# Run Fast When You Can: Loop Pipelining with Uncertain and Non-uniform Memory Dependencies

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*—As a key optimisation method in high-level synthesis (HLS), high-performance loop pipelining is enabled by the static scheduling algorithm. When there are non-trivial memory dependencies in the loop, current HLS tools have to apply conservative pipeline schedule that also leads to nearly sequential execution. In this paper, we demonstrate using parametric polyhedral model to mathematically capture uncertain (i.e., parameterised by an undetermined variable) and/or non-uniform (i.e., varying between loop iterations) memory dependence patterns. According to this static analysis, if we always execute the loop with an aggressive (fast) pipeline schedule, we can generate the parameter conditions in which this execution is safe and the parametric break points when the execution encounters memory conflicts. Then, we apply these information into an automated source-to-source code transformation, which implements parametric loop pipelining and loop splitting. The transformed loop is synthesised by Vivado HLS and its execution speed can be adjusted at runtime to avoid memory conflicts. The experiments over a set of benchmark loops show that our optimisation can improve the runtime pipeline performance significantly with a reasonable overhead of hardware resources.**

## I. Introduction

High-level synthesis (HLS) enables high hardware design productivity especially for specialised computing on on field-programmable gate arrays (FPGAs). State-of-the-art HLS tools like Xilinx Vivado HLS [1], Intel FPGA SDK for OpenCL [2] and LegUp [3] are able to synthesise programs written in high-level languages like C/C++/OpenCL into hardware designs described in VHDL/Verilog. Unfortunately, high-performance hardware design still needs sufficient manual effort on source code refactoring and optimisation tuning. Hardware design knowledge is essential to realising high-quality HLS designs.

Computational bottlenecks are typically located in some critical loops of high-level programs, and hence loop pipelining has emerged as one of the preeminent optimisation techniques in HLS. Loop-pipelining techniques work by automatically overlapping the execution of loop iterations without violating any memory dependency. Nevertheless, it relies on comprehensive static analysis and can only perform well when loop bounds and memory accesses are all determined. Optimising loops with uncertainty is still poorly supported in HLS tools. To preserve correct loop behaviours, the generated hardware architecture can be too conservative, so that loop iterations have to be executed sequentially.

The motivational loop shown in Listing 1 contains a parameterised affine recurrence equation [4]. In this loop, there

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

Listing 1: Motivational loop with uncertain dependency.

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

Listing 2: Motivational loop with non-uniform dependency.

is an undetermined variable `m` in the write access pattern of array `A`. The loop iterator `i` ranges from zero to $\mathtt{N}-1$, 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`. 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.

In this paper, it is demonstrated that efficient dynamic pipeline execution can be realised with static optimisations. We 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. The demonstration of this analysis is 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. To keep the pipeline of Listing 1 as busy as possible, we need to halt pipeline execution at appropriate iterations to avoid the memory conflicts. The strategy of breaking the pipeline execution can also optimise 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 optimised by loop splitting, first proposed in [6]. The proposed optimisations have been implemented into 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 high-performance pipeline architecture. Therefore, our transformation is also flexible enough to be applied to different HLS tools.

## II. Related Work

In the recent work of loop pipelining in [7] and [8], the authors rely on knowing, at compile time, all the dependencies that exist between operations to exploit pipeline schedule. 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. Among other recent efforts to optimise loop pipelining for HLS, polyhedral analysis has frequently been used. Morvan *et al.* [9] propose a method using polyhedral analysis to improve nested loop pipelining. Li *et al.* [10] introduced an index-set splitting technique with classical affine loop transformations [11] to improve inner loop parallelism.

Besides regular loop structures, there are active HLS research efforts as in investigating pipelining for loops with irregular structures. Tan *et al.* [12] describe an approach called ElasticFlow synthesise parallel pipeline instances of dynamic-bound inner loop. Alle *et al.* [13] implement a compilation method that synthesize disambiguation logic in the hardware pipelines that can fully analyse the inter-iteration dependency at runtime. Dai *et al.* [14] 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. Although the techniques in both [13], [14] are 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.

## III. Optimization Method

### 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 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

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

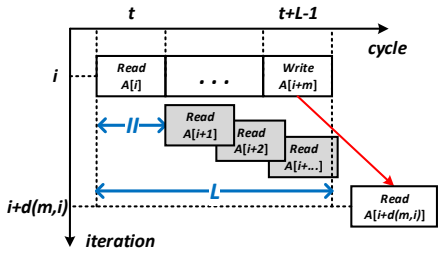


Fig. 1: The conflict region of $d(\mathtt{m}, \mathtt{i})$ in Listing 1.

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 [7], [8]. 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.

