Exploiting the Correlation between Dependence Distance and Latency in Loop Pipelining for HLS

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

***Abstract*—High-level synthesis (HLS) automatically transforms high-level programs in a language such as C/C++ into a low-level hardware description. In this context, loop pipelining is a key optimisation method for improving hardware performance. The main performance bottleneck of a pipelined loop is the ratio between two values: the latency of each iteration and the dependence distance of the operations in the loop. These two values are usually not known exactly, so existing HLS schedulers model them independently, which can cause sub-optimal performance. This paper extends state-of-the-art static schedulers with a fully automated pass that exposes and takes advantage of potential correlation between these two values, enabling smaller initiation intervals (II). We use the Microsoft Boogie software verifier to prove the existence of these correlations, which allows HLS tools to automatically find a high-performance hardware solution while maintaining correctness. Our results show that for a certain class of programs, our approach achieves, on average, an 11.1× performance gain at the cost of a 95% area overhead.**

***Index Terms*—High-Level Synthesis, Loop Pipelining, Formal Methods.**

I. Introduction

High-level synthesis (HLS) tools use loop pipelining as one of the most common optimisation methods [1], [2]. Loop pipelining allows multiple iterations of a loop to be executed concurrently. The same operation in two consecutive iterations has a time difference in clock cycles, also known as the initiation interval (II). A small II leads to high throughput of the hardware design, since it allows loop iterations to be executed at early time. Traditional modulo schedulers are not flow-sensitive, and therefore need to make conservative approximations that cover all possible control paths. In order to preserve correctness, the scheduler has to ensure that any data processed by an operation is always valid before the operation starts. Since the exact hardware behaviour can vary, the static scheduler has to assume the “worst case” for loop pipelining.

There are two properties of a loop that limit how small an II can be achieved.

- The first is the **iteration latency**; *i.e.*, the latency of each iteration. High iteration latencies can lead to high IIs, especially if an early operation in one iteration has to wait for a late operation in a previous iteration to complete. Iteration latencies can vary at run-time between iterations, but existing static schedulers just take the maximum value.
- The second is the **dependence distance**; *i.e.*, the number of iterations that separate an operation from its dependants. Low dependence distances can lead to high IIs, because they limit the number of iterations that can be safely overlapped. Dependence distances can vary between iterations, but existing static schedulers just take a constant distance of 1 if it is not a constant.

The observation that drives this paper is that *these two constraints may be correlated*, and then the hardware may be optimised if these correlations can be captured. For instance, a loop may not have the maximum iteration latency at the minimum dependence distance. In existing static scheduling approaches, the iteration latency and dependence are approximated independently [1], [2]. Ignoring such correlations can lead to sub-optimal performance in the hardware design.

Capturing these correlations for arbitrary code is challenging, since most scheduling techniques only are based on linear-programming formulation and restricted to a certain class of code patterns. This paper overcomes two challenges for HLS tools: 1) How to efficiently explore the correlations between these two constraints for arbitrary code? 2) How to bring the benefit of these correlations into loop pipelining?

Our main contributions include:

- A technique that describes the correlations between the iteration latency and the dependence distance for arbitrary loops in a Boogie program.
- A fully automated HLS pass that calls Vitis HLS and Boogie verifier to find a minimum loop II by formally proving the absence of dependence violation in a loop schedule.
- Analysis and results showing that on average the proposed approach achieved 11.1× performance gain and 95% area overhead on average over a set of benchmarks.

The rest of the paper is organised as follows: Sec. II gives a motivating example where our approach can find a significantly smaller II than Vitis HLS in loop pipelining. Sec. III revisits the related works. Sec. IV illustrates the formalisation of loop pipelining and Boogie program generation. Sec. V evaluates the effectiveness of our approach.

