Loop Splitting for Efficient Pipelining in High-Level Synthesis

Junyi Liu, John Wickerson, 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

***Abstract*—Loop pipelining is widely adopted as a key optimization method in high-level synthesis (HLS). However, when complex memory dependences appear in a loop, commercial HLS tools are still not able to maximize pipeline performance. In this paper, we leverage parametric polyhedral analysis to reason about memory dependence patterns that are *uncertain* (*i.e.*, parameterised by an undetermined variable) and/or *non-uniform* (*i.e.*, varying between loop iterations). We develop an automated source-to-source code transformation to split the loop into pieces, which are then synthesised by Vivado HLS as the hardware generation back-end. Our technique allows generated loops to run with a minimal interval, automatically inserting statically-determined parametric pipeline breaks at those iterations violating dependencies. Our experiments on seven representative benchmarks show that, compared to default loop pipelining, our parametric loop splitting improves pipeline performance by $4.3\times$ in terms of clock cycles per iteration. The optimized pipelines consume $2.0\times$ as many LUTs, $1.8\times$ as many registers, and $1.1\times$ as many DSP blocks. Hence the area-time product is improved by nearly a factor of 2.**

I. Introduction

With strong demand for higher computation efficiency, the adoption of specialized FPGA-based hardware accelerators is expanding in both systems-on-chip and data centers. High-level synthesis (HLS) has become a key FPGA design tool due to the promise of high development productivity. Software programs written in C/C++ or OpenCL can be compiled to register-transfer level (RTL) hardware designs by tools such as Xilinx Vivado HLS, Altera SDK for OpenCL, and academic tools such as LegUp [1]. Among various optimizations in HLS, loop pipelining is probably the first one considered when designers would like to improve performance of their critical loops. A high performance pipeline is able to start one new iteration at every clock cycle. However, this is often not possible for those loops with non-trivial loop-carried dependencies.

Although automatic loop pipelining is widely supported in many state-of-art HLS tools, they can only handle limited memory dependence patterns, which need to be fixed and known at compile-time. Often, in order to obtain efficient loop pipelining in practice, a designer has to analyse dependence patterns manually. It is also necessary to tune program directives and source code iteratively so that HLS tools can be guided to implement appropriate pipeline architectures. Therefore, with regard to loop pipelining, the performance gap between fully-automatic HLS and expert-tuned design is still considerable. We aim to realize formal support of *uncertain* (*i.e.*, parameterised by an undetermined variable) and *non-uniform* (*i.e.*, varying between different loop iterations) memory dependencies for loop pipelining in HLS.

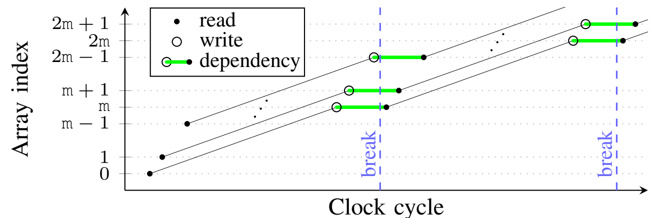


Fig. 1: Breaking pipeline execution at $(\mathtt{m}+1)^{\mathrm{th}}$ and $(2\mathtt{m}+1)^{\mathrm{th}}$ iterations for the loop shown in Listing 1.

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

Listing 1: Motivational loop with uncertain dependency.

Uncertain dependencies: The loop in Listing 1 provides a motivational example. It contains a parameterized affine recurrence equation [2] whose write access to the array A depends on a single undetermined variable m. HLS tools like Vivado HLS are not able to apply automatic loop pipelining to this loop, due to the uncertain memory dependency caused by variable m. When $\mathtt{m} \geq 1$, HLS tools cannot determine at compile time how many iterations can be overlapped in the pipeline without violating a read-after-write (RAW) memory dependency. However, if we force the pipeline to start one new iteration at each clock cycle, we can recognize that the first potential RAW conflict will happen at the $(\mathtt{m}+1)^{\mathrm{th}}$ iteration. In this iteration, the read access of `A[m]` may happen before the end of write access visiting the same array element in the first iteration. As shown in Fig. 1, we can break pipeline execution at the $(\mathtt{m}+1)^{\mathrm{th}}$ iteration ($\mathtt{i} = \mathtt{m}$) to resolve this RAW conflict; nevertheless, the next potential conflict will happen at the $(2\mathtt{m}+1)^{\mathrm{th}}$ iteration ($\mathtt{i} = 2\mathtt{m}$). In order to avoid memory conflicts when it is necessary, loop pipeline needs to halt the execution of the $(k\mathtt{m}+1)^{\mathrm{th}}$ iteration, where $k = 1, ..., \lfloor \mathtt{N}/\mathtt{m} \rfloor$. Consequently, in comparison to executing iterations sequentially – the behaviour of commercial tools with this code – loop parallelism can be improved significantly by keeping the pipeline as busy as possible without violating inter-iteration dependency.

