Polyhedral-based Dynamic Loop Pipelining for High-Level Synthesis

Junyi Liu*, John Wickerson*, Samuel Bayliss*†, and George A. Constantinides*
*Department of Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, United Kingdom
{junyi.liu13, j.wickerson, g.constantinides}@imperial.ac.uk
†Research Labs, Xilinx, San Jose, CA 95124, USA
samuel.bayliss@xilinx.com

***Abstract*—Loop pipelining is one of the most important optimization methods in high-level synthesis (HLS) for increasing loop parallelism. There has been considerable work on improving loop pipelining, which mainly focuses on optimizing static operation scheduling and parallel memory accesses. Nonetheless, when loops contain complex memory dependencies, current techniques cannot generate high performance pipelines. In this work, we extend the capability of loop pipelining in HLS to handle loops with uncertain dependencies (*i.e.*, parameterised by an undetermined variable) and/or non-uniform dependencies (*i.e.*, varying between loop iterations). Our optimization allows a pipeline to be statically scheduled without the aforementioned memory dependencies, but an associated controller will change the execution speed of loop iterations at runtime. This allows the augmented pipeline to process each loop iteration as fast as possible without violating memory dependencies. We use a parametric polyhedral analysis to generate the control logic for when to safely run all loop iterations in the pipeline and when to break the pipeline execution to resolve memory conflicts. Our techniques have been prototyped in an automated source-to-source code transformation framework, with Xilinx Vivado HLS, a leading HLS tool, as the RTL generation backend. Over a suite of benchmarks, experiments show that our optimization can implement optimised pipelines at almost the same clock speed as without our transformations, running approximately 3.7-$10\times$ faster, with a reasonable resource overhead.**

***Index Terms*—High-level synthesis, loop pipelining, polyhedral model, FPGA, reconfigurable computing**

I. Introduction

The continual improvement of field-programmable gate array (FPGA) technology has led to an increasing desire to use such devices for compute. High-level synthesis (HLS) tools have recently reached commercial maturity, and are now a stable technology, enabling high hardware design productivity. State-of-the-art HLS tools like Xilinx Vivado HLS [1], Intel FPGA SDK for OpenCL [2] and LegUp [3] are able to synthesize programs written in high-level languages like C/C++/OpenCL into hardware designs described in VHDL/Verilog. Hardware architectures are automatically optimized and synthesized in the process.

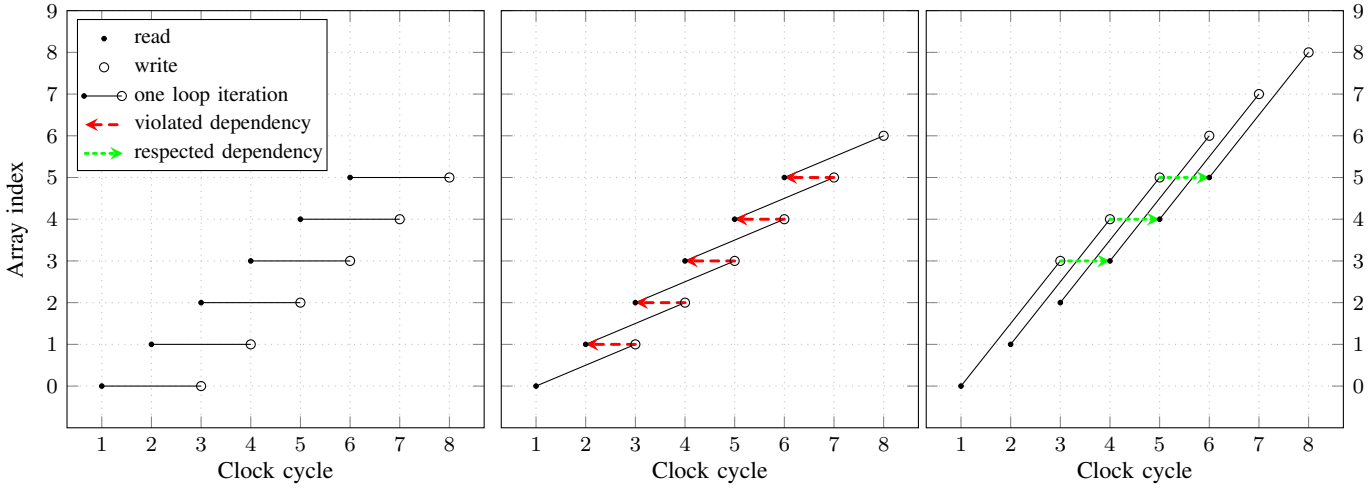
For many applications, there is still a considerable gap between the quality of results produced by HLS tools and those obtained through manual optimization of an RTL hardware design. Computational bottlenecks are typically located in some critical loops of high-level programs, and hence loop pipelining has emerged as one of the preeminent optimization techniques in HLS. Loop-pipelining techniques work by automatically detecting when a loop iteration does not depend on its predecessors, and hence can begin executing before its predecessors have completed. However, complex inter-iteration dependencies can hinder this process, and cause existing HLS tools to take an overly conservative approach to scheduling. The optimization method presented in this paper aims to make high-performance loop pipelining possible for the loops having uncertain dependencies (*i.e.*, parameterised by an undetermined variable) and/or non-uniform dependencies (*i.e.*, varying between loop iterations).

```
for (i=0; i<N; i++)
  A[i+m] = A[i] + 0.5f;
```

Listing 1: Motivational loop with uncertain dependency.

The motivational loop shown in Listing 1 contains a parameterised affine recurrence equation [4]. In this loop, there is an undetermined variable `m` in the write access pattern of array `A`. The loop iterator `i` ranges from zero to `N` $-$ 1, where `N` is constant. The value of `m` is not known at compile time. Therefore, the sequence of write accesses to elements of array `A` cannot be completely determined. Indeed, whether the loop can be pipelined actually depends on the value of the parameter `m`, as illustrated in Fig. 1. When `m` $= 0$, there is no memory dependency in the loop execution as shown in Fig. 1(a). When `m` $= 1$, the result of each iteration has to be generated before the start of the next iteration, which implies an inter-iteration dependency (also known as a recurrence). As shown in Fig. 1(b), loop pipelining with an initiation interval of one cycle would violate the read-after-write (RAW) dependency. When `m` ≥ 3, there will be no recurrence violation in the pipeline as shown in Fig. 1(c). This uncertain data dependency prevents existing HLS tools from exploiting loop pipelining by default, because they only support a fixed initiation interval. As a result, a sequential pipeline schedule will be synthesised for this loop.

Our optimization presented in this paper enables the statically scheduled pipeline to run at dynamic speed. This is the basic idea of our approach: implement the pipeline scheduled for the smallest initiation interval and throttle the execution of loop iterations according to a compile-time dependency analysis. To understand when the pipeline needs to slow down, we use parametric polyhedral analysis to firstly synthesise a lightweight runtime check, such as $1 \leq$ `m` ≤ 2 for Listing 1 according to Fig. 1. The demonstration of this analysis is



(a) $\mathtt{m} = 0$ (b) $\mathtt{m} = 1$ (c) $\mathtt{m} = 3$

Fig. 1: The impact of the undetermined variable $\mathtt{m}$ on fully pipelining the loop shown in Listing 1. We assume that $\mathtt{N} = 6$, and the iteration latency is 3 cycles.

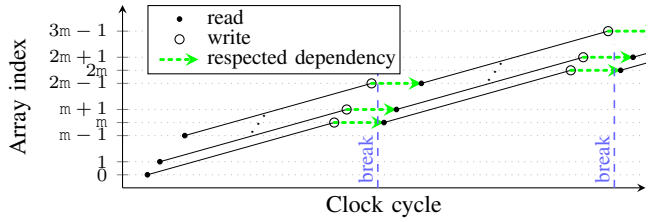


Fig. 2: Breaking the pipeline execution at $(\mathtt{m}+1)^{\text{th}}$ and $(2\mathtt{m}+1)^{\text{th}}$ iterations, when the loop shown in Listing 1 is running at initiation interval of one cycle.

preliminarily presented in [5] as parametric loop pipelining. When there exist memory conflicts that have to be resolved, the polyhedral analysis is further used to synthesise the pipeline break points. As demonstrated in Fig. 2, we can break the pipeline execution at the $(\mathtt{m}+1)^{\text{th}}$ iteration ($\mathtt{i} = \mathtt{m}$) to resolve the RAW conflict like those shown in Fig. 1b; nevertheless, the next potential conflict will happen at the $(2\mathtt{m}+1)^{\text{th}}$ iteration ($\mathtt{i} = 2\mathtt{m}$). To keep the pipeline of Listing 1 as busy as possible, we need to halt its execution after every m iterations when the memory conflicts appear. The strategy of breaking the pipeline execution can also optimize loops with non-uniform memory dependencies, which can appear in many applications such as matrix decomposition and triangular matrix computation. In these applications, the critical loops have memory dependencies that are statically analysable but vary with the value of the induction variable. An example of such a loop is shown in Listing 2. These loops can be optimized by loop splitting, first proposed in [6].

We implement the proposed optimization as a source-to-source code transformation applied *before* invoking a commercial HLS tool. The lightweight runtime throttle check and the pipeline breaks can be introduced, alongside appropriate loop-pipelining directives, to guide HLS to implement the desired pipeline architecture. Therefore, our transformation is

```
for (i=0; i<N; i++)
  A[2*i] = A[i] + 0.5f;
```

Listing 2: Motivational loop with non-uniform dependency.

also flexible enough to be applied to different HLS tools. The rest of this paper explains how our optimization approach can be generalized and automated in a prototype flow.

