs g and g' may have a dependency, and let $d'_c(g, g')$ denote that there exists a subgraph between the execution of g and g' in the same iteration including g that may have a dependency with subgraph g'.

$$d'(g, g') = (\exists b, b'. \, b \in B_g \wedge b' \in B_{g'} \Rightarrow d'(b, b')) \tag{19}$$

$$d'_c(g, g') = d'(g, g') \vee (\exists g'', m. \, (g, m) \prec (g'', m) \prec (g', m) \Rightarrow d'(g'', g')) \tag{20}$$

Constraint 6 is now relaxed to:

$$\forall g, g', m. \, (g, m) \prec (g', m) \wedge d'_c(g, g') \Rightarrow t(b_\mathrm{n}(g, m)) < t(b_0(g', m)) \tag{21}$$

$$\forall b, b', k, k', g. \, b \in B_g \wedge b' \in B_g \wedge g \in G \wedge (b, k) \prec (b', k') \Rightarrow t(b, k) < t(b', k') \tag{22}$$

Constraint 21 restricts the starting time of a subgraph execution by its most recently executed subgraph that has dependencies. Inside each subgraph, BB execution remains to start sequentially, as shown in Constraint 22.

The optimised schedule still respects all the inter-BB dependencies since it only modifies the start times of independent subgraph execution. The intra-BB dependencies remain unchanged since the transformation is applied at only the subgraph level. Therefore Constraint 7 still holds for the optimised schedule. Also, Constraint 8 and Constraint 9 still hold as the hardware pipelining and dataflow architecture remain the same.

The following sections explain how to solve two main problems: 1) How to efficiently determine a large set of G and a highly parallelised schedule for G? 2) How to map a parallelised schedule into efficient hardware?

```
 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 dependency.

Fig. 5. A Boogie program generated for the example in Fig. 3. It tries to prove the absence of memory dependency between two sequential loops `loop_0` and `loop_1`.

4.2.2 Subgraph Extraction. Given an input program, our toolflow analyses sequential loops in each loop 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. 3 has a set of two subgraphs, corresponding to `loop_0` and `loop_1`. Our toolflow then checks the dependency among the subgraphs for each set. Dynamatic translates data dependency into handshake connections in hardware for correctness. Our toolflow does not change these connections, so the data dependency is still preserved. For memory dependencies, our toolflow generates a Boogie program to prove the absence of dependency among subgraphs. For this example, Boogie proves that the two loops do not conflict on any memory locations and can be safely reordered.

For example, Fig. 5 shows the Boogie program that proves the absence of a dependency between `loop_0` and `loop_1` in Fig. 3. The Boogie program consists of two procedures. First, the procedure in Fig. 5a 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. 5b. The `for` loop structures are automatically translated using an open-sourced tool named EASY [13]. 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. 3a is reachable by the procedure in Fig. 5a. The assertions in Fig. 5b must hold for any possible memory location returned by the procedure in Fig. 5a to pass verification.

Fig. 5b 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 execution of `transformVector`. The assertion describes the dependency 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 independency, where they either touch different arrays or different indices, or they are both load operations. If the assertion always holds, 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 two subgraphs from k. The subgraphs are rescheduled based on the verification results. If a subgraph is independent of any

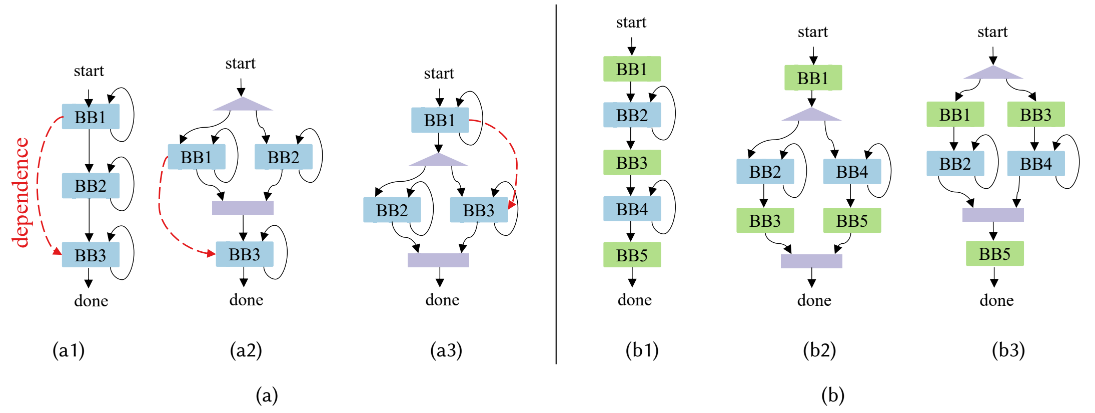


Fig. 6. A CFG may be parallelised differently, depending on (a) parallelising in top-to-bottom/bottom-to-top order, and (b) grouping BBs with the loop before/after. The dashed arrows represent memory dependency.

of its preceding subgraphs within a distance of n, it can simultaneously start with its $(m - n)$th last subgraph. In this article, we analyse at loop level for parallelism instead of the BB level for better scalability, and the search space is already huge. In the cases 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. 6a1 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. 6a2 or in Fig. 6a3. Our current approach greedily parallelises BBs in top-to-bottom order, so it yields Fig. 6a2 by default, but this order can be overridden via a user option. It may be profitable in future work to consider Fig. 6a3 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. 6b1 can be parallelised to Fig. 6b2 or to Fig. 6b3. In Fig. 6b2, BB2 is grouped with its succeeding loop BB3, and so is BB4. In Fig. 6b3, 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. 6b2 is memory-legal, and Fig. 6b3 is invalid (our toolflow will keep the CFG as in Fig. 6b1). This grouping can be controlled via a user option.

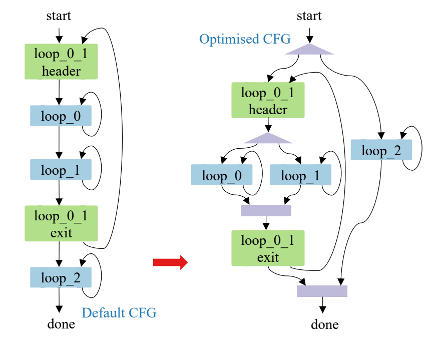
4.3 Hardware Transformation

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.

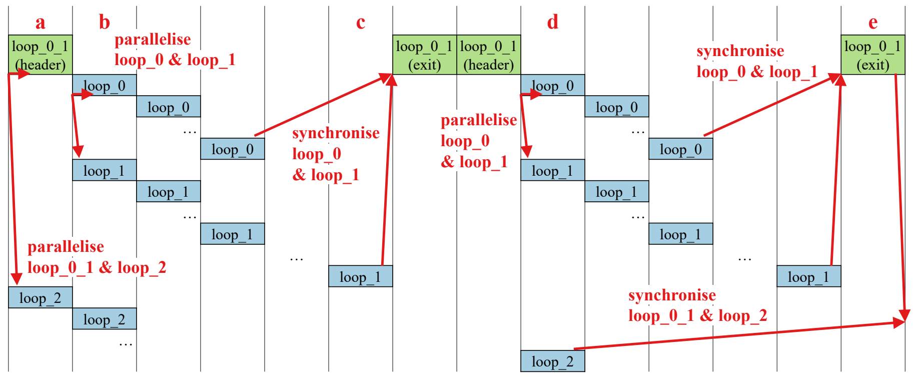
4.3.1 Components Insertion for Parallelism. With given sets of subgraphs that start simultaneously, our toolflow inserts additional components into the dynamically scheduled hardware to enable parallelism. For each set, our toolflow finds the start of the first subgraph and the exit of the last subgraph 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. 4b, the start of the function is forked to trigger