Non-uniform dependencies: The pipeline strategy above can also optimize loops with non-uniform memory dependen-

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

Listing 2: Motivational loop with non-uniform dependency.

cies, which can appear in many applications such as matrix decomposition and triangular matrix computation. An example of such a loop is shown in Listing 2, where the dependency distance changes with the value of the induction variable. The optimization of this example is discussed in Section III as a detailed demonstration of our splitting methods.

In this paper, we propose a new pipeline optimization technique, which addresses the issue of avoiding both uncertain and non-uniform memory conflicts. The intuition of our technique is to burst as many iterations as possible down the loop pipeline at the target initialization interval, until sending another iteration would violate a dependency. The technique is implemented as a source-to-source code transformation, splitting the original loops, and inserting appropriate directives to steer downstream Vivado HLS optimization. This static, offline transformation can determine when to break pipeline execution at runtime, as a function of a set of run-time parameters; light-weight run-time checks of these parameters are automatically synthesised to determine when to insert pipeline bubbles. The rest of this paper explains how our problem can be formulated in terms of parametric polyhedral optimization and hence automated in a tool flow. In particular, we make the following contributions:

- We formulate a general parametric polyhedral analysis to capture inter-iteration memory dependences (Section IV-B).
- We develop a parametric polyhedral transformation to realize parametric loop splitting in order to ensure dependences are respected (Section IV-C).
- We prototype our new optimization technique as a source-to-source code transformation tool, which uses Xilinx Vivado HLS as the backend for RTL generation (Section IV-D).
- We evaluate loop splitting on seven benchmarks exhibiting complex loop-carried dependencies, and obtain $4.3\times$ average speed-up with moderate hardware overhead (Section V).

II. Related Work

There have been a number of approaches for improving the performance of loop pipelining in HLS. One approach is to develop efficient scheduling methods under the constraints of physical resources and memory dependencies. Zhang *et al.* [3] propose the 'System of Difference Constraints' to capture these constraints with timing requirements. By solving an optimization problem with these constraints, their solution can explore different schedules that trade-off resources and speed. Canis *et al.* [4] apply recurrence minimization to reduce initiation interval; their scheduling solution also enhances Zhang *et al.*'s method with a backtracking approach.

Among other recent efforts to optimize loop pipelining for HLS, polyhedral analysis has frequently been used. Morvan *et al.* [5] 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 is specially developed for loops with uncertain or non-uniform memory dependencies.

Alle *et al.* [6] propose a loop transformation technique inserting additional code that dynamically detects memory conflicts in the pipeline at runtime. In contrast to detecting and resolving memory conflicts at runtime, our loop splitting applies static analysis and transformation to customize conflict-free pipeline architectures at compile time.

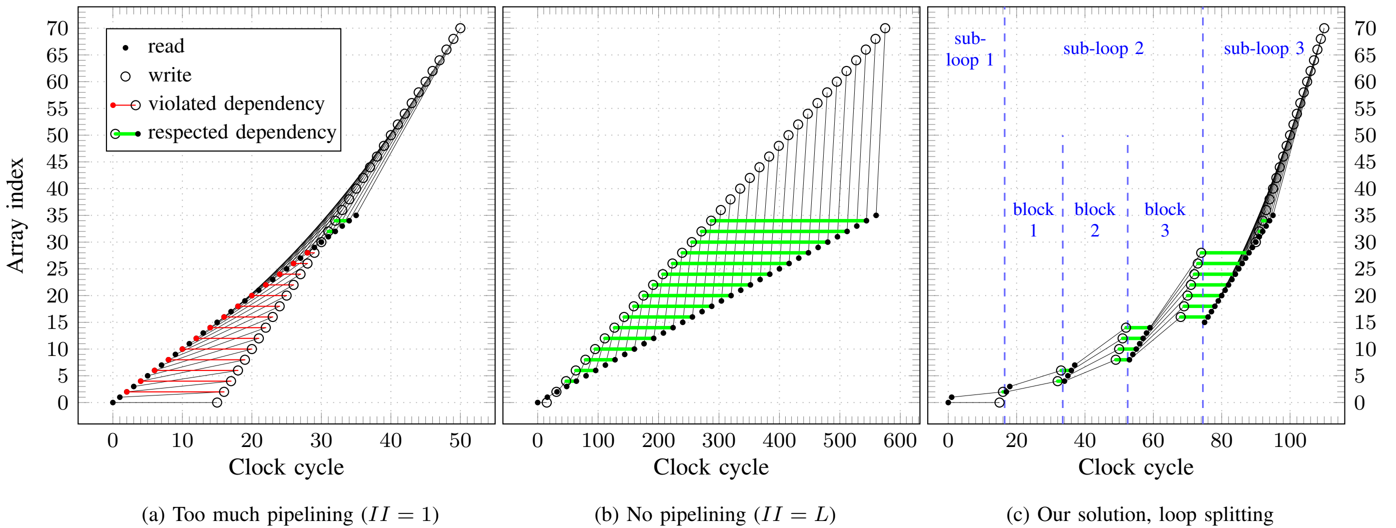
When there are uncertain or non-uniform memory dependence patterns in the loop, the existing static scheduling methods cannot resolve memory conflicts with a fixed and small initiation interval, which means that a conservative pipeline schedule with low parallelism has to be selected. Liu *et al.* [7] introduced a transformation for loops with uncertain memory dependence patterns extending previous approaches based on polyhedral analysis to a parametric polyhedral analysis – as we do here – to generate the condition in which the input loop can be pipelined without considering inter-iteration memory dependency. However, they only identify ranges of the parameters for which there are no significant dependencies, and so the loop can be pipelined fully – it is an 'all or nothing' approach. Our work generalises their technique by demonstrating that through a novel loop partitioning and transformation, even those loops that Liu *et al.* are unable to accelerate can have their performance significantly improved.