In particular, we make the following contributions:

- We formulate the problem with a general parametric polyhedral model, allowing us to precisely characterize the inter-iteration dependencies from both uncertain and non-uniform memory access patterns.
- We develop an algorithm that can generate the conflict region of parameters which is used as the runtime check to decide when a high-throughput pipelined schedule (*i.e.* loop pipelining with a low initiation interval) can be achieved without violating memory conflicts.
- We develop a polyhedral transformation that realises the efficient insertion of pipeline breaks for HLS.
- We implement our entire optimization as a fully automated source-to-source code transformation framework, which is compatible with, and builds on top of Vivado HLS. This tool is open-source in a public Github repository.[1]

The remainder of this paper is organized as follows. Section II presents related work in HLS. Section III gives a motivational example and its analysis for loop pipelining with uncertain variables. Section IV describes the formulation of our parametric polyhedral analysis, transformation and its implementation details. Section V presents the benchmarks and experimental results, and conclusions are drawn in Section VI.

II. Related Work

Loop pipelining, known in software compilers as 'software pipelining', was originally designed for Very Long Instruc-

[1] https://github.com/Junyi-Liu/Potholes

tion Word (VLIW) processor architectures [7]. When loops can be shown to be free of inter-iteration dependencies, the instructions from several iterations can be unrolled and interleaved to mitigate the impact of long intra-iteration dependencies between instructions. The technique can ensure memory bandwidth is effectively utilized by keeping multiple memory operations inflight at once. For VLIW machines, independent operations can be scheduled on a fixed number of parallel computational units. Classical compilation techniques like Iterative Modulo Scheduling [8] can find an effective time-space mapping to the fixed computational units within a processor.

A. Static Scheduling for Loop Pipelining

Where loop pipelining is applied in the context of an HLS tool, we have the additional freedom to select how many computational units we wish to implement. A trade-off can be made between the number of dependent operations chained within a single clock cycle, and the minimum clock-period of an implementation. Zhang *et al.* [9] propose a sophisticated approach to exploit this, which captures the dependent operations and their associated latency, and models resource and clock frequency requirements. The 'System of Difference Constraints' that they establish can be solved efficiently to explore a range of schedules achieving different area-time trade-offs. The approach by Canis *et al.* [10] further improves the method by trying to reduce the latency between inter-iteration dependent memory accesses. Their recurrence minimization helps to increase the likelihood of achieving higher parallelism. In both of these works, the authors rely on knowing, at compile time, all the dependencies that exist between operations. Where parameters are uncertain and there is the possibility of loop-carried dependencies, their approaches must adopt a conservative schedule that assumes iterations contain recurrences. Our work overcomes this conservatism by selecting from different schedules at *runtime*, when the values of all parameters are known.

B. Polyhedral-based Transformation for Loop Pipelining

Among other recent efforts to optimize loop pipelining for HLS, polyhedral analysis has frequently been used. Morvan *et al.* [11] propose a method using polyhedral analysis to improve nested loop pipelining. To overcome conflicts of memory dependencies in a pipeline, their approach firstly flattens the nested loop and then inserts wait states ('bubbles') to resolve memory conflicts. However, their bubble insertion requires that there is no conflict of memory dependencies in the innermost loop. Unlike their approach, our optimization can be applied at the innermost loop level, and it is developed for the nested loops with uncertain and/or non-uniform memory dependencies. Li *et al.* [12] introduced an index-set splitting technique on top of classical affine loop transformations [13] to improve inner-loop parallelism. The index-set splitting is used for non-uniform memory dependencies and limited memory ports. In the first case, their approach directly separates the innermost loop into several sub-loops according its dependence patterns. Then, fast pipelining is applied on those sub-loops without any dependency. Similarly, in the second case, the parallel innermost loop is split into sub-loops according to different memory port conflict properties. The generated sub-loops can be pipelined at the inner loop with the best possible parallelism, so that the execution speed of the entire loop will not be limited by the worst property. However, our splitting technique is different from separating out loop iterations with irregular memory dependencies, because the purpose of our splitting is to insert the pipeline breaks for resolving memory conflicts when necessary. After our fine-grained transformation, we could apply fast pipelining on all the sub-loops split from the original loop.

C. Irregular Loop Pipelining

Besides regular loop structures, there are active HLS research efforts investigating pipelining for loops with irregular Tan *et al.* [14] describe an approach called ElasticFlow to apply loop pipelining on a class of irregular loop nests. In their proposed pipeline architecture, multiple pipeline instances of dynamic-bound inner loop are scheduled to execute in parallel, so that this approach prefers no inter-iteration dependencies in the outer loops. Our work is targeted to a different set of irregular loops from those of ElasticFlow, where we improve the pipeline parallelism by analysing inter-iteration dependencies. Alle *et al.* [15] implement a compilation method that transforms loops with dynamic data dependencies into specialized pipeline architectures. They add disambiguation logic in the hardware pipelines that can fully analyse the inter-iteration dependency at runtime. Dai *et al.* [16] propose the integration of a template hazard resolution unit in HLS to resolve runtime conflicts on memory ports and data dependencies caused by indirect or conditional memory accesses. The pipeline is executed speculatively with a small initiation interval, and it will replay some iterations when a memory conflict is detected. For both [15], [16], the hardware complexity of the runtime detection circuits is proportional to the number of dependent memory accesses and the depth of the pipeline stages. Although these techniques are also able to optimize our target loops, we apply more comprehensive static analysis to generate efficient and lightweight logic to control the pipeline execution at runtime.

D. Polyhedral Model for Memory Optimisation

Polyhedral optimization has been widely studied as an optimization tool for modern software compilers [17], [18]. In recent years, it has also been applied for optimizing custom memory systems of loop programs in HLS research. Liu *et al.* [19] proposed a mathematical formalization and an algorithm to implement minimized on-chip data reuse buffers in FPGA designs. Considering SDRAM as the off-chip memory in a common FPGA system, Bayliss *et al.* [20] implemented a polyhedral tool to produce address sequencers for an SDRAM interface to optimize off-chip memory bandwidth. Pouchet *et al.* [21] proposed another automated polyhedral HLS framework for better data reuse that combines loop transformations. Wang *et al.* [22] introduced a polyhedral theory and algorithm for generalized memory partitioning.

These previous works apply polyhedral analysis to study memory reuse or partitioning problems for improving loop latency and parallelism. In this work, we focus on developing a new parametric polyhedral analysis for both uncertain and non-uniform memory dependencies, enabling us to pursue aggressive loop pipelining that is not yet possible in modern HLS tools.

E. Extending the Polyhedral Model for Software Compilation

Polyhedral analysis and transformations can realize powerful optimizations because the underlying loop manipulation can be precisely expressed by algebraic representations. Unfortunately, using the polyhedral model also restricts the input program to be statically analysable. To overcome the limitation of static analysis, there are several ways to extend the applicability of polyhedral model. In [23], Benabderrahmane *et al.* developed an approach that extends polyhedral expressibility by over approximation. Predication statements are used to handle non-affine conditionals and loop bounds, so that general programs can be converted into the polyhedral model for analysis and transformations. To eliminate pessimistic conservatism, some approaches leverage runtime information to enable dynamic exploitation of loop parallelism. Jimborean *et al.* [24] and Sukumaran-Rajam *et al.* [25] have developed comprehensive speculative execution frameworks, where polyhedral transformations are used to promote parallelism at runtime. Alternatively, Venka *et al.* [26] proposed to use a dedicated inspector to support non-affine transformations at runtime. Index arrays in loop bounds and memory accesses are represented by uninterpreted functions in static analysis, and both affine and non-affine loop transformations are composed to increase loop parallelism effectively. More recently, loop versioning using polyhedral techniques is shown to improve compiler-based loop transformations with low overhead. Doerfert *et al.* [27] have developed a framework to derive a minimized set of preconditions, so that a variety of complex loop transformations can be enabled at runtime according to the validation of these preconditions. Similarly, Sampaio *et al.* [28] proposed a quantifier elimination scheme that combines static test generation and runtime evaluation to trigger appropriate loop transformations.

III. Motivation

A. Loop Pipelining

Loop pipelining is implemented by overlapping the execution of loop iterations. The logical operations within successive loops are mapped to hardware resources. The mapping must ensure that each hardware resource only executes one operation in each clock cycle. Where read-after-write loop-dependencies exist in the original code (a value is written in one iteration and read in a subsequent iteration), a static pipeline schedule must be constrained to preserve these dependencies. The constant interval between the start of successive iterations is called the *initiation interval* (II), and reflects the degree of parallelism, in the sense that for the same latency, a pipeline with smaller II has more iterations running in parallel at any given clock cycle.

If we denote the latencies of the operations executed before the loop body and of a single loop iteration by L_{pre} and L_{iter} respectively, and the loop trip count is N, then the latency of the entire loop is equal to

$$L_{pre} + L_{iter} + (N-1) \times II. \tag{1}$$

When N is large enough, this latency is approximately equal to $N \times II$. Therefore, the performance of a loop is mainly determined by its II.

To achieve a small II for loop pipelining, HLS tools need to solve complex scheduling problems [9], [10]. Unlike resource constraints that may vary with the requirements of different hardware implementations, iteration-dependency constraints are quite intrinsic. A complex dependency constraint could significantly constrain our ability to reduce the II of a loop pipeline.