```
loop_0_1:
for(int i = 0; i < A; i++){

  loop_0:
  for(int j = 0; j < B; j++)
    ...

  loop_1:
  for(int k = 0; k < C; k++)
    ...

}

loop_2:
for(int h = 0; h < D; h++)
  ...
```

(a) Source



(b) CFG transformation.


a
b
c
d
e
loop_0_1 (header)
parallelise loop_0 & loop_1
loop_0
loop_1
...
parallelise loop_0_1 & loop_2
loop_2
synchronise loop_0 & loop_1
loop_0_1 (exit)
loop_0_1 (header)
parallelise loop_0 & loop_1
synchronise loop_0 & loop_1
synchronise loop_0_1 & loop_2
loop_0_1 (exit)

(c) Parallel BB starting schedule.

Fig. 7. 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 starts simultaneously after the transformation in (b). (c) only shows the time when a BB starts, where a BB may take multiple cycles to execute.

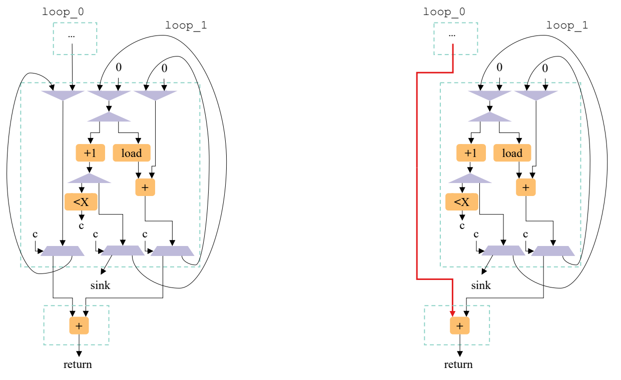
both `loop_0` and `loop_1`. A join is used to synchronise the BB starting signals in `loop_0` and `loop_1`. The join waits for all the BBs in both loops to start and then starts the succeeding BB of the loops.

The BB starting order 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 in 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 dependencies between these subgraphs and the succeeding BB are still respected at run time as they start in order. This 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.