II. Motivating Example

This section presents a motivating example of how the iteration latency and the dependence distance can be correlated. Fig. 1 illustrates a program that contains a `for` loop named `loop_0`. In the loop, the program checks the given

```
1  double f(double a) {
2    return (((((a+0.64)*a+0.7)*a+0.21)
3      *a+0.33)*a+0.25)*a+0.125;
4  }
5  void example(double vec[M]){
6    loop_0:
7    for (int i = 0; i < N; i++){
8  #pragma HLS PIPELINE
9      double e = vec[i];
10     if (e > 0) vec[i+63] = f(e);
11     else vec[i*i+9] = e * e;
12   }
13 }
```

Fig. 1: Catapult HLS and Intel HLS Compiler fail to pipeline the loop. Vitis HLS achieves an II of 41. Traditional techniques such as loop splitting and polyhedral analysis cannot improve the performance. Our approach finds an optimal II of 2.

condition on the loop iterator `i`. If it is true, the array element at `vec[i+63]` is overwritten by a polynomial function of `vec[i]`, otherwise, `vec[i*i+9]` is over-written by the square of `vec[i]`.

`loop_0` is to be pipelined with an optimal II for the best performance. The iteration latency of the true path is 82 cycles, and the iteration latency of the false path is 5 cycles. The minimum memory dependence distance is 9. We investigated three well-known HLS tools in industry.

- Catapult HLS [3] and Intel HLS compiler [4] fail to pipeline the loop, requiring the user to manually find the optimal II. The output hardware contains a sequential loop, corresponding to an II of 82.
- Vitis HLS 2020.1 [5] pipelines the loop with an II of 41. The reason is that the scheduler cannot see the correlation between the two branches and conservatively approximates the control flow.

Taking Vitis HLS for example, if we force the tool to pipeline the loop with an II of 2, the resulting hardware still preserves correctness. With 2.55× LUTs and 3× DSPs, the computation of the hardware with an II of 2 achieves 22.2× speedup compared to the hardware with an II of 41. The motivation of our work is to find a smaller II by formally analysing these *hidden* correlations.

Extracting such correlations from arbitrary code for loop pipelining is challenging. For instance, polyhedral analysis does not support non-affine memory addresses, and loop splitting does not support data dependent conditions in a loop. Our tool translates the scheduling problem into a verification problem at high abstraction level that can be solved by a existing verifier named Boogie [6]. Replying on the existing techniques in Boogie verifier, the correlations between the iteration latency and dependence distance can be efficiently explored for loop pipelining.

III. Background

Loop pipelining has been well-studied in the past decades. Zhang and Liu [1] propose efficient scheduling methods on exploring memory dependences and resources. Canis *et al.* [2] further optimise the approach by reducing the recurrence when reducing IIs. They model the dependence distance and iteration latency independently, while we explore the correlations between these constraints. Polyhedral analysis for loop pipelining is also popular [7], [8]. Other program techniques [9]–[11] are also used for scheduling optimisation. However, these techniques all work under the same dependence approximation made by the scheduler, while we prove a new dependence approximation with a smaller set of dependences for certain applications. There are also works on dynamic pipelining which cost additional hardware [12]–[15] and our work does not add any area to the circuit apart from by increasing IIs.

Formal methods are commonly used in software verification. The Satisfiability-Modulo Theory (SMT)-based optimisation for HLS has been explored in memory banking [16], [17]. This paper investigates loop pipelining using Microsoft Boogie [6], an automatic program verifier built on top of SMT solvers. Boogie has its own intermediate verification language (IVL) to describe the program behaviour being verified. An SMT solver then reasons about the program behaviour, including the values that its variables may take. Encoding of verification as SMT queries is automatically performed by Boogie, hidden from the user. Here we list some Boogie structs that are used in this paper:

1) `if (*) {A} else {B}` tells the verifier that either branch might be taken non-deterministically.
2) `havoc x` assigns arbitrary values to a variable or an array `x`. This can be used to capture all the cases the program behaves.
3) `assert c` proves the condition `c` for all the values that the variables in `c` may take.

IV. Methodology

In this section, we first extend the formulation of loop pipelining and reduce it to a verification problem. Then we show how the automatically generated Boogie program describes loop behaviours. Finally, we demonstrate our tool flow on top of Vitis HLS and Boogie verifier.