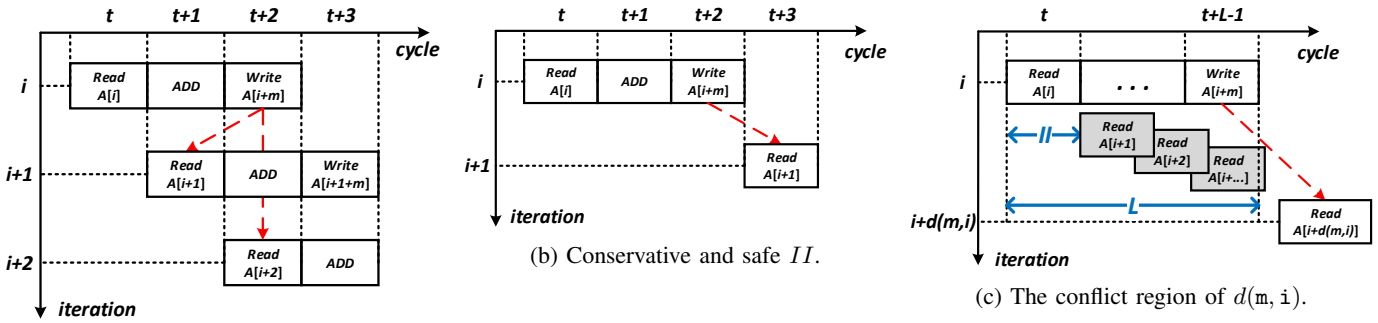
As an example, Fig. 3(a) and Fig. 3(b) illustrate two potential loop pipelining strategies with fixed IIs for the motivational loop shown in Listing 1. We still assume that the latency of each iteration is 3 clock cycles. Since there is an uncertain variable in the memory access pattern, the write reference of array `A` in the current iteration may be the same as the read reference of array `A` in a future overlapped iteration, which happens in Fig. 1(b). Due to this uncertain memory dependency, the best case of loop pipelining shown in Fig. 3(a) cannot be achieved with modern HLS tools. Current HLS tools will choose a conservative solution, in order to ensure correctness for all cases. We show this conservative solution in Fig. 3(b). Here, the possibility of a memory dependency between successive iterations prevents any loop pipelining with a fixed II.

B. Memory Dependence Analysis

With our parametric polyhedral analysis in Section IV, we wish to formally describe the memory dependencies in a nested loop so that we can determine when and in which pattern memory conflicts may happen. Here, we present an intuitive illustration of the analysis process with the one-dimensional loop shown in Listing 1. This process will be generalised and formalised in Section IV.

To analyse the memory dependencies of a loop, we need to formally model the memory access sequence. These patterns are described by array indexing functions and loop bounds, in which parameter (uncertain) variables may participate. We denote the vector of parameter variables by p. The loop bounds determine an iteration space for all memory accesses inside the loop. The dimension of the iteration space is equal to the number of loop iterators. Affine indexing functions map the iteration vectors v from the iteration space to the elements of each array in the loop. For example, $p = [\mathtt{m}]$ and $v = [\mathtt{i}]$ in Listing 1.

For each separate array from the source code, we can form two sets of indexing functions, one containing all the read accesses and the other all the write accesses. The Cartesian product of these two sets is a set of paired indexing functions. Two paired accesses are dependent if and only if the address written in the current iteration will be read in a future iteration.



(a) Unsafe II.

(b) Conservative and safe II.

(c) The conflict region of $d(\mathtt{m},\mathtt{i})$.

Fig. 3: Pipelining strategy for the loop shown in Listing 1.

The *dependence iteration distance* $d(p,v)$ is the smallest number of iterations between the execution of two such dependent data accesses, which can be derived from their affine indexing functions. Since the dependence iteration distance may vary in our target loops, we can evaluate the *conflict region* of $d(p,v)$, which will lead a read access to run before the completion of its dependent write access during the pipeline execution.

As shown in Fig. 3(c), we have $d(\mathtt{m},\mathtt{i})$ as the dependence iteration distance for the loop shown in Listing 1. The red dashed arrow indicates that the write access `A[i+m]` from iteration i has its first dependent read access `A[i+`$d(\mathtt{m},\mathtt{i})$`]` running at iteration $\mathtt{i} + d(\mathtt{m},\mathtt{i})$. To analyse this memory dependency, we can obtain $d(\mathtt{m},\mathtt{i}) = \mathtt{m}$. According to the given loop scheduling, the latency L is the period when the execution of the dependent read access will violate the inter-iteration memory dependency. In other words, `A[i+`$d(\mathtt{m},\mathtt{i})$`]` cannot be any grey read access shown in Fig. 3(c). If the target initiation interval is equal to II, there will be $\lceil \frac{L}{II} \rceil$ iterations being processed in the pipeline during the latency L. Thus, we could derive the cases in which the dependent read access will be executed in this period under the current pipeline schedule. In these cases, $d(\mathtt{m},\mathtt{i})$ will satisfy the conditions in (2), which denotes its *conflict region*.

$$1 \leq d(\mathtt{m},\mathtt{i}) \leq \left\lceil \frac{L}{II} \right\rceil - 1 \tag{2}$$

Intuitively, when $d(\mathtt{m},\mathtt{i})$ does not satisfy these conditions, no memory conflicts will happen in the given pipeline schedule. There will be either no memory dependency between a write and a future read (such as Fig. 1a) or enough iterations between them (such as Fig. 1c). According to Fig. 3(c), we obtain the conflict region as $1 \leq \mathtt{m} \leq 2$ based on (2), where $L = 3$ and $II = 1$.

C. Proposed Loop Transformation

In current HLS tools, only the worst case of uncertain and non-uniform memory dependencies is considered for loop pipelining. This leads a static pipeline schedule to have a large and conservative II. As illustrated in Listing 3, we propose a source-to-source code transformation, which will guide HLS tools to implement the pipeline as shown in Fig. 4.

Before the loop starts, the conflict region is firstly evaluated by the if-condition derived from (2). These conditions will

```
// Conflict region detection
if ( m >= 1 && m <= 2) {
  // Split execution
  for (k=0; k<N; k=k+m)
    // inner loop: force pipelining with II=1
    for (i=k; i<=min(N-1,k+m-1); i++)
      A[i+m] = A[i] + 0.5f;
}
else {
  // Fast execution
  // force pipelining with II=1
  for (i=0; i<N; i++)
    A[i+m] = A[i] + 0.5f;
}
```

Listing 3: Source-to-source code transformation of the motivational loop shown in Listing 1.

be synthesised into lightweight detection logic by HLS. The output of this detector will enable different pipeline execution modes. When the conflict region is not satisfied, the loop will be executed in the else-branch which is realised as a pipeline with $II = 1$. Otherwise, the loop will be executed with pipeline breaks in the then-branch. The pipeline breaks are realised by inserting a loop dimension outside the original loop. The step size of the new outer loop is determined by the dependence iteration distance $d(\mathtt{m},\mathtt{i}) = m$. The inner loop, which is the original loop, is also forced to be scheduled with $II = 1$. Like the runtime execution shown in Fig. 2, the split controller will still run the loop in a fast speed but pause the pipeline input after every `m` iterations are issued. The analysis can prove that there will be no memory conflict because the data written within the inner loop will be read only after the pipeline break.

In Fig. 4, the data paths of both execution modes are all statically scheduled with the smallest II. The related HLS directives (pragmas in Vivado HLS) are inserted in the real code. Their associated address generators (Addr Gen) are in charge of calculating array indices. Although the hardware logic appears to be duplicated in the pipeline body, they are in different branches of an if-condition. We therefore let the HLS tools decide how to exploit resource sharing for better timing or less resource overhead in the physical implementation. This also makes the transformation unrelated to any specific code tuning for resource sharing but flexible to support different HLS tools.

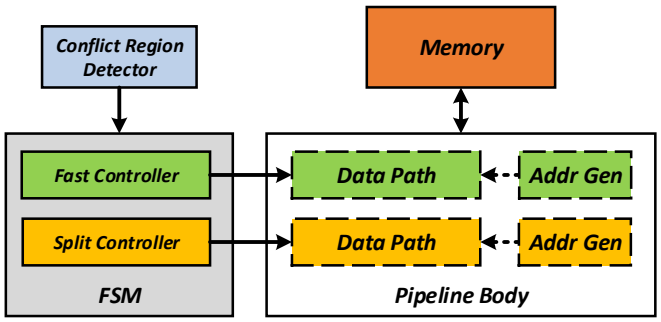


Fig. 4: Conceptual pipeline architecture.

IV. Polyhedral Optimization

In this section, we firstly give an introduction to our polyhedral model formulation in Section IV-A. After the introduction, we further formulate the parametric polyhedral analysis of the memory dependences in Section IV-B. To generate the conflict region, we present an algorithm based on the analysis in Section IV-C. Then in Section IV-D, we propose a polyhedral transformation for loop splitting. Finally, a source-to-source code transformation framework is introduced in Section IV-E, which is created to integrate our optimization method into existing HLS tools.

A. Preliminaries

The input of our analysis is a nested loop with d_I dimensions. In previous sections, we use the one-dimensional loop shown in Listing 1 to illustrate the idea of our analysis and transformation. In modern HLS tools like Vivado HLS, loop flattening (also known as loop coalescing) is enabled before the pipeline scheduling by default [1]. This transforms a multi-dimensional loop into a single-dimensional loop, so that the entire nested loop can be pipelined to achieve better throughput than just pipelining the innermost loop. Therefore, we aim to optimize the pipelining of the entire nested loop that will be flattened in the HLS backend. The theoretical background of our parametric polyhedral analysis can be found in [29]. Beyond the previous work, our optimizations in this paper are developed with the specific use of parameter properties in the polyhedral model.

In a given nested loop, there are N_{pair} pairs of memory accesses visiting the same arrays. Our analysis is described below as capturing RAW memory conflicts, but can also support other memory dependencies. The undetermined variables in the memory accesses can be represented by a parameter vector $p \in \mathbb{P}$, where $\mathbb{P} \subseteq \mathbb{Z}^{d_P}$ represents potentially known ranges of these variables and d_P is the number of undetermined variables. It is noteworthy that a parameter can also be an indirect array access whose index is not related to any induction variable of the given loop nest. If such indirect array access can be profiled statically, we can obtain its $\mathbb{P}$ to have a more accurate analysis. In this paper, we use the superscript p to indicate a dependence on parameters.

```
for (i = 0; i < 100; i++)
  for (j = 0; j < 2; j++)
    A[2*i+m][j] = A[i][j] + 0.5f;
```

Listing 4: The nested loop used as a walk-through example

Definition 1 (Iteration Domain)**.** *Given a d_I-dimensional loop nest, the iteration domain $\mathcal{D}^p$ is a parametric set of vectors of the form:*

$$\mathcal{D}^p = \left\{ v \in \mathbb{Z}^{d_I} \;\middle|\; Ax \leq b \text{ where } x = [v^T, p^T]^T \right\},$$

where v is the iteration vector of d_I induction variables. The inequality system represents the bounds for all loop levels, where A is a rational matrix and b is a rational vector.