```
1 loop_0:
2 for(int j = 0; j < Y; j++)
3   t += B[j];
4
5 loop_1:
6 for(int i = 0; i < X; i++)
7   s += A[i];
8
9 return s+t;
```

(a) Source.



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

Fig. 8. An example of simplified 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 dependencies, stalling `loop_1` before `t` is valid. Our toolflow identifies and removes these cycles, such that `loop_1` can start earlier.

Fig. 7 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 dependency 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 illustrated in Fig. 7b. `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. 7c, which only shows the time when each BB starts. A BB may have a long latency and execute in parallel with other BBs.

4.3.2 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 dependency between two operations.

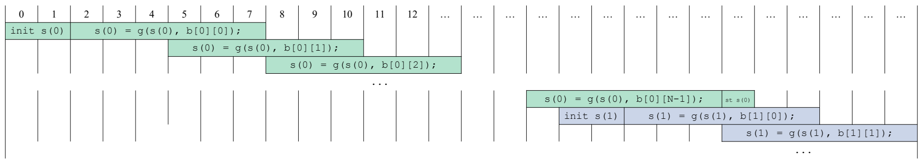
The data flow of a loop uses cycles for each variable that has carried dependency. The data circulates in the cycle and updates its value in each iteration. However, such an approach also maintains all the live variables in these cycles while executing a loop, even when they are not used inside the loop.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

For example, the loops in Fig. 8a 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. 8b. The loop iterator `i` and the variable `s` have carried dependency 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`.

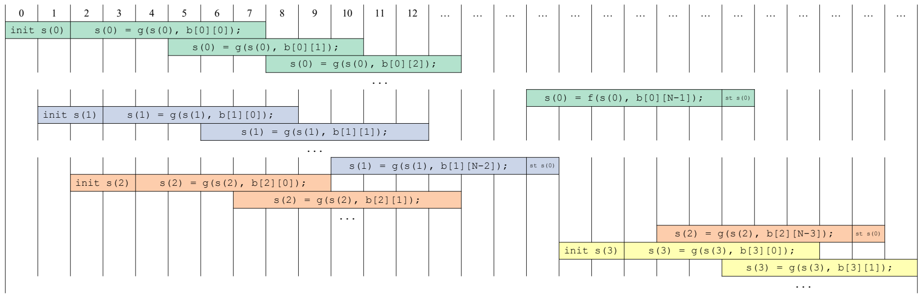
`loop_1` is stalled by the absence of `t` even when parallelised with `loop_0`, but `t` is not needed by `loop_1`. In order to remove these unnecessary cycles, our toolflow checks whether a live variable is used in the loop. If it is not, our toolflow removes the corresponding cycle and directly forwards the variable to its next used BB. Fig. 8c illustrates the transformed dataflow graph. `t` is now directly forwarded to the final adder, enabling two loops to start simultaneously.

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

(a) Source



(b) Default pipeline schedule.



(c) Proposed pipeline schedule.

Fig. 9. A motivating example of computing a triangle matrix. Assume there is no inter-iteration dependency 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.

4.3.3 LSQ Handling. The parallel BB schedule also affects the LSQs. First, the original Dynamatic starts BB sequentially, whereas 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 Section 4.2.2.

Second, the arbiter may cause deadlock if the LSQ depth is not sufficient to consume and reorder all memory accesses, (*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 work, 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).

5 DYNAMIC C-SLOW PIPELINING

This section demonstrates another application of the proposed dependency model for achieving out-of-order execution of independent and consecutive iterations of the same BB by C-slow pipelining in nested loops. It formalises our prior conference paper [17] in the proposed model. Section 5.1 demonstrates a motivating example of C-slow pipelining the innermost loop of a loop nest. Section 5.2 explains how to formulate the problem into the proposed model in Section 3 and shows how to use Microsoft Boogie to automatically determine the absence of dependency between the outer-loop iterations of the loop nest for C-slow pipelining. Section 5.3 explains how to realise C-slow pipelining in hardware.

5.1 Motivating Example

In this section, we use a motivating example to demonstrate the problem of pipelining a nested loop. In Fig. 9a, 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 each outer-loop iteration.

For simplicity, assume there is no inter-iteration dependency in the outer loop `loop_0`. Assume the latency of function `g` is three cycles. An inter-iteration dependency 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. 9b. 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 last iteration of `loop_1` in the first iteration of `loop_0` starts. Although the loop contains three registers, the control flow is not parallelised because of Constraint 6.

The schedule shown in Fig. 9c is also correct and achieves better performance. Since the II of `loop_0` is 3, the two empty slots between every two consecutive iterations allow the next two iterations of `loop_0` to 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 the 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. 9c is that the control flow in the latter schedule is out-of-order which breaks Constraint 6. 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 section, 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 performance. In this section, we propose a general approach that works for arbitrary nested loops.

5.2 Problem Formulation and dependency Analysis

Here we first show how to formalise the problem of C-slow pipelining based on the model in Section 3. We then show how to analyse the correct C for each loop nest based on the dependency analysis using the generated Boogie program by our tool.

5.2.1 Problem Formulation. 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. A loop in a CFG is defined as a set of consecutive BBs with back edges. Here we define the following terms for an inner loop in a loop nest:

- $B_L \subseteq B$ denotes the set of all the BBs in a loop,
- $E_L(j,k) \subseteq E_B$ denotes the BB executions in the kth iteration of the jth instance of the loop.

Since we only focus on loops, the abstraction has been raised to the loop level for this problem, here we use $E_L(j,k)$ to denote a particular loop execution event. This improves the scalability of our analysis. The II of each loop may vary between loop iterations as it is dynamically scheduled. For a loop, the maximum II of the loop at run time can be defined as:

$$\forall b, k.\, b \in B_L \land k > 1 \land (b,k) \in E_B \land (b,k-1) \in E_B \Rightarrow II \geq t(b,k) - t(b,k-1) \tag{23}$$

An II of greater than one means there are empty pipeline slots in the schedule, which could be attributed to a lack of hardware resources or inter-iteration dependencies. Here we assume infinite buffering and only analyse the case for the stall caused by inter-iteration dependencies.

Constraint 6 forces each iteration of a loop to start execution sequentially, which can derive the following constraints for the loop:

$$\forall j, k, e, e'.\, e \in E_L(j, k-1) \land e' \in E_L(j,k) \Rightarrow t(e) < t(e') \tag{24}$$

$$\forall j, k, k', e, e'.\, e \in E_L(j-1, k) \land e' \in E_L(j,k') \Rightarrow t(e) < t(e') \tag{25}$$

Constraint 24 means that all the BB executions in an iteration of the loop cannot start unless all the BB executions in its last iteration have started. Constraint 25 means that all the BB executions in an instance of the loop cannot start unless all the BB executions in its last instance of the loop have started. The difference of the start times between two iterations when an II is greater than 1, also known as the empty pipeline slots, cannot be filled with the following iterations because of Constraint 24 and Constraint 25. If it is proven that the jth instance and the (j-1)th instance of the loop are independent, Constraint 25 can be relaxed as the absence of dependency enables early execution of the jth instance. This allows the following iterations to start early, filling the empty slots as illustrated in Fig. 9c.

The parallelism among iterations depends on the number of consecutive independent instances, also known as the dependency distance of the outer loop. The minimum dependency distance Q of the outer loop of a loop l must satisfy the following:

$$d''(j, j') = (\exists k, k', e, e'.\, e \in E_L(j,k) \land e' \in E_L(j', k') \Rightarrow d(e, e')) \tag{26}$$

$$\forall j, q.\, j > Q \land 1 \leq q \leq Q \Rightarrow \neg d''(j-q, j) \tag{27}$$

The constraint for a C-slowed nested loop is that there are always *at most* Q outer-loop iterations executing concurrently.

Although the outer-loop iterations are parallelised, the execution of inner loops still follows Constraint 24, where the inter-iteration dependencies of the inner loops are respected. Constraint 25 transformed from Constraint 6 is then relaxed to the following combined with Constraint 24.

$$\forall j, k, e, e'.\, j > Q \land e \in E_L(j-Q, k) \land e' \in E_L(j,k) \Rightarrow t(e) < t(e') \tag{28}$$

```
 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 dependency 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 dependency for a given C.

Fig. 10. A Boogie program generated for the example in Fig. 9. It tries to prove the absence of memory dependency between any two outer-loop iterations with a distance less than C.

Constraint 7 still holds as only the independent iterations execute earlier. Constraint 8 and Constraint 9 still hold as the hardware property remains the same. The starting order of BB executions outside the C-slowed loop remains the same as vanilla Dynamatic. The starting order of BB executions inside the C-slowed loop is now in parallel and out-of-order.

The following sections explain how to solve two main problems: 1) How to efficiently determine C for a correct schedule with better performance and area efficiency? 2) How to transform the hardware to realise C-slow pipelining?

5.2.2 *Exploration for C using Dependency Analysis.* The dependency constraint above is equivalent to the minimum dependency distance of the outer loop must be greater than C, *i.e.* $C \leq Q$. A loop-carried data dependency always has a dependency distance of 1, therefore, we only need to analyse memory dependencies. Our toolflow automatically generates a Boogie program to describe the memory behaviour of the nested loop and calls the Boogie verifier to prove the absence of memory dependency within a given distance.

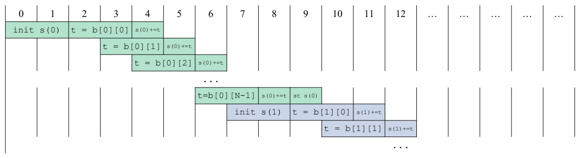
For example, Fig. 10 illustrates the Boogie program generated for the motivating example in Fig. 9. It tries to prove the absence of memory dependency between any two outer-loop iterations with a distance less than C, which mainly includes two parts. The Boogie procedure in Fig. 10a arbitrarily picks 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 in lines 6-11 in Fig. 10b.

In Fig. 10a, the `for` loop structures in Boogie are directly generated by the automated tool named EASY [13]. 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.