A. General Loop Pipeling Formulation

For a given loop that contains a set of statements, S_1, S_2, S_3, ... S_K, the goal of scheduling is to determine a start time $t_{n,k}$ for every instance of every statement. Let $L_{n,k}$ be the latency of instance n of statement k in clock cycles. There are dependences between statements as shown in Eq. 1, *e.g.* instance n_2 of statement k_2 depends on instance n_1 of statement k_1. Therefore, a feasible schedule satisfies Eq. 2, where t and L are the start time and the latency of the instance.

$$D \subseteq N^4 : (k_1, k_2, n_1, n_2) \in D \tag{1}$$

$$\forall (k_1, k_2, n_1, n_2) \in D, t_{n_2,k_2} \geq t_{n_1,k_1} + L_{n_1,k_1} \tag{2}$$

```
1  procedure pickOneMemStmtFromExample
2  (vec:[int]double) returns (valid:bool, k:int,
3  addr:int, n:int, mode:memTy, array:int){
4    // loop_0: for (int i = 0; i < N; i++){
5    ...
6    if (*){  // e = vec[i];
7      valid = 1; k = 0; addr = i; n = i;
8      mode = LOAD; array = 0; return; }
9    if (*){ // if (e > 0)
10     if (*){  // vec[i+63] = f(e);
11       valid = 1; label = 1; addr = i+63; n =
         i;
12       mode = STORE; array = 0; return; }
13   } else {
14     if (*){  // vec[i*i+9] = e*e;
15       valid = 1; k = 2; addr = i*i+9; n = i;
16       mode = STORE; array = 0; return; } }
17   ...
18   // } // end of loop_0
19   valid = 0;
20   return;
21 }
```

(a) The program behaviour is described in Boogie.

```
1  procedure main(II:int){
2    // we consider that array vec has arbitrary values.
3    havoc vec;
4    P = II; // for a given II
5    // and any two (k, n) from the loop,
6    call valid_1,k_1,addr_1,n_1,mode_1,array_1 =
       pickOneMemStmtFromExample(vec);
7    call valid_2,k_2,addr_2,n_2,mode_2,array_2 =
       pickOneMemStmtFromExample(vec);
8    // both (k_1, n_1) and (k_2, n_2) are valid
9    if (!valid_1 || !valid_2) return;
10   // and access the same array and address
11   if (array_1 != array_2 || addr_1 != addr_2) return;
12   // and at most one (k, n) is a load
13   if (mode_1 == LOAD && mode_2 == LOAD) return;
14
15   // assume w.l.o.g. (k_1, n_1) is earlier than (k_2,
        n_2)
16   if (n_1 > n_2) return;
17   // their offset & latency satisfy Eq. 7
18   assert getOffset(k_2) >= getOffset(k_1) +
       getLatency(k_1) - P*(n_2-iter1);
19 }
```

(b) Dependence condition verification.

Fig. 2: Automatically generated Boogie query for the example in Fig. 1.

B. Modulo Scheduling Formulation

Typical HLS tools use modulo scheduling for loop pipelining [18]. In modulo scheduling, the time constraints and dependences are approximated to simplify the static analysis. The start times and latencies are restricted to be:

$$t_{n,k} = \alpha_k + Pn, \alpha_k \geq 0 \tag{3}$$

$$L'_k = \max_n L_{n,k} \tag{4}$$

where α_k is a constant for a statement as its offset time, and P is the initiation interval. The model assumes that a statement k always takes L'_k cycles to execute, where L'_k is an upper bound of $L_{n,k}$. This means that the total execution time of the loop is bounded by $\max_{k,n} = P(N-1)+\max_k(\alpha_k + L'_k) \approx PN$, where N is the total number of instances.