The nested loop shown in Listing 4 is a walk-through example, where `m` is the undetermined variable appearing in write access of array `A`. This loop has an uncertain and non-uniform memory dependency. We have $x = [\mathtt{i}, \mathtt{j}, \mathtt{m}]^T$, and the inequality constraint defining its $\mathcal{D}^p$ is shown below.

$$\underbrace{\begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \mathtt{i} \\ \mathtt{j} \\ \mathtt{m} \end{bmatrix}}_{x} \leq \underbrace{\begin{bmatrix} 0 \\ 99 \\ 0 \\ 1 \end{bmatrix}}_{b}.$$

Definition 2 (Lexicographic Order)**.** *Given two iteration vectors $v = [v_0, v_1, \ldots, v_{d_I-1}]^T$ and $v' = [v'_0, v'_1, \ldots, v'_{d_I-1}]^T$ in $\mathcal{D}^p$, $v' \succ v$ holds if and only if*

$$\exists 0 \leq i < d_I, \forall 0 \leq j < i, v'_i > v_i \wedge v'_j = v_j.$$

This lexicographic order on v represents the execution order of loop iterations, which means that v' is executed after v in the pipeline.

For example, we have $[0,1]^T \succ [0,0]^T$ and also $[1,0]^T \succ [0,1]^T$ in the two-dimensional loop shown in Listing 4.

Definition 3 (Array Indexing Expression)**.** *An array indexing expression $f(v,p)$ is an affine function that transforms the vectors from an iteration domain into the d_D-dimensional index of a data array. In this paper, $f(v,p)$ is assumed to take the general affine form:*

$$y = f(v,p) = A_f v + B_f p + c_f,$$

where $y \in \mathbb{Z}^{d_D}$ is a vector of data array indices, and $v \in D^p$. $A_f \in \mathbb{Z}^{d_D \times d_I}$ and $B_f \in \mathbb{Z}^{d_D \times d_P}$ are rational coefficient matrices, and $c_f \in \mathbb{Z}^{d_D}$ is a rational constant vector. This expression indicates which element of the data array is accessed at a given iteration.

For example, the memory write access `A[2*i+m][j]` in Listing 4 has an affine function of the array index vector $y = [2\mathtt{i} + \mathtt{m}, \mathtt{j}]^T$, where

$$\underbrace{\begin{bmatrix} 2\mathtt{i} + \mathtt{m} \\ \mathtt{j} \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}}_{A_f} \underbrace{\begin{bmatrix} \mathtt{i} \\ \mathtt{j} \end{bmatrix}}_{v} + \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{B_f} \underbrace{\begin{bmatrix} \mathtt{m} \end{bmatrix}}_{p}.$$

Definition 4 (Iteration Dependency Map)**.** *Given the k^{th} pair of write and read accesses to the same array, the iteration dependency map $\mathcal{Q}_k^p$ links write and read iterations in $\mathcal{D}^p$ such that*

$$\mathcal{Q}_k^p = \left\{ \begin{pmatrix} v \\ v' \end{pmatrix} \;\middle|\; \begin{array}{l} w_k(v,p) = r_k(v',p) \\ \wedge\, v' \succ v \wedge v \in \mathcal{D}^p \wedge v' \in \mathcal{D}^p \end{array} \right\},$$

where v and v' represent the source and sink iteration vectors respectively. $w_k(v,p)$ and $r_k(v',p)$ are the array indexing expressions of the dependent write and read accesses. The equality constraint, $w_k(v,p) = r_k(v',p)$, can be expanded into the form:

$$A_w v + B_w p + c_w = A_r v' + B_r p + c_r. \tag{3}$$

An element in $\mathcal{Q}^p$ indicates that two iterations access the same data element in $\mathcal{D}^p$, i.e. *signifies the possible presence of a read-after-write memory dependence.*

For example, the equality constraint of the iteration dependency map in Listing 4 is

$$\underbrace{\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}}_{A_w} \underbrace{\begin{bmatrix} \mathtt{i} \\ \mathtt{j} \end{bmatrix}}_{v} + \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{B_w} \underbrace{\begin{bmatrix} \mathtt{m} \end{bmatrix}}_{p} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{A_r} \underbrace{\begin{bmatrix} \mathtt{i'} \\ \mathtt{j'} \end{bmatrix}}_{v'}.$$

In practice, the equality constraint in $\mathcal{Q}_k^p$ may be piece-wise affine when there are conditions in loop bounds or around memory accesses. For space reasons, we assume the equality constraint is always affine in this paper, but our implementation also supports the piece-wise case.

B. Parametric Polyhedral Analysis

1) Memory conflicts in loop pipelining: As mentioned in Section III, the loop transformation is affected by both inter-iteration memory dependencies and pipeline scheduling. The information about pipeline scheduling is assumed to be available for our analysis. To formally evaluate memory conflicts in loop pipelining, we need to determine which iterations will violate memory dependencies.

Definition 5 (Conflict Domain)**.** *Given an iteration dependency map $\mathcal{Q}_k^p$, target initiation interval II and scheduled latency L_k between the k^{th} pair of dependent memory accesses, the conflict domain $\mathcal{S}_k^p$ is a parametric set of iteration vectors in $\mathcal{D}^p$ such that*

$$\mathcal{S}_k^p = \left\{ v \;\middle|\; \begin{array}{l} (\exists v', (v,v') \in \mathcal{Q}_k^p) \\ \wedge\, I^p\left(Z_k^p(v)\right) - I^p(v) \le \left\lceil \frac{L_k}{II} \right\rceil - 1 \end{array} \right\},$$

where

$$Z_k^p(v) = \textit{lexmin}\left(\left\{ v' \;\middle|\; (v,v') \in \mathcal{Q}_k^p \right\}\right), \tag{4}$$

which generates the lexicographically minimum point of v' linked to v based on the equality constraint (3) *in $\mathcal{Q}_k^p$, and*

$$I^p(v) = \#\left\{ u \;\middle|\; u \in \mathcal{D}^p \wedge v \succ u \right\}, \tag{5}$$

which counts the number of iterations that are executed before the iteration v. In general, $I^p(v)$ can be expressed as a parametric pseudo-polynomial, as known as an Ehrhart polynomial, that counts the integer points of a parametric polytope [30]. Similar to (2)*, the inequality condition checks the existence of memory conflicts based on given pipeline scheduling.*

As shown in Def. 5, the conflict domain $\mathcal{S}_k^p$ includes the iterations that will violate memory dependencies when the nested loop is flattened and pipelined with the target II. To include the dependencies implied from all pairs of dependent accesses, the global conflict domain, $\mathcal{S}_{\text{conf}}^p = \bigcup_{k=1}^{N_{\text{pair}}} \mathcal{S}_k^p$, is the union of all $\mathcal{S}_k^p$.

2) Constructing the conflict domain: In this work, we use the Integer Set Library (`isl`) [31] to construct and analyse the parametric polyhedral models. The general form of $I^p(v)$ is a parametric pseudo-polynomial, which are representable with `isl`. However, sophisticated analysis of parametric pseudo-polynomial is limited in `isl`. Intuitively, $I^p\left(Z_k^p(v)\right) - I^p(v)$ calculates the smallest number of iterations between v and v'. Morvan *et al.*[11] estimated the lower bound of counting iterations between v and $Z_k^p(v)$ to check pipeline legality for nested loops, but this bound was mentioned to be not always tight and is not parametric. As a compromise, we limit the form of $I^p\left(Z_k^p(v)\right) - I^p(v)$ to be an affine expression so that `isl` can be used to count iterations parametrically for sophisticated integer set analysis.

In our following implementation, the memory dependencies incurred by $(v,v') \in \mathcal{D}^p$ leads (5) to count the integer points in a rectangular subset of $\mathcal{D}^p$, such that

$$d_k(v) = I^p\left(Z_k^p(v)\right) - I^p(v) = s^T\left(Z_k^p(v) - v\right).$$

For a given value of v, $s^T v$ can be interpreted as the "time stamp" within the sequence of all loop iterations, at which the iteration v begins processing. To obtain $d_k(v)$, we firstly generate the difference of iteration vectors, which is $Z_k^p(v) - v$, with $\mathcal{Q}_k^p$.

Definition 6 (Dependence Difference)**.** *Given an iteration dependency map $\mathcal{Q}^p$, the dependence difference $\delta_k^p(v)$ is an affine expression indicating the vector difference from v to $Z_k^p(v)$ with the following form.*

$$\delta_k^p(v) = Z_k^p(v) - v = A_\delta v + B_\delta p + c_\delta, \tag{6}$$

where $A_\delta \in \mathbb{Z}^{d_I \times d_I}$ and $B_\delta \in \mathbb{Z}^{d_I \times d_P}$ are rational coefficient matrices, and $c_\delta \in \mathbb{Z}^{d_I}$ is a constant vector. It is noteworthy that $\delta_k^p(v)$ is always single-valued due to the lexicographic optimization applied to obtain $Z_k^p(v)$.