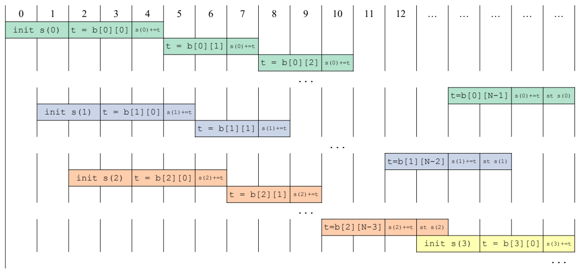
If all these memory accesses are skipped, the procedure exits at line 23 with a false `valid` bit, indicating that the returned parameters are invalid. Fig. 10b describes the main Boogie procedure for dependency analysis. It takes a

```c
int a[N], b[N][M];

void vecTrans() {
  loop_0:
  for (int i = 0; i < N; i++) {
    int s = a[i];
    loop_1:
    for (int j = 0; j < N-i; j++) {
      int t = b[i][j];
      s += t;
    }
    a[i] = s;
  }
}
```

(a) Source



(b) Default pipeline schedule.



(c) 3-slowed schedule.

Fig. 11. A large C may not improve the overall throughput but slow the execution of single instances. A C of 3 only leads to an additional area for the code example above.

given C as an input. In 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. 10a.

The assertion at line 15 proves Constraint 27 for a given C. First, the picked two memory accesses from lines 12 and 13 must hold valid parameters (line 15). Second, two accesses touching different arrays cannot have dependency (line 16). Two accesses executed by the same statement are safe (line 17), where the dependency is captured by the hardware logic based on the starting order of BB iteration under Constraint 8. Two loads cannot have dependency (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 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.

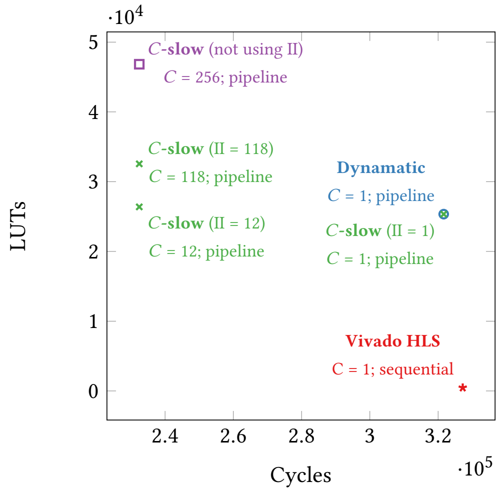
Besides the dependency constraints, a resource constraint of C-slow pipelining is that each path must be able to hold at least C sets of data. Dynamatic already inserts buffers into the hardware for high throughput. Our hardware transformation pass inserts an additional FIFO with a depth of C (named C-slow buffers) in each control path cycle of the inner loop to adapt C-slow pipelining, which can hold at least C tokens.

5.2.3 Exploration for C using Throughput Analysis. The dependency analysis above only defines a set of Cs that are suitable for C-slow pipelining. We show how to automatically determine an optimised C among these Cs using throughput analysis. The Boogie program determines an upper bound C that cannot break any dependency. However, a large C may not improve the overall throughput but only cause more area overhead.

For example, Fig. 11 illustrates an example where C-slow pipelining does not improve overall throughput. Fig. 11a shows a function named vecTrans that transforms an array named a. The function contains a nested loop. In the outer

```
float a[N], b[N][M];
void dynamicVecAccum() {
  loop_0:
  for (int i = 0; i < N; i++) {

    int s = a[f(i)];

    loop_1:
    for (int j = 0; j < i; j++) {

      // Dynamic carried dependency on s
      // causes II = 1 or 118
      if (b[i][j] != 0)
        s = g(s, b[i][j]);

    }

    // Dynamic memory dependency distance
    // causes dynamic II
    // d >= 256
    a[h(i)] = s;

  }
}
```

(a) Source code.



(b) Area and performance using different approaches.

Fig. 12. An example where both the inner loop and the outer loop have a dynamic carried dependency, leading to dynamic IIs. Choosing C from average IIs achieves the best performance, and has the minimum area among dynamically scheduled hardware. The results are measured from uniformly distributed data.

loop, it loads the element in array a and accumulates the values in matrix b onto the element. In the inner loop, the elements in matrix b are accumulated in a triangle form.

The default pipeline schedule of function vecTrans is shown in Fig. 11b. In the schedule, both the inner loop and the outer loop are fully pipelined. Although there is a carried dependency in the inner loop on the variable s, the integer adder has a latency of one clock cycle, leading to an II of 1.

This example is not amenable for C-slow pipelining, however, this cannot be identified by Boogie. In the dependency distance analysis, Boogie found that C can be any positive integer as there is no inter-iteration dependency in the outer loop. For example, Fig. 11c shows a 3-slowed schedule for function vecTrans. In the schedule, the start time of iterations in the first inner loop instance is delayed by three cycles, allowing two iterations in the second and third inner loop instances to start early. Such transformation preserves correctness but has no impact on the overall throughput. For this example, C-slow pipelining only causes more area overhead by adding the additional scheduler for out-of-order execution.

A condition where C-slow pipelining potentially improves the overall throughput is that the II of an inner loop is greater than 1. This leaves empty pipeline slots for early execution of the later inner loop instances. When the program is dynamically scheduled, the II of an inner loop may vary at run time. Here we use probabilistic analysis to statically infer an optimised C from the average II.

Fig. 12a shows a code example where the dependencies in both the inner loop and the outer loop are dynamic. In the outer loop, the function loads and computes an element in array a at an index of f(i). It then updates the element in the same array at an index of h(i). In the inner loop, a variable s updates itself based on the elements in matrix b, which is used to update array a in the outer loop.

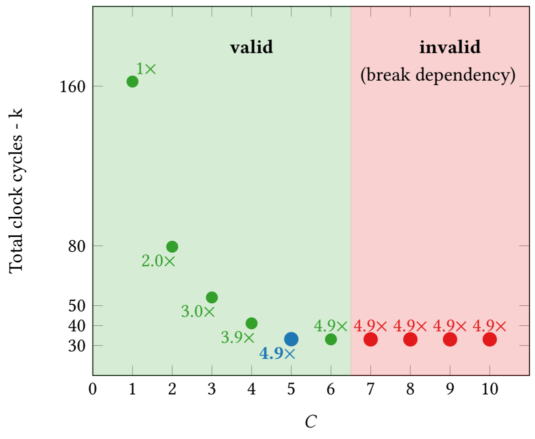


Fig. 13. Speedup by varying C for the example in Fig. 9. $C > 6$ breaks the memory dependency in the outer loop. Increasing C initially improves the throughput. However, once all the empty slots are filled, further increasing C has less effect on the throughput. Our tool automatically determines an optimal $C = 5$, shown in blue. `f(i) = i`, `g(x, y) = x + y` and `h(i) = i * i + 7`.

In the outer loop, the memory dependencies between the loads and stores with array `a` are dynamic and depend on the iteration index of the out-loop `i`. For simplicity, assume that the minimum dependency distance in the outer loop for given `f(i)` and `g(i)` is no less than 256. That is, C-slow pipelining is valid for any C where $1 \leq C \leq 256$.