Li *et al.* [8] introduced an index-set splitting technique to improve inner loop parallelism. Their approach recognizes the loop iterations that are free of memory dependencies and memory port conflicts so that fast loop pipelining is applied on these iterations. Although their technique can also improve pipelining on some loops with non-uniform memory dependencies, our optimization is formalized to realize fine-grained splitting and support uncertain dependencies.

In addition to loop pipelining, there are various other polyhedral-based optimizations targeting data reuse and memory partitioning in HLS [9], [10], [11], [12]. These focus on optimizing custom memory sub-systems to reduce memory bottlenecks for better loop parallelism, and are complementary to our approach.

III. Motivation

In the context of HLS, loop pipelining is a technique to enable the execution of multiple successive iterations to be overlapped in a single synthesized pipeline hardware. The pipeline performance is reflected by the achieved *initiation interval* (II), which is the constant interval between the starts of successive loop iterations. If we assume the latency of one iteration is L clock cycles, the maximum number of parallel iterations in this pipeline is equal to $\lceil L/II \rceil$. Due to the physical limit of hardware resources and memory dependencies, the maximum parallelism of loop pipelining (*i.e.*, $II = 1$) is



(a) Too much pipelining ($II = 1$)

(b) No pipelining ($II = L$)

(c) Our solution, loop splitting

Fig. 2: Different pipelining strategies for the motivational loop shown in Listing 2

not always achievable, especially when there are non-trivial inter-iteration memory dependencies in the loop. If memory dependencies are constant for all iterations, HLS will increase II during the scheduling process to avoid memory conflicts safely. However, memory dependencies can also vary between iterations, or contain uncertainty. In these complex cases, a pipeline with fixed II is not able to resolve all potential memory conflicts. To realize loop pipelining, current HLS tools have to select II close to pipeline latency, which is often conservative in terms of parallelism.

In contrast to the conservative pipeline approach, we propose to synthesize a pipeline that can execute its iterations with highest parallelism when it is possible, as illustrated in Fig. 1. To realize this new optimization, we apply a loop transformation to split the loop, which guides the HLS tool to insert runtime break points into the generated pipeline.

A. *Dependence distance*

The analysis of memory dependencies is essential to determine how to split a loop. Given a pair of dependent write and read accesses in a loop, the data written in one iteration should be only read in a later iteration. The number of iterations between these two accesses is the *dependence distance* d, which will be further formalized in Section IV-B. In the motivational loop shown in Listing 2, the data written to `A[2*i]` at the $(\mathtt{i}+1)^{\mathrm{th}}$ iteration is read from `A[i+`d`]` at the later $(\mathtt{i}+d+1)^{\mathrm{th}}$ iteration. Therefore, $d = \mathtt{i}$, which indicates that the dependence distance of this loop varies between iterations.

For the sake of simplicity, assume for the example that the latency of the loop body is $L = 15$ clock cycles. Read and write accesses of the loop are assumed to be executed at the first and last pipeline stages respectively. In Fig. 2, thin black lines connect memory accesses in the same loop iteration. Horizontal lines connect reads and writes to the same array index; these lines are green when the write occurs after the read, and red otherwise. If we apply pipelining with $II = 1$ as shown in Fig. 2a, there will be L successive iterations executed in parallel in the pipeline. When

$$1 \leq d \leq L - 1, \qquad (1)$$

read accesses will be executed before the finish of their dependent write accesses, so that the memory dependency is violated as illustrated with red lines in Fig. 2a. Since d could be as low as 1, HLS tools like Vivado HLS have to select $II = L$ to satisfy the worst case, as shown in Fig. 2b (noting the greatly enlarged x-axis).

B. *Splitting the loop*

The conservatism of loop pipelining in HLS can be reduced by loop splitting. Firstly, we need to recognize which iterations cause memory conflicts when the loop is pipelined with $II = 1$. In Fig. 2a, the first and last iterations with d satisfying (1) are the 2nd and 15th iterations. The loop is then partitioned at these two iterations into three sub-loops as shown in Fig. 2c. The first (1st and 2nd iterations) and third (16th to N^{th} iterations) sub-loops are safely pipelined with $II = 1$.