For example, we can derive the following dependence difference from the write and read accesses in Listing 4.

$$\underbrace{\begin{bmatrix} \mathtt{i'} - \mathtt{i} \\ \mathtt{j'} - \mathtt{j} \end{bmatrix}}_{Z^p(v) - v} = \underbrace{\begin{bmatrix} \mathtt{i} + \mathtt{m} \\ 0 \end{bmatrix}}_{\delta^p(v)} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}}_{A_\delta} \underbrace{\begin{bmatrix} \mathtt{i} \\ \mathtt{j} \end{bmatrix}}_{v} + \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{B_\delta} \underbrace{\begin{bmatrix} \mathtt{m} \end{bmatrix}}_{p}.$$

In general, $s = [s_0, s_1, \ldots, s_{d_I - 2}, 1]^T$ and s_j represents the number of iterations of the inner loops under the j^{th} dimension of the nested loop. Given a rectangular loop nest, we can calculate $s_j = \Pi_{i=j+1}^{d_I - 1} t_i$, where t_i represents the trip count of the i^{th} loop dimension. For the nested loop in Listing 4, we

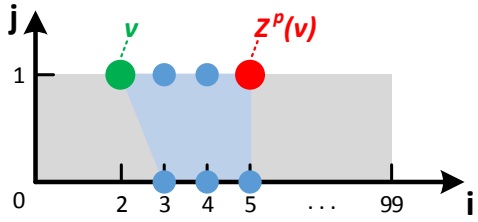


Fig. 5: The calculation of $d(v)$ for the example loop shown in Listing 4. Each dot is one integer point and represents one iteration. If we count after iteration v, there are 6 points including $Z^p(v)$ in the blue area when $v = [2,1]^T$ and $\mathtt{m} = 1$.

have $s = [2\times 1, 1]^T$, so that its $d(v) = s^T\delta^p(v) = 2\mathtt{i}+2\mathtt{m}$. An example of counting integer points is shown in Fig. 5. When $v = [2,1]^T$ and $\mathtt{m} = 1$, we have $d(v) = 2\times 2 + 2\times 1 = 6$, which gives the same counting result as illustrated in Fig. 5.

3) Supported cases: In this paper, we aim to analyse the nested loop with the iteration domain as defined in Def. 1. However, as mentioned before, our analysis requires that $d_k(v)$ needs to be an affine expression due to the limitation of `isl`. Since $d_k(v)$ is the dot product of s and $\delta^p_k(v) = \left[\delta^p_{k,0}, \delta^p_{k,1}, \ldots, \delta^p_{k,d_I-1}\right]^T$, we need to ensure that the product of $\Pi_{i=j+1}^{d_I-1} t_i$ and $\delta^p_{k,j}$ is also affine, where $0 \le j < d_I$. The nested loops supported by our analysis can have undetermined bounds (*i.e.* trip count t_i can be parametric) or non-rectangular iteration space (*i.e.* trip count t_i can vary with outer loop iterators). Here, we provide a summary of the supported cases of calculating $d_k(v)$.

- **Rectangular case.** If every loop level has a uniform trip count (which means that

$$d_k(v) = \left[\Pi_{i=1}^{d_I-1} t_i,\ \Pi_{i=2}^{d_I-1} t_i,\ \ldots,\ t_{d_I-1},\ 1\right]\delta^p_k(v)$$

holds), then there must be at most one dimension (say, j) with a parametric trip count, and the dependence difference at every dimension outside j must be constant (*i.e.*, $\forall 0 \le i < j$, $\delta^p_{k,i}$ is constant). An example is shown in the following loop,

```
for (i=0; i<10; i++)
  for (j=a; j<b; i++)
    A[i+2][j+m] = A[i][j] + 0.5f;
```

where $d(v) = [\mathtt{b}-\mathtt{a}, 1][2,\mathtt{m}]^T = 2\mathtt{b} - 2\mathtt{a} + \mathtt{m}$.

- **Non-rectangular case.** Otherwise, let j be the innermost level with a trip count varying with some outer loops. Then every level inside j must have a constant trip count (which means

$$d_k(v) = \left[\ldots, \Pi_{i=j}^{d_I-1} t_i,\ \Pi_{i=j+1}^{d_I-1} t_i,\ \ldots, t_{d_I-1}, 1\right]\delta^p_k(v)$$

holds), the dependence difference at level $j-1$ must be 0 or 1 (*i.e.*, $\delta^p_{k,j-1} \le 1$), and the dependence difference at every level outside $j-1$ must be 0 (*i.e.*, $\forall 0 \le i < j-1, \delta^p_{k,i} = 0$). An example is shown in the following loop,

```
for (i=0; i<10; i++)
  for (j=a; j<i; i++)
    A[i+1][j+m] = A[i][j] + 0.5f;
```

where $d(v) = [\mathtt{i}-\mathtt{a}, 1][1,\mathtt{m}]^T = \mathtt{i} - \mathtt{a} + \mathtt{m}$.

For unsupported cases, we can alternatively analyse the inner loops of the given nested loop. In the worst case, the innermost loop can always be analysed by our approach, because we have $d_k(v) = \delta^p_k(v)$ in a one-dimensional iteration domain.

Algorithm 1 Generate the conflict region

1: **Input:** loop iteration domain $\mathcal{D}^p$, dependent memory access pairs, pipeline schedule information, and target II
2: **Output:** $\mathcal{P}_{\text{conf}}$
3: $\mathcal{P}_{\text{safe}} \leftarrow \mathbb{Z}^{d_P}$
4: **for each** dependent memory access pair k **do**
5: $\quad\mathcal{Q}^p_k \leftarrow$ ComputeFlow$(w_k, r_k, \mathcal{D}^p)$
6: $\quad\mathcal{Q}'^p \leftarrow$ LexMin$(\mathcal{Q}^p_k)$
7: $\quad\text{MAff}_{\text{snk}} \leftarrow$ TakeMultiAff$(\mathcal{Q}'^p)$
8: $\quad\text{MAff}_{\text{src}} \leftarrow$ CreateMultiAff(v)
9: $\quad\text{MAff}_{\delta} \leftarrow$ SubMultiAff(MAff$_{\text{snk}}$, MAff$_{\text{src}}$)
10: $\quad\text{Aff}_d \leftarrow$ ComputeIterDist(MAff$_{\delta}$, $\mathcal{D}^p$)
11: $\quad\mathcal{S}^p_k \leftarrow$ CreateConflictSet$(L_k, II,$ Aff$_d)$
12: $\quad\mathcal{P}_k \leftarrow$ LexOpt$(\mathcal{S}^p_k = \emptyset)$
13: $\quad\mathcal{P}_{\text{safe}} \leftarrow \mathcal{P}_{\text{safe}} \cap \mathcal{P}_k$
14: **end for**
15: $\mathcal{P}_{\text{conf}} \leftarrow \mathbb{Z}^{d_P} \setminus \mathcal{P}_{\text{safe}}$

C. Conflict Region Generation

Following the analysis in Section III-B, we generate the conflict region by constructing its complement set in $\mathbb{Z}^{d_P}$, *i.e.* the *safe region*. The safe region of the k^{th} pair of dependent memory accesses is a set of parameter vectors $\mathcal{P}_k \subseteq \mathbb{P}$ such that

$$\mathcal{P}_k = \left\{p \in \mathbb{P} \mid \mathcal{S}^p_k = \emptyset\right\},$$

which includes all possible parameter values that make the conflict domain empty. The global safe region $\mathcal{P}_{\text{safe}} = \bigcap_{k=1}^{N_{\text{pair}}} \mathcal{P}_k$, is the intersection of all local safe regions, and allows conflict-free pipelining of the entire loop nest.

The main algorithm for generating the conflict region for a given nested loop is described in Algorithm 1, where we simplify many operations of the `isl` library into abstract functions. This algorithm requires a given pipeline scheduling and a target II that is relatively small for the short execution time of the input loop. The iteration domain $\mathcal{D}^p$ is also extracted from the loop beforehand. First, the global safe region $\mathcal{P}_{\text{safe}}$ is initialized as $\mathbb{Z}^{d_P}$ and further constrained as the algorithm progresses. Next, in the for-loop (lines 4-14), we analyse all pairs of possible dependent memory accesses labelled with k.

In lines 5-6, the iteration dependency map $\mathcal{Q}^p_k$ is generated and simplified. Function ComputeFlow$(w_k, r_k, \mathcal{D}^p)$ is to create the equality constraint in (3), which will map the write access (source) to the read accesses (sink) visiting the same data point. Both write and read iterations should satisfy that $v \in \mathcal{D}^p \wedge v' \in \mathcal{D}^p$. We only need to check the dependency of the read access in the earliest sink iterations. Therefore, function LexMin is applied to filter out the lexicographically

minimal sink iteration, which is equivalent to $Z_k^p(v)$ in (4), so that a simplified dependency map is assigned to $\mathcal{Q}'^p$.

The dependence difference $\delta_k^p(v)$ is generated in lines 7-9. From $\mathcal{Q}'^p$, a multi-affine function MAff_{snk} is extracted in line 7 to represent the multi-dimensional sink iteration corresponding to $Z_k^p(v)$, which is a function of source iteration v and parameter vector p. With the multi-affine function MAff_{src} created for v in line 8, MAff_δ is the subtraction in line 9 between MAff_{snk} and MAff_{src}, which produces the lexicographically smallest vector difference as the form shown in (6). In line 10, Aff_d is computed as the affine expression $s^T\delta_k^p(v)$, which is equivalent to (5) counting the integer points in a rectangular space. At this step, the trip counts of all loop dimensions are also derived according to the shape and size of $\mathcal{D}^p$. Then in line 11, we construct the conflict domain $\mathcal{S}_k^p$ where L_k is the scheduled latency between two dependent memory accesses in each iteration, similar to the latency L in Fig. 3(c).

To derive the safe region $\mathcal{P}_k$, we use `isl` to apply a lexicographic optimization over $\mathcal{S}_k^p$. In line 12, the function $\text{LexOpt}(C_{k,l}^p = \emptyset)$ can efficiently generate the constraints that specify which parameter values make $\mathcal{S}_k^p$ always empty. The global safe region P_{safe} of the input loop is its intersection with all P_k as shown in line 13. Finally, the global *conflict region* $\mathcal{P}_{\text{conf}}$ is obtained by $\mathbb{Z}^{d_P}$ minus P_{safe}. Similar to (2), the constraints in $\mathcal{P}_{\text{conf}}$ will be used as a if-condition and synthesised into the lightweight detection logic in Fig. 4.