In the inner loop, the data-dependent condition causes two possible IIs. When the condition at line 11 is false, function `g` is skipped, leading to an II of 1. When the condition is true, function `g` is executed and the carried dependency on `s` causes an II of 118. The II of 118 is contributed by the latency between the input and the output of function `g`. The overall throughput then depends on the distribution of elements in matrix `b`, which affects the condition.

We then have 256 options to choose C no greater than 256, limited by the dependency in the outer loop. Here we discuss three approaches for choosing C based on the II of the loop. First, choosing a C based on an optimistic II reduces the area overhead of adding the scheduler for C-slow pipelining. However, a small C may limit the parallelism among inner loop instances when there are more empty slots for certain input data. For this example, an extreme case is where the minimum II is 1, which disables C-slow pipelining ($C = 1$ is equivalent to a single thread). Second, choosing a C based on a conservative II enables sufficient pipeline slots for early execution following inner loop instances. However, a large C may add unnecessary area overhead when there are only a small number or none of the empty pipeline slots, as illustrated in Fig, 11b. Finally, choosing a C based on an average II may balance the tradeoff between area overhead and performance improvement, achieving a more efficient hardware design. Our toolflow reuses the results of the throughput analysis during the Dynamatic synthesis flow. The buffering process in Dynamatic already uses static analysis to estimate the II of each control path.

Fig. 12b shows the results of the area and performance of the example in Fig. 12a using different pipeline approaches. First, we evaluate three baselines: vanilla Dynamatic [32] for dynamic scheduling, Vivado HLS [51] for static scheduling, naive C-slow pipelining with only dependency constraints [17]. Vanilla Dynamatic allows pipelining loop with dynamic dependency, achieving better performance than static scheduling. However, the use of load-store queues that dynamically schedules memory operations causes significant area overhead. On the other hand, Vivado HLS that uses static scheduling cannot resolve data-dependent dependencies at compile time and keeps the loop sequential. The resultant hardware design below vanilla Dynamatic has poor performance but high area efficiency because of resource sharing at compile

Algorithm 1 The algorithm for finding C using average II.

```
Require: $M$                                          ▷ the input program module
  $C_L \leftarrow \{\}$                               ▷ a dictionary to return, which indicates an $C$ for each nested loop
  $L \leftarrow get\_nested\_loops(M)$                ▷ a set of nested loops from the input program
  for each nested loop $l$ in $L$ do
      $L' \leftarrow get\_sub\_loops(l)$              ▷ a set of sub-loops for loop $l$
      $II \leftarrow get\_average\_II(L')$            ▷ the set of average IIs of the sub-loops in loop $l$
      $C \leftarrow \min II$                          ▷ the starting $C$ for searching by picking the minimum value of IIs
      $B \leftarrow get\_Boogie\_program(M, l)$       ▷ the Boogie program for checking $C$ for loop $l$
      $D \leftarrow Boogie\_verify(B, C)$             ▷ the verification state returned by Boogie verifier
      while $C > 1$ & $D \neq$ success do
          $C \leftarrow C - 1$                        ▷ a decremented $C$ for next-round verification
          $D \leftarrow Boogie\_verify(B, C)$
      end while
      $C_L(l) = C$                                    ▷ the optimised $C$ for loop $l$
  end for
  return $C_L$
```

time. Finally, the naive C-slow pipelining only analyses the dependency distance in the outer loop for choosing C. It generates hardware sitting on the top left that has better performance than both because it allows early execution of later inner loop instances. However, the value of C is over-approximated to a large value. This results in unnecessary area overhead since only a small portion of the inserted C-slow buffer in the control path is used.

We also evaluate the three approaches mentioned above for choosing an optimised C. First, the minimum II indicates that there is no empty slot in the inner loop schedule, preventing the transformation for C-slow pipelining. This leads to the same design as vanilla Dynamatic. The case where these two points overlap is only for this particular benchmark, where the minimum II of 1. Otherwise, the result with the minimum II greater than one may not overlap with the result by vanilla Dynamatic. Second, the maximum II indicates the best case that maximises parallelism for c-slow pipelining. However, this could still cause unnecessary area overhead when the iterations that have the maximum II are rare, such as in the case where only one iteration has the maximum II and the rest have the minimum II. Finally, the average II estimates the overall throughput of the inner loop. Such analysis reduces the C-slow buffer size while preserving sufficient slots for parallelism and maintaining the high performance of the naive C-slow approach. The constraint for an optimised C is then:

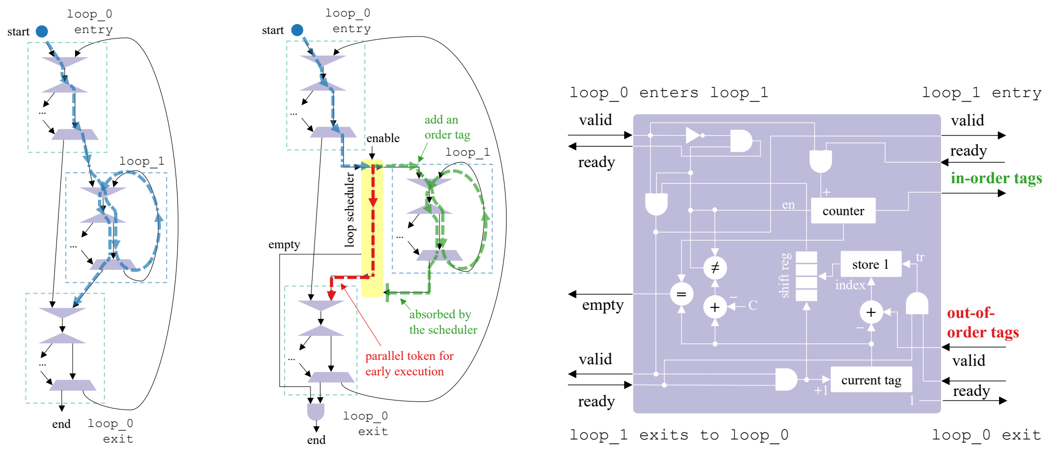
$$C \leq \min(Q, II_{av}) \tag{29}$$

where Q is the maximum C that passes the Boogie verification (also known as minimum dependency distance), and II_{av} is the average II of the inner loop. Detailed algorithm implementation is shown in Algorithm 1.

Here we take the motivating example as a case study and then discuss the overall results for all the benchmarks. Fig. 13 shows the total clock cycles of the hardware for the motivating example with different C. Only $C \leq 6$ for this example does not break memory dependency, where $C_D = 6$. When C increases initially, more outer-loop iterations are parallelized, significantly improving the throughput. The average II of the inner loop is 5. When C is greater than 5, the throughput remains the same since almost all the empty pipeline slots have been filled. The overhead caused by $C = 6$ is a larger size of FIFOs used for storing the data.