After the fast parts of the motivational loop are separated, the bottleneck of pipeline parallelism is located in the second sub-loop. Since d of each iteration in this sub-loop satisfies (1), we further partition these iterations into loop blocks as shown in Fig. 2c. These blocks are pipelined respectively with $II = 1$, which is similar to breaking pipeline execution illustrated in Fig. 1. The size of each block is equal to the dependence distance of its first iteration. For instance, the first iteration of the first block is the 3rd iteration with $d = 2$. When the first block is executed in the pipeline with $II = 1$, the pipeline has to delay the start of the 5th iteration to avoid its RAW conflict with the 3rd iteration. Hence, there are two iterations (3rd and 4th) packed into the first block. It is notable that the pipeline break created for the first iteration in a block also resolves the RAW conflicts of the other iterations in this block. Similarly,

```
// Sub-loop 1: force pipelining with II=1
for (i=0; i<=1; i++)
  A[2*i] = A[i] + 0.5f;
// Sub-loop 2:
for (k=2; k<=14; k=k+k)
  // force pipelining inner loop with II=1
  for (i=k; i<=min(14,k+k-1); i++)
    A[2*i] = A[i] + 0.5f;
// Sub-loop 3: force pipelining with II=1
for (i=15; i<N; i++)
  A[2*i] = A[i] + 0.5f;
```

Listing 3: Transformed loop from Listing 2.

we partition the next four iterations (5th to 8th) into the second block and the rest (9th to 15th) into the third block.

C. Transformation template

The proposed splitting of the motivational loop can be represented concisely by the template shown in Listing 3. Three sub-loops can be pipelined entirely or block-wise with $II = 1$ by setting directives to force HLS tools to ignore any inter-iteration dependency. In the second sub-loop, an outer loop level is inserted to realize block-wise splitting. To create pipeline break points for runtime execution, we force the back-end HLS to apply pipelining with $II = 1$ on the inner loop by instructing it to ignore dependences. The bounds of the outer loop level determine the range of iterations in the original loop. The new induction variable `k` represents the start point of each block partition. The step size of `k` is equal to the dependence distance of the first iteration in the inner loop, and this $d = \mathtt{k}$ also defines the trip count of each block. The `min` function is used to ensure that the iterations of the last block do not exceed the upper bound of sub-loop 2. For example, the third block could contain eight iterations, but only the first seven ones (9th to 15th) are inside the second sub-loop as shown in Fig. 2c.

IV. Polyhedral-based Loop Splitting

In this section, we formulate a parametric polyhedral analysis to capture memory dependence patterns for loop pipelining. The parameters in the polyhedral models represent the undetermined variables in memory access patterns. Based on the information from the analysis, loop splitting are realized by transforming polyhedral models. Finally, we provide implementation details of our tool flow.

A. Preliminaries

The input of our analysis is a single nested loop with n levels. There are N_{pair} pairs of memory accesses visiting the same arrays in this loop, . In this paper, our analysis is described to capture RAW conflicts, but can also support other memory dependencies. In addition, m undetermined variables existing in memory accesses are represented as a parameter vector $p \in \mathbb{P}$, where $\mathbb{P} \subseteq \mathbb{Z}^m$ represents potentially known ranges of these variables.

Definition 1 (Iteration Domain): The iteration domain $\mathcal{D}^p$ is a parametric set of iteration vectors of the form:

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

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

For example, in Listing 1, $x = [\mathtt{i}, \mathtt{m}]^T$, and the inequality constraint of $\mathcal{D}^p$ is $\left[\begin{smallmatrix} -1 & 0 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} \mathtt{i} \\ \mathtt{m} \end{smallmatrix}\right] \leq \left[\begin{smallmatrix} 0 \\ \mathtt{N}-1 \end{smallmatrix}\right]$.

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

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

According to the sequential order of loop iterations, this lexicographic order represents that v' is executed after v in pipeline.

Definition 3 (Array Indexing Function): Given one memory access to a w-dimensional array, array indexing functions are assumed in this paper to take the general affine form:

$$y = f(v, p) = Av + Bp + c,$$

where $v \in \mathcal{D}^p$, and $y \in \mathbb{Z}^w$ is a vector of array indices. $A \in \mathbb{Z}^{w \times n}$ and $B \in \mathbb{Z}^{w \times m}$ are given as two coefficient matrices, and $c \in \mathbb{Z}^w$ is given as one constant vector. This parametric affine expression transforms one iteration vector into one multi-dimensional array index.

The dependency relations of loop iterations can be assessed by array indexing functions of dependent memory accesses.

Definition 4 (Iteration Dependency Map): In a given loop nest, the kth pair of write and read accesses visiting the same array can form one iteration dependency map $\mathcal{Q}_k^p$. This map links two dependent iteration vectors 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 $v' \succ v$ ensures that the sink iteration v' is executed after the source iteration v. $w_k(v, p)$ and $r_k(v', p)$ are array indexing functions of the kth pair of write and read accesses, whose equality constraint implies a read-after-write memory dependency between these accesses.