D. Polyhedral Transformation for Loop Splitting

According to the memory dependency analysis, the iteration domain $\mathcal{D}^p$ is partitioned based on the conflict domain and dependence difference, when $p \in \mathcal{P}_{\text{conf}}$. In general, our transformation modifies the model of static control parts (SCoP) [32] that represents the loop programs.

1) Determining the conflict dimension: In order to determine which dimension of a given nested loop should be split, each non-zero dependence difference δ_k^p is assessed to locate the conflict dimension, denoted q, as follows. For each pair of dependent memory accesses, we firstly locate its local conflict dimension i. This is the outermost loop level causing memory dependency through this pair of accesses, that is:

$$\delta_{k,i}^p \neq 0 \text{ and } 0 \leq \forall j < i, \delta_{k,j}^p = 0.$$

Then, the conflict dimension q is calculated as the largest local conflict dimension for all $1 \leq k \leq N_{\text{pair}}$. Therefore, when the loop nest is split at the q^{th} level, all potential memory conflicts should be resolved.

2) Splitting by conflict domain: The first stage is to apply fast pipelining on the iterations outside the conflict domain when $p \in \mathcal{P}_{\text{conf}}$. This is realised by partitioning the iteration domain into three sub-domains at the conflict dimension. We use lexicographic optimizations to generate the first and last iterations that have the write accesses initiating memory conflicts:

$$l^p = \text{lexmin}(\mathcal{S}_{\text{conf}}^p), \text{ and } u^p = \text{lexmax}(\mathcal{S}_{\text{conf}}^p).$$

In particular, we take the q^{th} elements of l^p and u^p as the split points at the conflict dimension. These two parametric

```
for (k=0; k<N; k++){
  for (i=k; i<=min(N-1,k+m-1); i++)
    A[i+m] = A[i] + 0.5f;
  k=k+m-1;    }
```

Listing 5: Alternative form of block-wise splitting

elements are represented as l_q^p and u_q^p, whose expressions may be piece-wise affine. The iteration domain $\mathcal{D}^p$ is partitioned along l_q^p and u_q^p such that

$$\mathcal{D}^p = \bigcup \begin{cases} \mathcal{D}_1^p = \{v \mid v \in \mathcal{D}^p \wedge v_q \leq l_q^p\} \\ \mathcal{D}_2^p = \{v \mid v \in \mathcal{D}^p \wedge l_q^p < v_q \leq u_q^p\} \\ \mathcal{D}_3^p = \{v \mid v \in \mathcal{D}^p \wedge u_q^p < v_q\} \end{cases} . \quad (7)$$

These three sub-domains correspond to three sub-loops executed in sequential order. The pipeline break points are introduced during the transitions between sub-loops, so that the sub-loops with $\mathcal{D}_1^p$ and $\mathcal{D}_3^p$ can be fully pipelined by ignoring loop-carried dependencies.

3) Splitting by dependence difference: The second stage is to insert pipeline break points in the sub-loop with $\mathcal{D}_2^p$, so that the pipeline can execute as many iterations in parallel as possible. These break points are created by splitting the q^{th} loop level block-wise, and fast pipelining is applied on the loop blocks. Correspondingly in $\mathcal{D}_2^p$, a new dimension is inserted between the $(q-1)^{\text{th}}$ and q^{th} dimensions to create a new iteration domain of the second sub-loop such that

$$\mathcal{D}'^p_2 = \left\{ v' \;\middle|\; \begin{array}{l} v' = [v_0, \ldots, v_{q-1}, v_{blk}, v_q, \ldots, v_{d_I-1}]^T \\ \wedge\, [v_0, \ldots, v_{q-1}, v_q, \ldots, v_{d_I-1}]^T \in \mathcal{D}_2^p \\ \wedge\, l_q^p < v_{blk} \leq u_q^p \wedge t_{blk} \mid (v_{blk} - l_q^p - 1) \\ \wedge\, v_{blk} \leq v_q < v_{blk} + t_{blk} \end{array} \right\}, \quad (8)$$

where v_{blk} is the induction variable of the inserted dimension, and t_{blk} represents the trip count of the conflict dimension. In general, t_{blk} is a piece-wise affine expression with the following form: $t_{blk} = \alpha^p v_{blk} + \beta^p p + \gamma^p$, where α^p, β^p and γ^p are piece-wise constant coefficients. We have $\mathcal{P}_{\text{conf}} = \bigcup_{i=1}^{N_{\text{piece}}} \mathcal{P}_{\text{conf}}^i$, and N_{piece} represents the number of disjoint parameter sets (*i.e.* pieces) in the conflict region. In different $\mathcal{P}_{\text{conf}}^i$, α^p, β^p and γ^p have different values. Thus, t_{blk} represents a union of minimum positive $\delta_{k,q}^p$ in all disjoint parameter sets, and for each $\mathcal{P}_{\text{conf}}^i$, it is derived by

$$\min \left\{ \delta_{k,q}^p \mid \delta_{k,q}^p > 0 \wedge p \in \mathcal{P}_{\text{conf}}^i \wedge 1 \leq k \leq N_{\text{pair}} \right\}.$$

The divisibility constraint, $t_{blk} \mid (v_{blk} - l_q^p - 1)$, can be represented by existential quantification in `isl` only when t_{blk} is a constant. To avoid this limitation, we insert one additional statement in our SCoP model to explicitly define the stride of the inserted loop level, which is equivalent to the divisibility constraint of v_{blk}. With this approach, the alternative form of the split execution shown in Listing 3 is illustrated in Listing 5, where induction variable `k` is incremented by $t_{blk} =$ `m` after the execution of the inner loop dimension.

E. Source-to-source Code Transformation

To prototype our new loop optimization and make it compatible with an HLS tool, we integrated our analysis algo-

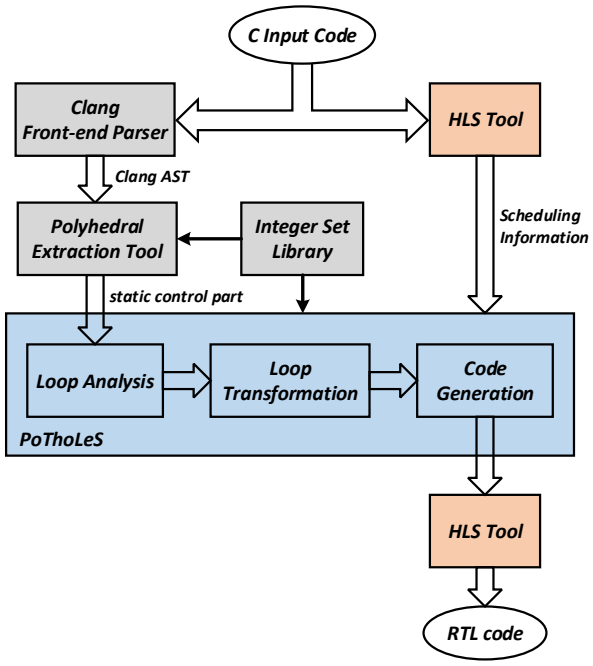


Fig. 6: Tool flow for code transformation framework.

```
// Original:
for (i = 0; i < 100; i++)
    for (j = 0; j < 2; j++)
    #pragma HLS PIPELINE
    A[2*i+m][j] = A[i][j] + 0.5f;

// Transformed:
if (m >= -97 && m <= 8){
    for (i = 0; i < -m + 1; i++)                                    sub-loop 1
        for (j = 0; j <= 1; j++)
            #pragma HLS PIPELINE
            #pragma HLS DEPENDENCE variable=A array inter false
            A[2*i+m][j] = A[i][j] + 0.5f;
    for (k = max(0, -m + 2); k <= -m + 8; k++){
        for (i = k; i < min(-m + 8, m + 2 * k - 1); i++)            sub-loop 2
            for (j = 0; j < 2; j++)
                #pragma HLS PIPELINE
                #pragma HLS DEPENDENCE variable=A array inter false
                A[2*i+m][j] = A[i][j] + 0.5f;
        k = k + (m + k - 1);
    }
    for (i = -m + 9; i < 100; i++)                                  sub-loop 3
        for (j = 0; j < 2; j++)
            #pragma HLS PIPELINE
            #pragma HLS DEPENDENCE variable=A array inter false
            A[2*i+m][j] = A[i][j] + 0.5f;
} else
    for (i = 0; i<100; i++)                                         fast loop
        for (j = 0; j < 2; j++)
            #pragma HLS PIPELINE
            #pragma HLS DEPENDENCE variable=A array inter false
            A[2*i+m][j] = A[i][j] + 0.5f;
```

Fig. 7: Demonstration of code transformation.

rithm into a source-to-source code transformation framework shown in Fig. 6. In this paper, we select Xilinx Vivado HLS, which generates hardware architectures from original and transformed C code, as the RTL generation back-end in our flow. The HLS tool is firstly used to synthesize the original loop without considering inter-iteration memory dependencies. The scheduling information for this pipeline is used for further analysis. Since Vivado HLS is a commercial tool, we can only use the tool as a black box without internal detailed scheduling information. This also means that our approach can be applied to other RTL generation back-ends. Currently, the achieved II is extracted from the first synthesis as the target II in Algorithm 1. We also extract the pipeline latency achieved from the first synthesis, and assign it to all L_k in Algorithm 1 instead of the scheduled latency between the k^{th} pair of write and read. This leads L_k to be an upper bound value, which has the potential to tighten the conflict region.