### B. Memory Dependence Analysis

Here, we present an intuitive illustration of our parametric polyhedral analysis with the one-dimensional loop shown in Listing 1. 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. 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. 1, we have $d(\mathtt{m}, \mathtt{i})$ as the dependence iteration distance for the loop shown in Listing 1. The red 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

```
// 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.

memory dependency. In other words, `A[i+`$d(\mathtt{m},\mathtt{i})$`]` cannot be any grey read access shown in Fig. 1(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*.

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

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 or enough iterations between them. According to Fig. 1(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. 2. 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.

Before the loop starts, the conflict region is firstly evaluated by the if-condition derived from (2). These conditions will 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 $II = 1$. The split controller will still run the loop in a fast speed but pause the pipeline input after every `m` iterations are issued. Our 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.

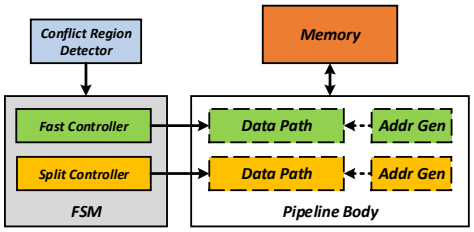


Fig. 2: Conceptual pipeline architecture.

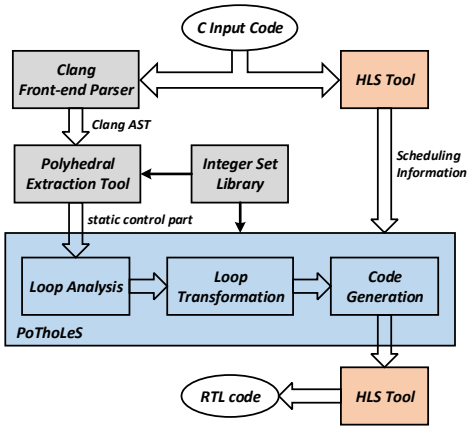


Fig. 3: Tool flow for code transformation framework.

### D. Source-to-source Code Transformation Framework

To prototype our new loop optimization and make it compatible with an HLS tool, we integrated our analysis algorithm into a source-to-source code transformation framework shown in Fig. 3. In this work, 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 synthesise 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 $II$. We also extract the pipeline latency achieved from the first synthesis as the latency between all dependent memory accesses, which leads it to be an upper bound value. If our transformation is applied inside an HLS tool, we can derive more accurate scheduling information, so that the conflict region has the potential to be further tighten.

As shown in Fig. 3, the loop information is captured by two open-source tools. The Clang front-end parser [15] generates an abstract syntax tree (AST) from the input C code. The Polyhedral Extraction Tool (`pet`) [16] extracts the loops as the static control parts (SCoPs) from the Clang AST. Finally, the transformed C code is generated by PoTHoLeS. PoTHoLeS

is a polyhedral compilation tool developed by us, which conducts user-specified loop analysis and transformation based on `isl` [17]. The tool implementing our optimisation is available in a public Github repository. [1]

## IV. Experimental Results

In this paper, our code transformation framework is named as polyhedral-based dynamic loop pipelining, denoted PolyDLP. It 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.

### A. Benchmarks

We choose seven benchmark loops used in our previous works [5], [6] for our experimental study. 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]

Our target loops can alternatively be optimised by runtime dependence analysis (RT-Dep) proposed by *et al.*Alle [13] for loop pipelining. RT-Dep is available online as a plug-in of a source-to-source compiler framework called Gecos [18]. In this experimental section, we also evaluate RT-Dep with our benchmarks for a comparative study. To make RT-Dep compatible with Vivado HLS, we have to manually insert loop pipelining and unroll pragmas in the transformed codes, which can be found in another public repository.[3] It is noteworthy that RT-Dep is developed for loops with more general inter-iteration dependencies, such as those having iteration-dependent indirect array accesses (*e.g.* `A[B[i]]` where `i` is a loop iterator). This results in RT-Dep implementing logic for detecting and resolving runtime memory conflicts.