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 dependency in loop pipelining: As mentioned in Section III, pipeline architecture is affected by both inter-iteration memory dependency and loop scheduling. The information about pipeline scheduling is assumed to be available for our analysis. To formally evaluate memory conflicts in loop pipelining, we firstly 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 kth 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

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

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

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

which counts the number of iterations that are executed before the iteration corresponding to v. Generally, $I^p(v)$ can be expressed as a parametric pseudo-polynomial, as known as an Ehrhart polynomial, that counts the integer points of a parametric polytope [13]. Similar to (1), the inequality conditions check the existence of memory conflicts based on given pipeline scheduling.

As shown in Def. 5, the conflict domain $\mathcal{S}_k^p$ includes all iterations that will violate memory dependencies when the entire loop is pipelined with the target II. To consider 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 $\mathcal{S}_k^p$.

The set of parameter vectors p for which $\mathcal{S}_k^p = \emptyset$ is the *safe region*, denoted as $\mathcal{P}_k \subseteq \mathbb{P}$. 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 on the entire loop nest.

In this work, the Integer Set Library (ISL) [14] is used to implement our parametric polyhedral analysis. As mentioned before, the general form of $I^p(v)$ is a parametric pseudo-polynomial, which are representable with ISL. Morvan et al. [5] estimated the lower bound of $I^p\left(Z_k^p(v)\right) - I^p(v)$ to check pipeline legality for nested loops, but this bound was mentioned to be not always tight. Therefore, sophisticated analysis of parametric pseudo-polynomial is limited in ISL.

2) Constructing the conflict domain: In our analysis, we limit the form of $I^p\left(Z_k^p(v)\right) - I^p(v)$ so that we can use lexicographic optimization functions from ISL to generate $\mathcal{P}_k$ from $\mathcal{S}_k^p$. Currently, we restrict this expression with counting integer points to be affine such that

$$d_k(v) = I^p\left(Z_k^p(v)\right) - I^p(v) = c_0^T v + c_1^T p + c_2^T,$$

where c_0, c_1, c_2 are constant vectors of coefficients. To create this affine expression, $\mathcal{Q}_k^p$ is firstly used to generate the difference of iteration vectors, which is $Z_k^p(v) - v$.

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

$$\delta_k^p(v) = Z_k^p(v) - v = Av + Bp + c,$$

where $A \in \mathbb{Z}^{n\times n}$ and $B \in \mathbb{Z}^{n\times m}$ are coefficient matrices, and $c \in \mathbb{Z}^n$ is a constant vector. Since lexicographic optimization is applied to obtain $Z_k^p(v)$, δ_k^p is still single-valued when the coefficient matrix of v' in the read access function $r(v', p)$ is not invertible.

Then, we use $\delta_k^p(v)$ and trip counts of loop levels to count integer points between v and $Z_k^p(v)$. We note that $\delta_k^p(v) = \left[\delta_{k,0}^p, \delta_{k,1}^p, \dots, \delta_{k,n-1}^p\right]^T$, and for $0 \le i < n$, t_i represents the trip count of the ith loop level. Due to the affine restriction, the supported cases of calculating $d_k(v)$ are shown as follows.

- **Rectangular case.** If every loop level has a uniform trip count (which means that
  $$d_k(v) = \left[\Pi_{i=1}^{n-1} t_i,\ \Pi_{i=2}^{n-1} t_i,\ \dots,\ t_{n-1},\ 1\right]\delta_k^p(v)$$
  holds), then there must be at most one level (say, j) with a parametric trip count, and the dependence difference at every level outside j must be constant (*i.e.*, $\forall 0 \le i < j$, $\delta_{k,i}^p$ is constant).
- **Non-rectangular case.** Otherwise, let j be the innermost level with a non-uniform trip count. Then every level inside j must have a constant trip count (which means
  $$d_k(v) = \left[\dots, \Pi_{i=j}^{n-1} t_i,\ \Pi_{i=j+1}^{n-1} t_i,\ \dots, t_{n-1}, 1\right]\delta_k^p(v)$$
  holds), the dependence difference at level $j-1$ must be 0 or 1 (*i.e.*, $\delta_{k,j-1}^p \le 1$), and the dependence difference at every level outside $j-1$ must be 0 (*i.e.*, $\forall 0 \le i < j-1, \delta_{k,i}^p = 0$).

C. Polyhedral Transformation

According to the memory dependency analysis, the iteration domain $\mathcal{D}^p$ is partitioned based on the conflict domain and dependence difference, when parameters are in the *conflict region*, $\mathcal{P}_{\text{conf}} = \mathbb{P} \setminus \mathcal{P}_{\text{safe}}$. In general, our transformation modifies the model of static control parts (SCoP) [15] for representing loop programs.