5.3 Hardware Transformation

Once the value of C for a loop is determined, our toolflow 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



(a) Default CFG. (b) Transformed CFG. (c) Loop scheduler

Fig. 14. Our toolflow considers each instance of the innermost loop as a thread and achieves the schedule in Fig. 9c. The dashed arrows represent the token transition in the control flow. The scheduler tags the control tokens in the innermost loop to reorder them at the output after out-of-order execution.

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. 14 shows the proposed loop scheduler integrated into a dynamically scheduled control flow graph. For example, the control flow graph of the code in Fig. 9 from vanilla Dynamatic is shown in Fig. 14a. 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 the current BB execution. Then a fork is used to produce other tokens to trigger all the data operations 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. 14a 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 is 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. 14b 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

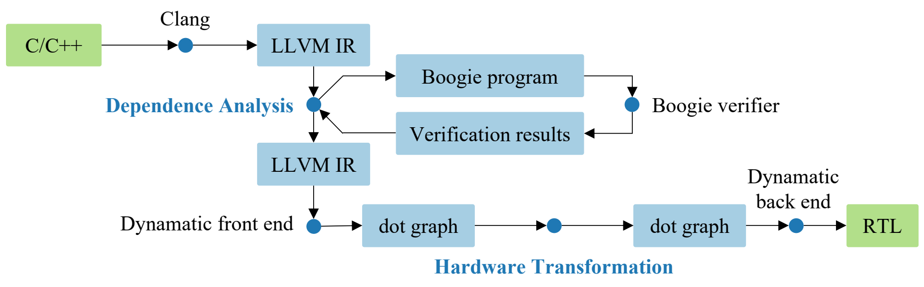


Fig. 15. Our work integrated into Dynamatic. Our contributions are highlighted in bold blue text.

another token t_2 and sends it to the bottom BB to execute the control flow early (indicated as the red dashed arrow).

(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 is 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 remaining in the inner loop.

Since the execution order of loop iterations has changed, all the data flows must be strictly scheduled by the control flow. Dynamatic uses merges to accept input data at the input of a BB when there is always at most one valid input. There may be multiple valid inputs at the start of BB after the transformation. In order to preserve correctness, we replace all the merges in the inner loop with muxes, such that the data is always synchronised with the control token and can recover in-order using the tag in the control token. An advantage of this approach is that only the control tokens need to be tagged to preserve the original order, where the data flow is always synchronised by the control flow.

The design of the loop scheduler is shown in Fig. 14c. It guards the entry and exit of the inner loop. The ready signal at the bottom right is forced to 1 as the scheduler already controls the input throughput using C, and there cannot be back pressure at the output. 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.

6 TOOLFLOW

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. 15, the input C program is first lowered into LLVM IR. Then the dependency in the code is analysed by the generated Boogie program as explained in Section 4.2.2 and Section 5.2.2. Our Boogie program generator generates Boogie assertions and calls the Boogie verifier to automatically verify the absence of dependency for the extracted instances (subgraphs or outer iterations). 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 for the corresponding transformation explained in Section 4.3 and Section 5.3, 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. Our work can also be integrated into other HLS tools, such as CIRCT HLS [21].

7 EXPERIMENTS

We compare our work with Xilinx Vivado HLS [51] and the original Dynamatic [32]. 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 the UltraScale+ family of FPGA devices for experiments, and the version of Xilinx software is 2019.2.

7.1 Experiment Setup and Benchmarks

The benchmarks are chosen based on whether our approaches are applicable. Finding suitable benchmarks is a perennial problem for papers that push the limits of HLS, in part because existing benchmark sets such as Polybench [45] and CHStone [27] tend to be tailored to what HLS tools can already comfortably handle. We use two open-sourced benchmark sets for evaluation. One is the LegUp benchmark set by Chen and Anderson [12] for evaluating multi-threaded HLS. The LegUp benchmark set manually specifies the threads using Pthreads [2]. We inlined all the threads to a sequential program. The other benchmark set is from Polybench [45] but modified for sparse computation for evaluating dynamic-scheduling 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 our target FPGA. We also partitioned the memory in the blocking 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.

7.2 Dynamic Inter-Block Scheduling

Fig. 16 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 `fft` indicate speedup factors above 1. Placing more subgraphs in parallel leads to more speedup, with `histogram` and `matrixtrans` achieving optimal speedups. In the `fft` benchmark, two reasons for the lack of speedup are: 1) that other parts of the CFG have to be started sequentially, and 2) that the memory is naively partitioned in a block scheme, so the memory bandwidth is limited, and there is serious contention between BBs for the LSQs.

Detailed results for both benchmark sets are shown in Table 2. For the LegUp benchmark set, we observe the following:

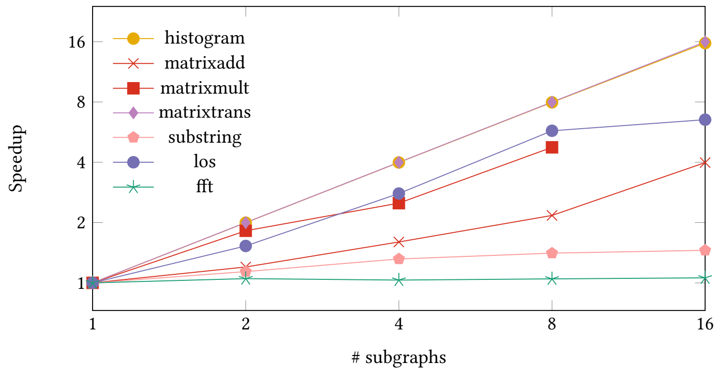


Fig. 16. Speedup, compared to original Dynamatic, as more subgraphs in the CFG are parallelised.

(1) Static scheduling (Vivado HLS) is the clear winner in terms of area (see rows '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) Inter-block scheduling requires substantially fewer cycles than the original Dynamatic thanks to the parallelism it exploits between BBs (see column 'Cycles').
(3) Inter-block scheduling achieves up to 8.05× speedup (see row 'Wall clock time' for `histogram`). We further observe that the area-delay products we obtain are down to 0.125× of the original Dynamatic.
(4) 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 the innermost subgraphs still start sequentially, leading to large cycle counts.
(5) The performance of `fft` does not change because the performance bottleneck is the memory bandwidth, where the parallelised loops always access the same memory block concurrently. This could be further improved by optimising the array partitioning scheme, but it is orthogonal to this work.

7.3 C-slow Pipelining

For the C-slow pipelining benchmark set, we make the following observations for C-slow pipelining:

(1) The observations on the area are similar to the results for inter-block scheduling as the C-slow scheduler is small.
(2) C-slow pipelining requires substantially fewer cycles than the original Dynamatic thanks to the parallelism it exploits between outer-loop iterations (see rows 'Cycles').
(3) C-slow pipelining enables a speedup up to 4.5× with only 8% area overhead (see rows 'Wall clock time' for `matrixmult`). We further observe that the area-delay products we obtain are down to 0.28× of the original Dynamatic.