### B. Impact on pipeline scheduling

Table I indicates the change of pipeline scheduling caused by PolyDLP. In this table, columns with the title "Orig" indicate characteristics of the original pipeline and columns with the title "Tran" indicate characteristics of the pipeline transformed by PolyDLP. The numbers under the title "ratio" are calculated against the metrics of the original pipeline implementation. Columns with the title "Fast" indicate the pipeline performance achieved when the lightweight checks of the conflict region 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 and "Iteration Cycles" represents the number of cycles for one loop iteration.

TABLE I. The impact of loop splitting on pipeline scheduling.

| Benchmark | Pre-Loop Cycles | | | Iteration Cycles | | | Initiation Interval | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Orig | Tran | ratio | Orig | Split | Fast | Orig | Split | Fast | ratio |
| dist_param | 1 | 2 | 2.00 | 12 | 14 | 14 | 12 | 1 | 1 | 0.08 |
| dist_itr | 1 | 1 | 1.00 | 14 | 14 | - | 14 | 1 | - | - |
| dist_itr_param | 1 | 8 | 8.00 | 15 | 17 | 17 | 6 | 1 | 1 | 0.17 |
| typ_loop | 8 | 10 | 1.25 | 17 | 15 | 19 | 12 | 1 | 1 | 0.08 |
| row_col | 8 | 9 | 1.13 | 15 | 15 | 17 | 12 | 2 | 2 | 0.17 |
| tri_sp_slv | 1 | 3 | 3.00 | 22 | 31 | 30 | 18 | 2 | 2 | 0.11 |
| floyd_warshall | 1 | 1 | 1.00 | 18 | 20 | - | 14 | 2 | - | - |
| **Geomean** | | | 1.83 | | | | | | | 0.12 |

As shown in Fig. 2, the conflict region detector is synthesised to execute before the start of the loop body. These additional operations are observed to cost a few cycles, which indicates that the complexity of the detector logic is lightweight. $L_{iter}$ in the both splitting and fast modes is slightly increased because the loops are all aggressive pipelined. However, according to (1), these increases cannot signify the impact of $L_{pre}$ and $L_{iter}$, especially when there is a large number of iterations to be executed. 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, $II$ ranging from just 1 to 2 cycles is achieved, which leads to $8.3\times$ higher peak performance in the *fast* mode.

### C. Timing Overhead and Performance Improvement

In Table II, columns with the title "RT" indicate characteristics of the pipeline transformed by RT-Dep. For all benchmarks, our transformation has minimal impact on the achievable clock period. However, RT-Dep makes the timing much worse by $33\%$ on average. This difference is mainly caused by more complex runtime detection of memory conflicts in the pipeline controller generated by RT-Dep. Although there is also increased control logic in the FSM generated by PolyDLP, its timing overhead can be amortised by the HLS scheduling effort.

In this paper, the runtime performance evaluation is focused on the benchmark loops running in their conflict regions. We measured the latency of loop execution with additional 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. These 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. With PolyDLP, the average cycles per iteration in Table II shows a $4.2\times$ speed-up of the pipeline throughput in the conflict region. After including the timing effect, PolyDLP can sustain $3.7\times$ average speed-up, while RT-Dep can only achieve $2.17\times$ average speed-up.

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

[2] https://github.com/Junyi-Liu/benchmarks-HLS/tree/master/PolyDLP

[3] https://github.com/Junyi-Liu/benchmarks-HLS/tree/master/Thesis/RT-Dep

TABLE II. The experimental results of timing, pipeline performance and resource usages.

| Benchmark | Clock (ns) | | | | | Avg. Cycles/Iter | | | | | Avg. Time/Iter (ns) | | | | | LUT | | | | | | FF | | | | | | DSP48E1 | | | | | | Area-Time Product* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Orig | RT | ratio | Tran | ratio | Orig | RT | ratio | Tran | ratio | Orig | RT | ratio | Tran | ratio | Orig | HP | RT | ratio | Tran | ratio | Orig | HP | RT | ratio | Tran | ratio | Orig | HP | RT | ratio | Tran | ratio | Orig | RT | ratio | Tran | ratio |
| dist_param | 2.02 | 3.66 | 1.81 | 2.32 | 1.15 | 12.0 | 3.9 | 0.32 | 5.7 | 0.48 | 24.3 | 14.2 | 0.59 | 13.3 | 0.55 | 239 | 268 | 400 | 1.67 | 487 | 2.04 | 340 | 425 | 575 | 1.69 | 595 | 1.75 | 2 | 2 | 2 | 1.00 | 2 | 1.00 | 5.8 | 5.7 | 0.98 | 6.5 | 1.11 |
| dist_itr | 2.02 | 3.66 | 1.81 | 2.33 | 1.15 | 14.0 | 1.6 | 0.11 | 1.8 | 0.13 | 28.3 | 5.7 | 0.20 | 4.2 | 0.15 | 230 | 242 | 391 | 1.70 | 400 | 1.74 | 405 | 417 | 572 | 1.41 | 623 | 1.54 | 2 | 2 | 2 | 1.00 | 2 | 1.00 | 6.5 | 2.2 | 0.34 | 1.7 | 0.26 |