1) Determining the conflict dimension: In order to determine which level of a given loop nest should be split, each non-zero dependence difference δ_k^p is assessed to locate the conflict dimension named as q. 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: $\delta_{k,i}^p \ne 0$ and $\forall 0 \le j < i, \delta_{k,j}^p = 0$. Then, the conflict dimension q is generated as the largest local conflict dimension for all $1 \le k \le N_{\text{pair}}$. Therefore, when the loop nest is split at the qth 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 realized by partitioning the iteration domain into three sub-domains at the conflict dimension. We use lexicographic optimization to generate the first and last iterations causing memory conflict:

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

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

$$\mathcal{D}^p = \bigcup \begin{cases} \mathcal{D}_1^p = \left\{v \;\middle|\; v \in \mathcal{D}^p \wedge v_q \le l_q^p\right\} \\ \mathcal{D}_2^p = \left\{v \;\middle|\; v \in \mathcal{D}^p \wedge l_q^p < v_q \le u_q^p\right\} \\ \mathcal{D}_3^p = \left\{v \;\middle|\; v \in \mathcal{D}^p \wedge u_q^p < v_q\right\} \end{cases}.$$

```
for (k=2; k<=14; k++){
  for (i=k; i<=min(14,k+k-1); i++)
    A[2*i] = A[i] + 0.5f;
  k=k+k-1;   }
```

Listing 4: Alternative form of sub-loop 2 shown in Listing 3.

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 qth loop level block-wise, and fast pipelining is applied on split 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

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

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. It is generated by taking the union of minimum positive $\delta_{k,q}^p$ such as

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

where $\mathcal{P}_{\text{conf}} = \bigcup_{i=1}^{N_{\text{piece}}} \mathcal{P}_{\text{conf}}^i$, and N_{piece} represents the number of disjoint parameter sets in the conflict region.

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}. This alternative transformation is illustrated in Listing 4, where induction variable `k` is incremented by $t_{blk} = $ `k` after the execution of the inner loop level.