7.4 Combining Inter-Block Scheduling and C-slow Pipelining

For the C-slow pipelining benchmark set, we also make the following observations for combining both approaches:

(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]).

Table 2. Evaluation of our work on two benchmark sets. For each benchmark, we highlight the **best** results in each dimension. vhls = Vivado HLS; dhls = original Dynamatic; cslow = C-slow pipelining; inter-block = inter-block scheduling; both = inter-block scheduling + C-slow pipelining. The code size lists the number of loops, BBs, instructions and extracted subgraphs.

Benchmarks		LegUp benchmarks [12]							C-slow benchmarks [17]			
		histogram	matrixadd	matrixmult	matrixtrans	substring	los	fft	trVecAccum	covariance	syr2k	gesummv
Code size	loops	9	8	72	8	16	24	24	16	48	24	16
	bbs	91	17	145	17	54	89	65	49	113	49	33
	insts	384	112	1521	81	255	513	457	249	593	385	297
	graphs	9	8	72	8	8	8	8	8	24	8	8
LUTs (1000s)	vhls	**1.67**	**1.11**	**6.87**	**0.103**	**0.938**	**2.68**	**2.65**	**1.21**	**3.83**	**2.04**	**2.05**
	dhls	156	9.15	79.4	2.67	14.4	46.5	351	149	56.6	30.5	26.9
	inter-block	156	9.17	79.8	2.69	14.6	46.1	351	150	55.5	30.7	26.9
	cslow	156	9.15	101	2.67	14.4	46.5	351	151	56.8	30.6	27.6
	both	156	9.17	100	2.69	14.6	46.1	351	153	65.7	34.4	30.2
DSPs	vhls	**0**	**2**	**5**	**0**	**0**	**0**	**16**	**10**	**5**	**5**	**144**
	dhls	0	30	320	0	0	0	192	40	72	152	144
	inter-block	0	30	320	0	0	0	192	40	72	152	144
	cslow	0	30	320	0	0	0	192	40	72	152	144
	both	0	30	320	0	0	0	192	40	72	152	144
Cycles (1000s)	vhls	197	262	4195	65.6	98.3	48.8	86	1060	668	647	2130
	dhls	317	106	1090	65.6	217	114	5.39	393	605	602	787
	inter-block	**39.8**	**48.9**	229	**8.2**	**154**	**19.9**	**5.15**	161	77.1	84.1	327
	cslow	317	106	**164**	65.6	217	114	5.39	256	263	255	524
	both	**39.8**	**48.9**	**164**	**8.2**	**154**	**19.9**	**5.15**	**33.2**	**33.6**	**33.9**	**66.6**
Fmax (MHz)	vhls	**464**	**159**	**155**	**562**	**470**	281	**155**	**159**	**155**	**155**	155
	dhls	57.7	110	123	227	126	272	81.8	132	132	126	162
	inter-block	58.3	110	104	210	129	**282**	103	121	102	98.9	**163**
	cslow	57.7	110	83.3	227	126	272	81.8	157	89.7	130	119
	both	58.3	110	72.3	210	129	**282**	103	117	102	128	120
Wall-clock time (ms)	vhls	**0.424**	1.65	27	0.117	**0.209**	0.174	0.555	6.65	4.3	4.17	13.7
	dhls	5.49	0.968	8.86	0.288	1.72	0.419	0.0659	2.97	4.57	4.79	4.87
	inter-block	0.682	**0.446**	2.2	**0.039**	1.2	**0.0705**	**0.0501**	1.32	0.759	0.85	2.01
	cslow	5.49	0.968	**1.97**	0.288	1.72	0.419	0.0659	1.64	2.93	1.96	4.39
	both	0.682	**0.446**	2.27	**0.039**	1.2	**0.0705**	**0.0501**	**0.284**	**0.328**	**0.264**	**0.553**
Relative area-delay product	vhls	**1**	**1**	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	1
	dhls	1210	4.83	3.79	64	126	41.8	15.7	55	15.7	94	4.67
	inter-block	151	2.23	**0.946**	8.71	88.9	6.96	12	24.7	2.56	16.5	1.93
	cslow	1210	4.83	1.07	64	126	41.8	15.7	40.3	9.61	39	5.17
	both	151	2.23	1.22	8.71	88.9	6.96	12	5.41	1.31	5.12	**0.595**

Table 3. Verification time in seconds for dependence check in Boogie.

Benchmarks	histogram	matrixadd	matrixmult	matrixtrans	substring	los	fft	triangleVecAccum	covariance	syr2k	gesummv
Inter-Block	1.128	0.002	1.080	1.102	1.122	1.197	1.182	1.080	1.098	1.210	1.103
C-Slow	-	-	-	-	-	-	-	1.215	1.098	1.088	1.050

(2) Unrolling alone is not enough to obtain substantial speedups because the BBs still have to start sequentially (see column 'Cycles $\longrightarrow$ unroll').

(3) 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.

7.5 Verification Times

Table 3 lists the verification time taken by the Boogie verifier to check the absence of dependency. It shows that for all the benchmarks, the verification time takes no more than two seconds in a run. The verification time scales exponentially with the number of memory statements in the input programs and the complexity of the memory access pattern. However, optimisations such as profiling and affine analysis could simplify the formal verification problem in Boogie for better scalability.

8 CONCLUSIONS

Existing dynamic-scheduling HLS tools require all BBs to start in strict program order, in order to respect any dependencies between BBs, regardless of whether dependencies are actually present. This leads to missed opportunities for performance improvements by having BBs start simultaneously. In this article, we propose a general dependency framework that analyses the inter-BB dependencies and identifies the existing restrictions of the dynamic scheduling approach for HLS. Our model helps guide the researchers to push the limit of the state-of-the-art scheduling technique by lifting these restrictions and exploring more optimisations. This could further improve the performance of the generated HLS hardware by identifying the absence of certain dependencies in the constraints in static analysis and exploiting hardware parallelism.

We also illustrate two examples of using the proposed model for optimising dynamic scheduling. First, we show how to parallelise two consecutive loops if they are proven independent. Our tool takes an arbitrary program and automatically generates a Boogie program to verify the absence of dependency between these two loops. Second, we show how to enable C-slow pipelining for a nested loop if both the II of the inner loop and the minimum dependency distance of the outer loop are greater than one. Our tool takes a nested loop and automatically generates a Boogie program to verify the dependency constraints and uses static throughput analysis for determining an optimised C for high-performance and area-efficient C-slow pipelining.