As shown in Fig. 6, the loop information is captured by two open-source tools. The Clang front-end parser [33] generates an abstract syntax tree (AST) from the input C code. The Polyhedral Extraction Tool (`pet`) [34] uses `isl` to extract the loops as the static control parts (SCoPs) from the Clang AST. Finally, the transformed C code is generated by PoTHoLeS [35]. PoTHoLeS is a polyhedral compilation tool developed by us, which conducts user-specified loop analysis, transformation and code generation based on `isl`. This tool is available in a public Github repository.[1]

In Fig. 7, we demonstrate the code transformation produced by our tool. The detection of the conflict region is realised by the outermost if-condition, which is generated by Algorithm 1. The fast loop in the else-case is same as the fast execution shown in Listing 3. The then-case includes three sub-loops split from the original loop according to (7), which will be synthesised into a split controller as shown in Fig. 4. Because the conflict domain of the original loop is parametrised, bounds of sub-loops contain `m` and code macros like `min()` and `max()`. From the original loop, the dependence difference is correctly recognised as `m+i`. Memory conflicts are only related to the iterator `i`, and thus our tool splits the original loop at the outer loop dimension. According to (8), a new loop level is inserted in sub-loop 2 with an induction variable `k`, which realises block-wise loop splitting. Since Vivado HLS cannot apply flattened pipelining on a nested loop with variable bounds, only the loop dimension inside the inserted one in sub-loop 2 can be pipelined. Therefore, we leverage this feature to implement block-wise loop pipelining.

In Vivado HLS, forcing resource sharing can be realised by replacing the duplicated loop bodies with the same function call and disabling the feature of function inlining. Such complementary transformation could be effective to reduce some resource overhead, but this may also sacrifice the sharing opportunities across the boundary of function calls. The design trade-off of resource sharing is both application and tool specific, which is out of the scope of this paper, and we leave it for future investigation.

V. Experimental Results

A. Experimental Setup

In this work, our code transformation framework uses Xilinx Vivado HLS 2017.2 as the RTL generation backend. The target FPGA device is a Virtex 7 XC7VX485T. In all experiments, the target clock period is set to 3ns, which is expected to produce a balanced trade-off between clock speed and resource usage. We export generated RTL codes to Xilinx Vivado Design Suite 2017.2 to collect clock and resource usage results after RTL synthesis, place and route. Furthermore, all generated pipelines are tested by C/RTL co-simulation with dedicated testbenches to confirm functional equivalence with the original code.

TABLE I. The details of the benchmark loops.

Benchmark	d_I	d_P	Non-uniform dependency	$\exists\mathcal{P}_{\text{conf}}$	Splitting stage		Trip Count in $\mathcal{P}_{\text{conf}}$
					by $\mathcal{S}^p_{\text{conf}}$	by $\delta^p_k(v)$	
dist_param	1	1	-	✓	-	✓	100
dist_itr	1	0	✓	-	✓	✓	100
dist_itr_param	2	1	✓	✓	✓	✓	200
row_col	2	2	-	✓	-	✓	$6 \sim 256$
pivot	2	2	-	-	-	-	-
tri_sp_slv	1	3	✓	✓	✓	-	$2 \sim 91$
adi_int	2	3	-	✓	-	-	≥ 3
floyd_warshall	3	0	✓	-	✓	-	2097152

B. Benchmarks

We choose eight benchmark loops used in our previous works [5], [6] for our experimental study. These benchmarks reflect some typical uncertain and non-uniform memory dependencies, which are usually not covered in a full benchmark suite like Polybench [36]. All memory arrays contain single precision floating point numbers. All uncertain variables are `int` values, *i.e.* lie between `INT_MIN` and `INT_MAX` as defined in `<limits.h>`. The source code of benchmarks, testbenches, and their transformation used in the experiments are available in a public Github repository.[2]

The details of the benchmark loops are summarised in Table I. The motivational loops shown in Listings 1, 2 and 4 are named **dist_param**, **dist_itr** and **dist_itr_param** respectively. The remaining benchmarks are derived from real applications and other publicly available benchmarks. **row_col** is a 2D loop having the same inter-iteration dependency as the example loop shown on page 151 of the user guide of Xilinx Vivado HLS 2017.2 [1]. **pivot** is a 2D loop extracted from the forward reduction step (line 208) in the Gaussian elimination with pivoting code [37]. **tri_sp_slv** is a 1D loop obtained from a triangular sparse matrix solver, which has one undetermined iteration causing a memory conflict. **adi_int** is a 2D loop from Kernel 8 in the Livermore benchmark suite [38]. **floyd_warshall** is a 3D loop for finding shortest paths from Polybench [36], which has one fixed iteration causing a memory conflict in its innermost loop.

C. Performance Improvement

As shown in Table I, various memory dependence patterns of the benchmarks lead to different optimisation strategies applied in the transformation. One special case is found in **pivot**, where the analysis guarantees that the loop can be always executed in the fast pipeline. In addition, the conflict dimension q of **adi_int** is at the innermost loop where $\delta^p_{k,q}$ is found to be 1, and thus it is not necessary to split this loop in its conflict region.

Table II provides the detailed results of pipeline performance. In this table, columns with the title "Orig" indicate characteristics of the original pipeline and columns with the title "Tran" indicate characteristics of the transformed pipeline implementation. Columns with the title "Fast" indicate the pipeline performance achieved when the generated lightweight checks determine lower initiation intervals are safe. Columns with the title "Split" indicate the performance when the pipeline breaks have to be inserted to avoid memory conflict. Furthermore, "Pre-Loop Cycles" represents the number of cycles executed before the start of loop body (L_{pre}) and "Iteration Cycles" represents the number of cycles for one loop iteration (L_{iter}).

[2]https://github.com/Junyi-Liu/benchmarks-HLS/tree/master/PolyDLP. The corresponding commit hash is `40e7e91`.

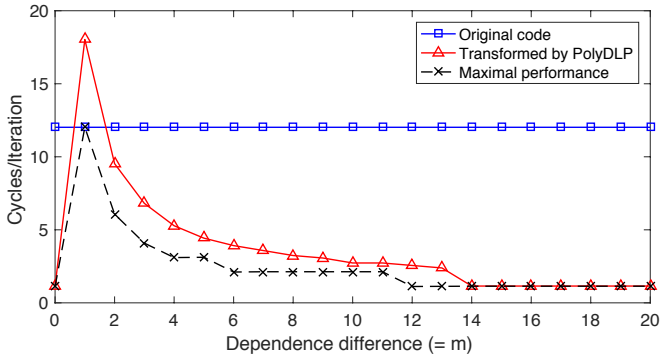


Fig. 8: The runtime performance evaluation of the transformed pipeline of **dist_param** as shown in Listing 1.

Following the architecture of the transformed pipeline shown in Fig. 4, the detector logic should be executed before the start of the loop body. These additional operations are observed to double L_{pre} on average, but they only cost a few cycles to finish. This indicates that the complexity of the detector logic is lightweight. L_{iter} is also slightly increased to support higher parallelism during the pipeline scheduling. Since the latency of executing the entire loop is calculated by (1), the impact of L_{pre} and L_{iter} is often negligible, especially when the loop body has a large trip count N.

After our proposed transformation, almost all the nested loops or sub-loops can be safely pipelined by the HLS backend tool without considering any inter-iteration dependency. Across our benchmarks, this allows HLS scheduling to achieve much smaller II ranging from just 1 to 3 cycles in the *fast* mode. These achieved IIs lead to $10\times$ higher peak performance of the transformed pipelines. To evaluate runtime performance of the benchmark loops in their conflict region, we measured loop execution latency with further experiments in RTL co-simulation. For each loop with uncertain memory accesses, we generated 100 test cases with random values of parameters that were ensured to be within the conflict region. The random tests already cover all combinations of the parameters in the conflict region. For each test with each benchmark, we also collected the corresponding loop trip count and execution latency in clock cycles. Their tested trip counts are also summarized in Table I. The average cycles per iteration in Table II shows a $3.7\times$ speed-up of the pipeline throughput in the conflict region.

D. Analysis of runtime performance

According to Table I, when the loops are split by conflict domain ($\mathcal{S}^p_{\text{conf}}$), the transformed loops have an average cycles/iteration close to II, as shown in Table II. In particular, the second sub-loops of **tri_sp_slv** and **floyd_warshall** are empty, so that there is no further splitting by dependence

TABLE II. The improvement of pipeline performance.

Benchmark	Pre-Loop Cycles			Iteration Cycles			Initiation Interval				Avg. Cycles/Iter		
	Orig	Tran	ratio	Orig	Split	Fast	Orig	Split	Fast	ratio	Orig	Tran	ratio
dist_param	1	2	2.00	12	14	14	12	1	1	0.08	12.0	5.7	0.48
dist_itr	1	1	1.00	14	14	-	14	1	-	-	14.0	1.8	0.13
dist_itr_param	1	8	8.00	15	17	17	6	1	1	0.17	6.1	1.7	0.29
row_col	8	9	1.13	15	15	17	12	2	2	0.17	12.2	5.5	0.45
pivot	4	7	1.75	49	-	55	47	-	3	0.06	-	-	-
tri_sp_slv	1	3	3.00	22	31	30	18	2	2	0.11	18.4	5.2	0.28
adi_int	3	9	3.00	63	65	68	52	52	3	0.06	-	-	-
floyd_warshall	1	1	1.00	18	20	-	14	2	-	-	14.0	2.3	0.16
Geomean			2.03							0.10			0.27

TABLE III. Timing and resource overheads of the proposed transformation.