D. Implementation

We prototype our optimization technique in a source-to-source code transformation framework. In our tool flow, PoTHoLeS [16] is used to implement our custom polyhedral analysis and transformation with the support of ISL. In PoTHoLeS, the SCoP model of the input loop is generated by the Polyhedral Extraction Tool (PET) [17]. Xilinx Vivado HLS is used as the back-end tool for generating RTL code. Due to the limitation of collecting detailed scheduling information in a commercial HLS tool, we use Vivado HLS to synthesize the input loop once without considering inter-iteration dependencies and take the implemented iteration latency as L_k for the

```
// 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 <= 9){
    for (i = 0; i < -m + 1; i++)                                         sub-loop 1
        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;
    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. 3: Demonstration of code transformation.

polyhedral analysis. To configure loop pipelining in Vivado HLS, we also customize the insertion of program directives (`#pragma`) in the generation of transformed code.

One example of our implemented code transformation is shown in Fig. 3. The original loop is a two-dimensional loop with one undetermined variable `m` in the write to `A[2*i+m][j]`, which implies uncertain and non-uniform memory dependency. Since we also implement parametric loop pipelining within our flow, the transformed code contains the detection of conflict region of parameters, which is implemented by the condition of the outermost if statement. The fast loop in the else-case is aggressively pipelined by ignoring inter-iteration dependency. The then-case includes three sub-loops split from the original loop. Because the conflict domain of the original loop is parametrized, bounds of sub-loops contain `m` and code macros like `min()` and `max()`. For the original loop, the dependence difference recognized by our tool is `m+i`. Memory conflicts are only related to `i`, and thus our tool splits the original loop at the outer level. In sub-loop 2, a new loop level is inserted with induction variable `k`, which realizes block-wise loop splitting. Since Vivado HLS cannot apply entire pipelining on a nested loop with variable bounds, only the loop levels inside the inserted one in sub-loop 2 can be pipelined. Therefore, we leverage this feature to implement block-wise loop pipelining.

V. Experiments

A. Benchmarks

As discovered in [8], [7], most loops in Polybench [18] and some other related benchmarks contain only fixed and uniform memory access patterns. Loop pipelining in HLS is also preferred to implement these relatively simple loops due to its lack of supporting uncertain and non-uniform memory dependencies. Therefore, to evaluate the effectiveness of our

TABLE I. Performance and resource results of proposed loop transformation.

Benchmark	Initiation Interval			Avg. Cycles/Iter			Clock (ns)			Avg. Time/Iter (ns)			LUT			FF			DSP48E1			Area-Time Product*		
	Orig	Fast	Split	Orig	Tran	ratio	Orig	Tran	ratio	Orig	Tran	ratio	Orig	Tran	ratio	Orig	Tran	ratio	Orig	Tran	ratio	Orig	Tran	ratio
dist_param	12	1	1	12.0	5.0	0.41	2.068	2.428	1.17	24.9	12.0	0.48	233	467	2.00	332	584	1.76	2	2	1.00	5.79	5.63	0.97
dist_itr	14	-	1	14.0	1.7	0.12	2.303	2.816	1.22	32.3	4.9	0.15	225	422	1.88	397	641	1.61	2	2	1.00	7.26	2.06	0.28
dist_itr_param	6	1	1	6.1	1.7	0.28	2.052	2.843	1.39	12.4	4.8	0.39	283	827	2.92	397	1037	2.61	4	6	1.50	3.52	3.98	1.13
typ_loop	12	1	1	12.0	1.7	0.14	3.174	2.751	0.87	38.1	4.6	0.12	636	742	1.17	908	1246	1.37	5	5	1.00	24.23	3.44	0.14
jacobi_2d	48	3	3	48.0	14.2	0.29	2.354	2.939	1.25	113.0	41.6	0.37	614	1875	3.05	879	2506	2.85	7	10	1.43	69.38	78.02	1.12
tri_sp_slv	18	2	2	18.8	7.5	0.40	2.976	3.010	1.01	55.9	22.5	0.40	447	783	1.75	704	1054	1.50	6	6	1.00	24.97	17.61	0.71
floyd_warshall	14	-	2	14.0	2.3	0.16	2.511	2.758	1.10	35.2	6.3	0.18	464	727	1.57	692	1078	1.56	2	2	1.00	16.31	4.60	0.28
Geomean						0.23			1.13			0.27			1.95			1.83			1.12			0.52

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

loop splitting, we select seven representative benchmarks whose dependence distances can be larger than one iteration in their conflict regions. The loop bodies of these benchmarks contain data paths of single precision floating point numbers. The undetermined variables may appear in memory indices and loop bounds, which are integers between `INT_MIN` and `INT_MAX` as defined in `<limits.h>`.

dist_param and **dist_itr** are 1D loops shown in Listing 1 and Listing 2 respectively. **dist_itr_param** is a 2D loop shown in Fig. 3. **typ_loop**, **jacobi_2d** and **tri_sp_slv** are used as benchmarks by Liu et al. [7]. Specifically, **jacobi_2d** is a 2D Jacobi stencil loop from Polybench modified to include uncertain dependencies. **tri_sp_slv** is a 1D loop with non-uniform dependencies, which has one undetermined iteration causing a memory conflict. **floyd_warshall** is a 3D loop for finding shortest paths from Polybench, which has one fixed iteration causing a memory conflict in the innermost loop. Source codes, testbenches, and transformed codes of all benchmarks are available in a public GitHub repository [1].

B. Experimental setup

In our experiments, we use Xilinx Vivado HLS 2015.4 as the RTL generation backend. The target FPGA platform is Xilinx Virtex 7 XC7VX485T. On-chip memory in Virtex 7 is dual-port block RAM, which is considered when calculating the target initiation interval. To achieve balanced results of clock frequency and resource usage, we set the target clock period as 3ns for all experiments. The functional correctness of our loop transformation is verified with dedicated testbenches through C/RTL co-simulation provided by Vivado HLS. Furthermore, Xilinx Vivado Design Suite 2015.4 is used to synthesize, place and route the generated RTL codes for precise timing and implementation results.

C. Results

1) Overall performance: Table I presents the overall performance results, which compares our transformation to the default loop pipelining of the HLS tool. Columns "Orig" and "Tran" contain the results of the original and transformed pipelines respectively. Column "Fast" refers to the duplicated loops whose parameters are in the safe region, and column "Split" refers to the split sub-loops. There are no undetermined variables in **dist_itr** and **floyd_warshall**, so that our transformation only splits these loops.

Since we safely guide the HLS tool to ignore inter-iteration dependency when it is necessary, the transformed pipelines have an II ranging from just 1 to 3 cycles. This achieved II only reflects the peak performance of the transformed pipelines. To evaluate runtime performance, we measured loop execution time with further experiments. For each loop with uncertain memory accesses, we generated 1000 test cases with random values of uncertain variables. These random values were ensured to be within the conflict region of parameters, so that we can assess the runtime performance of loop splitting. For each benchmark, we collected 1000 pairs of iteration numbers and execution time in clock cycles. The average cycles per iteration in Table I shows a 4.3× speed-up in clock cycles after loop splitting.