These two proposed optimisation techniques optimise the control flow of the generated hardware from different sides and have shown that they can be composited. One aims to optimise the schedules between different loop statements, and the other aims to optimise the schedules between different iterations of the same loop statement. However, both techniques achieve hardware parallelism by statically proving the absence of dependency between two run-time events. Particularly, both use Microsoft Boogie verifier as the formal verification tool in the back end for verifying the dependency constraints statically. The performance gain is significant, while the area overhead is negligible because of more dependency information obtained using the static analysis. Such static analysis is still conservative compared to the theoretical dynamic scheduling model because it over-approximates the dependency check for two run-time events to the dependency check for two statements. The proposed approaches close the performance gap between the dynamic scheduling implementation and the theoretical dynamic scheduling. Our future work will explore the fundamental limits of the proposed model, both theoretically and practically.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Amazon. 2022. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[2] B. Barney. 2021. POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[4] David R Butenhof. 1997. *Programming with POSIX threads*. Addison-Wesley Professional.

[5] Daniel Cabrera, Xavier Martorell, Georgi Gaydadjiev, Eduard Ayguade, and Daniel Jiménez-González. 2009. OpenMP extensions for FPGA accelerators. In *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*. IEEE, 17–24.

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

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

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

[9] V. G. Castellana, A. Tumeo, and F. Ferrandi. 2014. High-level synthesis of memory bound and irregular parallel applications with Bambu. In *2014 IEEE Hot Chips 26 Symposium (HCS)*. IEEE, Cupertino, CA, USA, 1–1.

[10] Catapult High-Level Synthesis. 2021. https://www.mentor.com/hls-lp/catapult-high-level-synthesis

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

[12] Yu Ting Chen and Jason H. Anderson. 2017. 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)*. 1–8. https://doi.org/10.23919/FPL.2017.8056841

[13] Jianyi Cheng, Shane T. Fleming, Yu Ting Chen, Jason Anderson, John Wickerson, and George A. Constantinides. 2022. Efficient Memory Arbitration in High-Level Synthesis From Multi-Threaded Code. *IEEE Trans. Comput.* 71, 4 (2022), 933–946. https://doi.org/10.1109/TC.2021.3066466

[14] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-Level Synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 288–298. https://doi.org/10.1145/3373087.3375297

[15] Jianyi Cheng, Lana Josipović, George A. Constantinides, and John Wickerson. 2022. Dynamic Inter-Block Scheduling for HLS. In *2022 32th International Conference on Field-Programmable Logic and Applications (FPL)*.

[16] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2021. 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)*. 341–346. https://doi.org/10.1109/FPL53798.2021.00066

[17] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2022. Dynamic C-Slow Pipelining for HLS. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–10. https://doi.org/10.1109/FCCM53951.2022.9786096

[18] Jongsok Choi, Stephen Brown, and Jason Anderson. 2013. From software threads to parallel hardware in high-level synthesis for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*. 270–277. https://doi.org/10.1109/FPT.2013.6718365

[19] N. Y. S. Chong. 2014. *Scalable Verification Techniques for Data-Parallel Programs*. Doctoral Thesis. Imperial College London, London, UK.

[20] Jason Cong and Zhiru Zhang. 2006. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *Proceedings of the 43rd Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '06)*. Association for Computing Machinery, New York, NY, USA, 433–438. https://doi.org/10.1145/1146909.1147025

[21] CIRCT contributors. 2023. CIRCT-based HLS compilation flows, debugging, and cosimulation tools. https://github.com/circt-hls/circt-hls.

[22] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. 2009. An Introduction to High-Level Synthesis. *IEEE Design Test of Computers* 26, 4 (2009), 8–17. https://doi.org/10.1109/MDT.2009.69

[23] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[25] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.

[26] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. 2003. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *16th International Conference on VLSI Design, 2003. Proceedings.* 461–466. https://doi.org/10.1109/ICVD.2003.1183177

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

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

[29] Intel FPGA SDK for OpenCL Software Technology. 2021. https://www.intel.co.uk/content/www/uk/en/software/programmable/sdk-for-opencl/overview.html

[30] Intel HLS Compiler. 2022. https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/hls-compiler.html

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

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

[33] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2022. From C/C++ Code to High-Performance Dataflow Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 7 (2022), 2142–2155. https://doi.org/10.1109/TCAD.2021.3105574

[34] Lana Josipović, Axel Marmet, Andrea Guerrieri, and Paolo Ienne. 2022. Resource Sharing in Dataflow Circuits. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–9. https://doi.org/10.1109/FCCM53951.2022.9786084

[35] K. Rustan M. Leino. 2008. This is Boogie 2. (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[36] Charles E Leiserson, Flavio M Rose, and James B Saxe. 1983. Optimizing synchronous circuitry by retiming (preliminary version). In *Third Caltech conference on very large scale integration*. Springer, 87–116.

[37] Y.Y. Leow, C.y. Ng, and W.f. Wong. 2006. Generating hardware from OpenMP programs. In *2006 IEEE International Conference on Field Programmable Technology*. 73–80. https://doi.org/10.1109/FPT.2006.270297

[38] Rui Li, Lincoln Berkley, Yihang Yang, and Rajit Manohar. 2021. Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures. In *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 1–8. https://doi.org/10.1109/ASYNC48570.2021.00009

[39] J. Liu, J. Wickerson, and G. A. Constantinides. 2016. Loop Splitting for Efficient Pipelining in High-Level Synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 72–79. https://doi.org/10.1109/FCCM.2016.27

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

[41] Yury Markovskiy and Yatish Patel. 2002. Simple symmetric multithreading in xilinx fpgas. (2002).

[42] Microsoft. 2022. Project Catapult. https://www.microsoft.com/en-us/research/project/project-catapult/

[43] Sayuri Ota and Nagisa Ishiura. 2019. 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)*. 1–4. https://doi.org/10.1109/ITC-CSCC.2019.8793453

[44] Ian Page and Wayne Luk. 1991. Compiling Occam into field-programmable gate arrays. In *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Vol. 15. Abingdon EE&CS Books 15 Harcourt Way, Abingdon OX14 1NV, UK, 271–283.

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

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

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

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

[49] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 93–104. https://doi.org/10.1145/3431920.3439292

[50] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. 2003. 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* (Monterey, California, USA) *(FPGA '03)*. Association for Computing Machinery, New York, NY, USA, 185–194. https://doi.org/10.1145/611817.611845

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

[52] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. 2020. Using OpenCL to Enable Software-like Development of an FPGA-Accelerated Biophotonic Cancer Treatment Simulator. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 86–96. https://doi.org/10.1145/3373087.3375300

[53] Z. Zhang and B. Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 211–218. https://doi.org/10.1109/ICCAD.2013.6691121

[54] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. 2017. 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* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 179–188. https://doi.org/10.1145/3020078.3021734

[55] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. Improving High Level Synthesis Optimization Opportunity through Polyhedral Transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/2435264.2435271