| dist_itr_param | 2.72 | 3.63 | 1.34 | 2.72 | 1.00 | 6.1 | 10.0 | 1.65 | 1.7 | 0.29 | 16.4 | 36.3 | 2.21 | 4.7 | 0.29 | 401 | 382 | 462 | 1.15 | 1214 | 3.03 | 454 | 538 | 650 | 1.43 | 1209 | 2.66 | 3 | 3 | 3 | 1.00 | 4 | 1.33 | 6.6 | 16.8 | 2.55 | 5.8 | 0.87 |
| typ_loop | 3.16 | 4.08 | 1.29 | 3.66 | 1.16 | 12.0 | 3.1 | 0.26 | 1.7 | 0.14 | 37.9 | 12.7 | 0.33 | 6.1 | 0.16 | 784 | 712 | 838 | 1.07 | 957 | 1.22 | 756 | 819 | 752 | 0.99 | 1045 | 1.38 | 4 | 4 | 2 | 0.50 | 4 | 1.00 | 29.7 | 10.6 | 0.36 | 5.9 | 0.20 |
| row_col | 2.39 | 3.03 | 1.27 | 2.59 | 1.09 | 12.2 | 5.7 | 0.47 | 5.5 | 0.45 | 29.2 | 17.4 | 0.60 | 14.4 | 0.49 | 809 | 827 | 739 | 0.91 | 988 | 1.22 | 1108 | 1255 | 1016 | 0.92 | 1392 | 1.26 | 8 | 8 | 4 | 0.50 | 8 | 1.00 | 23.6 | 12.8 | 0.54 | 14.2 | 0.60 |
| tri_sp_slv | 3.02 | 2.51 | 0.83 | 3.04 | 1.01 | 18.4 | 6.3 | 0.34 | 5.2 | 0.28 | 55.7 | 15.7 | 0.28 | 15.7 | 0.28 | 479 | 501 | 610 | 1.27 | 892 | 1.86 | 705 | 803 | 939 | 1.33 | 1106 | 1.57 | 6 | 6 | 6 | 1.00 | 6 | 1.00 | 26.7 | 9.6 | 0.36 | 14.0 | 0.53 |
| floyd_warshall | 2.28 | 2.77 | 1.22 | 2.90 | 1.28 | 14.0 | 3.3 | 0.23 | 2.3 | 0.16 | 31.9 | 9.1 | 0.29 | 6.7 | 0.21 | 477 | 542 | 568 | 1.19 | 787 | 1.65 | 713 | 862 | 954 | 1.34 | 1092 | 1.53 | 2 | 2 | 2 | 1.00 | 2 | 1.00 | 15.2 | 5.2 | 0.34 | 5.3 | 0.35 |
| **Geomean** | | | 1.33 | | 1.12 | | | 0.34 | | 0.24 | | | 0.46 | | 0.27 | | | | 1.25 | | 1.74 | | | | 1.28 | | 1.62 | | | | 0.82 | | 1.04 | | | 0.57 | | 0.47 |

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

### D. Resource Overhead

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. As shown in Table II, 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 $74\%$, $62\%$ and $4\%$ respectively. Due to much higher parallelism achieved with PolyDLP, more operations are required to work at the same time in the pipeline bodies compared to the pipeline schedule with PLP. The increase of LUTs and FFs is found to be also caused by the unshared address generators shown in Fig. 2. The resource sharing between the floating-point data paths is well supported by the HLS backend.

Since RT-Dep does not duplicate the loop body, it has less resource overhead than PolyDLP. In addition, after the transformation of RT-Dep, only the innermost dimension of the nested loop can be pipelined by Vivado HLS. This limitation also simplifies the pipeline scheduling problem, which will generally lead to less resource usage. However, resource overhead is still less significant than performance improvement in the conflict region. For PolyDLP, it is witnessed by a $53\%$ average reduction of the area-time product in Table II. In comparison, there is $10\%$ less average reduction with RT-Dep.

## V. Conclusion

In this paper, we proposed a new optimization method for a class of loops with uncertain and non-uniform memory dependencies. The method uses compiler-based analysis to generate high-performance runtime optimizations. The optimized pipelines can execute the loop iterations as fast as possible, when specific conditions are detected, or pipeline breaks are inserted at runtime. Compared to the approach of detecting and resolving memory conflicts at runtime (RT-Dep), our proposed optimisation is shown to be a better choice for those loops that can be analysed by the parametric polyhedral model. In the future, with the integration of both static and runtime optimisations, we aim to enable high-level synthesis to generate highly efficient and dynamically scheduled pipelines.

## Acknowledgments

We thank Alle *et al.* [13] for their help on making the detailed comparison to their work. The support of the EPSRC grants EP/I020357/1 and EP/K034448/1, the Royal Academy of Engineering, and Imagination Technologies is gratefully acknowledged.

## 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] 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.
[8] 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.
[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] Clang. [Online]. Available: http://clang.llvm.org
[16] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Int. Workshop on Polyhedral Compilation Techniques (IMPACT '12)*, 2012.
[17] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Proc. Int. Conf. on Mathematical Software (ICMS '10)*, 2010.
[18] The GeCoS (Generic Compiler Suite) Project. [Online]. Available: http://gecos.gforge.inria.fr/doku.php