In return for higher runtime parallelism, the transformed pipelines have a slower clock speed than the original pipelines. The clock period is increased by 13% on average, which indicates that our transformation does not significantly increase the critical paths of generated pipelines. To consider the impact of an increased clock period, average latency (in nanoseconds) per iteration has also been calculated in Table I. Our optimization still provides a 3.8× speed-up in real execution time.

2) Resource usage: Despite the improvement of loop parallelism, our optimization also leads to the growth of resource usage. As shown in Table I, after transformation, the average utilization of Look-up Tables (LUTs) and Flip-Flop registers (FFs) increases by 95% and 83% respectively. In addition, our HLS backend tends to share DSP blocks used for floating point computation across different loop bodies. As shown in Table I, the average DSP block usage is increased by only 12%.

Although the loop bodies of transformed loops are all duplicated from the originals as shown in Fig. 3, it is still difficult for our HLS backend to explore resource sharing, especially for the calculation of memory addresses. However, resource overhead is still less significant than performance improvement, as witnessed by a 48% average reduction of the area-time product. In Vivado HLS, forcing resource sharing can be realized by replacing duplicated loop bodies with same function call and disabling function inlining. Such complementary transformation could be effective to reduce resource

[1] https://github.com/Junyi-Liu/benchmarks-HLS/tree/master/LSP

TABLE II. Splitting stages applied to benchmarks.

Benchmark	Splitting Stage	
	conflict domain	dependence difference
dist_param	-	✓
dist_itr	✓	✓
dist_itr_param	✓	✓
typ_loop	-	✓
jacobi_2d	-	✓
tri_sp_slv	✓	-
floyd_warshall	✓	-

overhead when array index functions are complex.

3) Analysis of runtime performance: According to the different memory dependence patterns detected in the benchmarks, our transformation applied different combinations of splitting stages, which are summarized in Table II. When the loops are split by conflict domain, most transformed loops have an average cycles/iteration close to II, as shown in Table I. In particular, the second sub-loops of **tri_sp_slv** and **floyd_warshall** are empty, so that there is no further splitting by dependence difference in these loops. The transformed **tri_sp_slv** has a relatively larger cycles/iteration due to its undetermined loop bounds.

For **dist_param**, **typ_loop** and **jacobi_2d**, 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. 4 shows the performance of four types of pipeline architectures, where we compare our loop splitting (LSP) with parametric loop pipelining (PLP) [7]. We also collected performance upper bounds of **dist_param**. They are created by synthesising the loop with $\mathtt{m}$ replaced by constants inside the conflict region. Although the behaviour of the synthesised pipelines is only correct for their fixed dependence difference, these pipelines still represent the ideal loop performance. When $\mathtt{m}$ is 1, all iterations have to be executed sequentially. Due to extra operations added to support LSP, the pipeline optimized by LSP is slower than the original one only in this case. The conflict region of **dist_param** is $1 \leq \mathtt{m} \leq 13$, where the pipeline optimized by PLP acts as same as the original one. In contrast, when $\mathtt{m}$ becomes larger, there are fewer break points inserted in the execution of the loop transformed by LSP, so its runtime performance becomes closer to the ideal one. For both LSP and PLP, this ideal performance can be achieved when the loop is executed in fast mode (where $\mathtt{m} \geq 14$).

VI. Conclusion

In this paper, we introduce loop splitting to improve the performance of loop pipelining. It is realized based on parametric polyhedral analysis and transformation. We prototyped our proposed optimization in a source-to-source code transformation framework which is compatible with commercial HLS tools such as Vivado HLS. For a given loop, our splitting is able to resolve all potential conflicts of uncertain or non-uniform memory dependency at compile time. According to experiments on seven benchmark loops, runtime parallelism is

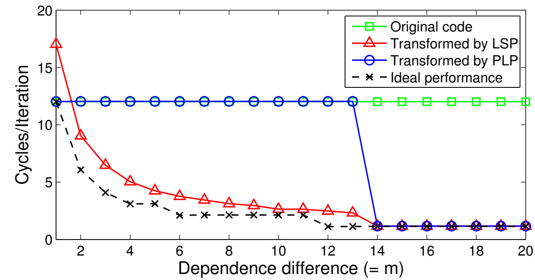


Fig. 4: Evaluating performance of transformed **dist_param**.

increased significantly by our optimization technique, specifically $4.3\times$ speed-up in clock cycles per iteration. At the same time, resource usage overhead is shown to be moderate.

Acknowledgments

The support of the EPSRC grants EP/I020357/1 and EP/K034448/1, the Royal Academy of Engineering, and Imagination Technologies is gratefully acknowledged. We thank the anonymous reviewers for their helpful suggestions.

References

[1] 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 *FPGA*, 2011.

[2] 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.

[3] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *ICCAD*, 2013.

[4] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," in *FPL*, 2014.

[5] 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.

[6] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *DAC*, 2013.

[7] J. Liu, S. Bayliss, and G. A. Constantinides, "Offline synthesis of online dependence testing: Parametric loop pipelining for HLS," in *FCCM*, 2015.

[8] 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.

[9] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic on-chip memory minimization for data reuse," in *FCCM*, 2007.

[10] S. Bayliss and G. A. Constantinides, "Optimizing SDRAM bandwidth for custom FPGA loop accelerators," in *FPGA*, 2012.

[11] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *FPGA*, 2013.

[12] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *FPGA*, 2014.

[13] 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.

[14] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Proc. Int. Conf. on Mathematical Software (ICMS '10)*, 2010.

[15] 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.

[16] "PoTHoLeS: Polyhedral Compilation Tool for High Level Synthesis." [Online]. Available: https://github.com/SamuelBayliss/Potholes

[17] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Int. Workshop on Polyhedral Compilation Techniques (IMPACT '12)*, 2012.

[18] "Polybench." [Online]. Available: http://web.cse.ohio-state.edu/~pouchet/software/polybench/