Benchmark	Clock (ns)				LUT				FF				DSP48E1				Area-Time Product*		
	Orig	HP	Tran	ratio	Orig	HP	Tran	ratio	Orig	HP	Tran	ratio	Orig	HP	Tran	ratio	Orig	Tran	ratio
dist_param	2.02	2.02	2.32	1.15	239	268	487	2.04	340	425	595	1.75	2	2	2	1.00	5.80	6.46	1.11
dist_itr	2.02	2.02	2.33	1.15	230	242	400	1.74	405	417	623	1.54	2	2	2	1.00	6.51	1.68	0.26
dist_itr_param	2.72	2.72	2.72	1.00	401	382	1214	3.03	454	538	1209	2.66	3	3	4	1.33	6.59	5.75	0.87
row_col	2.39	2.42	2.59	1.09	809	827	988	1.22	1108	1255	1392	1.26	8	8	8	1.00	23.59	14.19	0.60
pivot	2.32	2.27	2.32	1.00	1409	1441	1435	1.02	2324	2495	2582	1.11	7	7	8	1.14	-	-	-
tri_sp_slv	3.02	3.01	3.04	1.01	479	501	892	1.86	705	803	1106	1.57	6	6	6	1.00	26.66	14.03	0.53
adi_int	3.35	3.03	2.64	0.79	1202	2691	4328	3.60	1603	4336	5301	3.31	11	23	21	1.91	-	-	-
floyd_warshall	2.28	2.28	2.90	1.28	477	542	787	1.65	713	862	1092	1.53	2	2	2	1.00	15.20	5.25	0.35
Geomean				1.05				1.87				1.73				1.14			0.55

* Area-Time Product = LUT number × Clock (us) × Avg. Cycles/Iter

difference ($\delta_k^p(v)$) in these loops. The transformed **tri_sp_slv** has a relatively larger cycles/iteration due to its undetermined loop bounds. When the trip count is too small, the runtime performance of the transformed pipeline cannot benefit from the improved parallelism.

For **dist_param** and **row_col**, the entire loop can be treated as sub-loop 2. In these benchmarks, only splitting by dependence difference is applied, and pipeline performance changes with the parameters determined at runtime. We further evaluated the runtime performance of **dist_param** to illustrate the speed-up of this splitting stage. Fig. 8 compares two types of pipeline architectures, where our polyhedral-based dynamic loop pipeline is denoted by PolyDLP. We also collected the maximal performance of **dist_param** at different values of $\mathtt{m}$, which is plotted as a dash line. Each point of the maximal performance is collected from the pipeline synthesised from the loop whose $\mathtt{m}$ is replaced by a constant value. The behaviour of these pipelines is only correct for their specific values of $\mathtt{m}$, so that their performance represents the upper bound of the runtime parallelism. The conflict region of **dist_param** is $1 \leq \mathtt{m} \leq 13$, where the transformed pipeline have the dynamic breaks inserted at runtime. When $\mathtt{m}$ is 1, all iterations have to be executed sequentially. Due to extra operations added to support PolyDLP, the transformed pipeline is slower than the original one only in this case. When $\mathtt{m}$ becomes larger, there are fewer break points inserted in the pipeline execution, and its runtime performance becomes much closer to the maximal one. When the loop is executed in the fast mode (where $\mathtt{m} = 0$ or $\mathtt{m} \geq 14$), the transformed pipeline can achieve the maximal performance as expected. Therefore, our static analysis and transformation allows the pipeline to dynamically adjust its throughput to avoid any memory conflict.

E. Timing and Resource Overhead

As shown in Table III, our transformation has very little impact on the achievable clock period, but it generally increases the hardware resource usage to achieve higher parallelism. We also evaluate the design choice of the highest pipeline parallelism, which is obtained by synthesising the original loop without considering any inter-iteration dependency. Its results are shown under the columns with the title "HP", which helps us to better understand the effect of resource sharing. After our transformation, the average increase of Look-up Tables (LUTs), Flip-Flops (FFs) and DSP blocks is 87%, 73% and 14% respectively. However, resource overhead is still less significant than performance improvement even in the conflict region, as witnessed by a 45% average reduction of the area-time product.

Due to higher parallelism achieved after the transformation, more operations are required to work at the same time in the pipeline bodies shown in Fig. 4. Firstly, besides the detector logic and the more complex finite state machine, the increase of LUTs and FFs is mainly caused by the unshared address generators. These addressing logic mainly consists of integer arithmetic operators such as adders and multipliers. Their implementation in our relatively small benchmarks will cause little resource pressure for modern high-density FPGAs. Thus, the HLS backend tends to duplicate these operators across mutually exclusive conditionals in favour of using fewer

multiplexers for less routing complexity. In order to eliminate some unnecessary duplication, resource constraints on integer multipliers have been added in **dist_itr_param** and **tri_sp_slv**, which does not affect other resources or timing. Secondly, the resource sharing between the floating-point data paths is well supported by the HLS backend. This can be observed by the small difference of DSP usages between "HP" and "Tran", and thus the increase of DSP blocks is mainly due to the higher parallelism of the data path.

VI. Conclusion

In this paper, we proposed a new optimization method for a class of loops with uncertain and non-uniform memory dependencies. The method combines compiler-based analysis and runtime optimization. The optimized pipelines can execute the loop iterations as fast as possible, when specific conditions are detected, or pipeline breaks are inserted at runtime. We formulate a general parametric polyhedral analysis and transformation for resolving complex memory conflicts in these pipelines. A source-to-source code transformation framework is prototyped for evaluating our propose optimisations. With experiments over a suite of benchmarks, we show that the transformed pipelines can achieve a 3.7-10$\times$ speed-up with a reasonable resource overhead. In future work, we intend to lift the restriction of affine expressions in the analysis, allowing for the better support of indirect memory accesses. Furthermore, the static pipeline scheduling for HLS can be co-optimised with our techniques to minimise the resource overhead and further improve the performance.

Acknowledgments

The support of the EPSRC grants EP/P010040/1, EP/I020357/1 and EP/K034448/1, the Royal Academy of Engineering, and Imagination Technologies is gratefully acknowledged. The data sets published in this article are available at http://dx.doi.org/10.5281/zenodo.1069695.

References

[1] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis*.

[2] Intel, *Intel FPGA SDK for OpenCL Programming Guide*.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[4] P. Quinton and V. Dongen, "The mapping of linear recurrence equations on regular arrays," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 1, no. 2, pp. 95–113, 1989.

[5] J. Liu, S. Bayliss, and G. A. Constantinides, "Offline synthesis of online dependence testing: Parametric loop pipelining for HLS," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2015, pp. 159–162.

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

[7] M. Lam, "Software pipelining: An effective scheduling technique for vliw machines," *SIGPLAN Not.*, vol. 23, no. 7, pp. 318–328, Jun. 1988.

[8] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 63–74.

[9] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 211–218.

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

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

[12] P. Li and L.-N. Pouchet, "Throughput optimization for high-level synthesis using resource constraints," in *Int. Workshop on Polyhedral Compilation Techniques (IMPACT '14)*, 2014.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008.

[14] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 78–85.

[15] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 51:1–51:10.

[16] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 189–194.

[17] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[18] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "Graphite: Polyhedral analyses and optimizations for gcc," in *Proceedings of the 2006 GCC Developers Summit*, 2006, p. 2006.

[19] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic on-chip memory minimization for data reuse," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, April 2007, pp. 251–260.

[20] S. Bayliss and G. A. Constantinides, "Optimizing SDRAM bandwidth for custom FPGA loop accelerators," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 195–204.

[21] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 29–38.

[22] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 199–208.

[23] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, ser. CC'10/ETAPS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 283–303. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11970-5_16

[24] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. Martinez Caamaño, "Dynamic and speculative polyhedral parallelization using compiler-generated skeletons," *Int. J. Parallel Program.*, vol. 42, no. 4, pp. 529–545, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10766-013-0259-4

[25] A. Sukumaran-Rajam and P. Clauss, "The polyhedral model of nonlinear loops," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 48:1–48:27, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2838734

[26] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 185:185–185:194. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544141

[27] J. Doerfert, T. Grosser, and S. Hack, "Optimistic loop optimization," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 292–304.
[28] D. N. Sampaio, L.-N. Pouchet, and F. Rastello, "Simplification and runtime resolution of data dependence constraints for loop transformations," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017, pp. 10:1–10:11.
[29] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.
[30] P. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 19, no. 2, 1998.
[31] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Proc. Int. Conf. on Mathematical Software (ICMS '10)*, 2010.
[32] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Int. J. Parallel Programming*, vol. 34, no. 3, Jun. 2006.
[33] "Clang." [Online]. Available: http://clang.llvm.org
[34] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Int. Workshop on Polyhedral Compilation Techniques (IMPACT '12)*, 2012.
[35] "PoTHoLeS: Polyhedral Compilation Tool for High Level Synthesis." [Online]. Available: https://github.com/SamuelBayliss/Potholes
[36] "Polybench." [Online]. Available: http://web.cse.ohio-state.edu/~pouchet/software/polybench/
[37] "Gaussian elimination with pivoting." [Online]. Available: http://web.mit.edu/10.001/Web/Course_Notes/Gauss_Pivoting.c
[38] "Livermore loops coded in c." [Online]. Available: http://www.netlib.org/benchmark/livermorec

Junyi Liu Junyi Liu (S'14) received Bachelor's degree from Fudan University, China, in 2011, and Master's degree from cole Polytechnique Fdrale de Lausanne (EPFL), Switzerland, in 2013. He is finishing his PhD degree at Imperial College London. He has recently joined Microsoft Research Cambridge as a post-doc researcher working on FPGA acceleration for distributed systems.

John Wickerson John Wickerson (M'17) received a Ph.D. in Computer Science from the University of Cambridge in 2013. He currently holds an Imperial College Research Fellowship in the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include high-level synthesis, the semantics of programming languages, and software verification.

Samuel Bayliss Samuel Bayliss was awarded the Ph.D. degree in Electronic Engineering from Imperial College London in 2012. His principal research interest is the application of polyhedral analysis techniques to high level synthesis tools for FPGAs. Since 2015, he has worked in the Xilinx Research Labs in San Jose, California.

George Constantinides George A. Constantinides (S96-M01-SM08) received the Ph.D. degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently the Royal Academy of Engineering / Imagination Technologies Research Chair, Professor of Digital Computation, and Head of the Circuits and Systems research group. He has served as chair of the FPGA, FPL and FPT conferences. He currently serves on several program committees and has published over 150 research papers in peer refereed journals and international conferences. Prof Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.