The dependences are restricted as D' as shown in Eq. 5, where d is $n1 - n2$, also known as the dependence distance, and D' is an approximated set of D. For example, instance n of statement S_{k_2} depends on the output from instance $n - d$ of statement S_{k_1} only if $(k_1, k_2, d) \in D'$, *i.e.* $D = \{(k_1, k_2, n_1, n_2)|(k_1, k_2, n_2 - n_1) \in D'\}$. The dependence constraints that need to be solved are approximated as Eq. 6. Substituting the approximation of modulo scheduling in Eq. 3 into Eq. 2, the dependence constraint becomes:

$$\alpha_{k_2} + Pn_2 \geq \alpha_{k_1} + Pn_1 + L_{n_1,k_1}$$

This is then reformulated to:

$$\alpha_{k_2} \geq \alpha_{k_1} + P(n_1 - n_2) + L_{n_1,k_1}$$

The approximation of modulo scheduling Eq. 4 restricts the dependence constraint into:

$$\alpha_{k_2} \geq \alpha_{k_1} + P(n_1 - n_2) + L'_{k_1}$$
$$\geq \alpha_{k_1} + P(n_1 - n_2) + L_{n_1,k_1}$$

Modulo scheduling assumes $d = \min_{n_1,n_2}(n_2 - n_1)$, so the dependence constraint becomes:

$$D' \subseteq N^3 : (k_1, k_2, d) \in D' \tag{5}$$

$$\forall (k_1, k_2, d) \in D', \alpha_{k_2} \geq \alpha_{k_1} + L'_{k_1} - Pd \tag{6}$$

Eq. 4 restricts L' only depending on a set of L, and Eq. 5 restricts D' only depending on d. Such over-approximations can limit the capability of schedulers to solve the problem found in Sec. II.

C. Our Formulation

A novelty of our formulation lies in changing the constraints from Eq. 6 to Eq. 7. We introduce $D'' \subseteq D \subseteq N^4$ that depends on L to support variable dependence distance. The dependence constraints are then extended to:

$$\forall (k_1, k_2, n_1, n_2) \in D'', \alpha_{k_2} \geq \alpha_{k_1} + L'_{k_1} - P(n_2 - n_1) \tag{7}$$

The condition of the traditional modulo scheduling causes sub-optimal performance is:

$$\max_{n_1,n_2,k_1} (L'_{k_1} - P(n_2 - n_1)) < \max_{k_1} L'_{k_1} - P \min_{n_1,n_2} (n_2 - n_1) \tag{8}$$

The right-hand side is the independent approximation in traditional scheduling, and the left-hand side is the approximation including the correlation between the iteration latency and dependence distance. We keep the formulation of L' in Eq. 4 from Vitis HLS, and use Boogie to automatically extract D'' from our translation and prove Eq. 7.

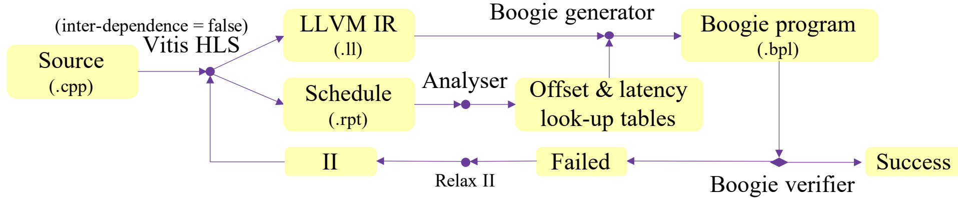

Fig. 3: Our tool flow using Boogie to find the minimum II.

TABLE I: Overall results on four benchmarks. Our approach gains better performance compared to Vitis HLS. F_{max} represents the maximum frequency, in MHz. t_w represents the wall clock time, in μs

	Vitis HLS by default							Our approach						
	II	LUTs	DSPs	Registers	F_{max}	Cycles	t_w	II	LUTs	DSPs	Registers	F_{max}	Cycles	t_w
vecTrans	81	2530	14	1220	118	72902	617	2	5961	42	4212	135	1881	13.9
loopCond	52	3168	8	610	139	52002	373	3	3690	8	1681	139	3050	21.9
dist_itr	1*	2245	14	1176	126	50961	406	11	2381	14	1920	127	10822	85.1
quard	41	2277	14	1427	118	4142	35	2	2595	39	3445	135	377	2.8
geom. mean	-	1×	1×	1×	1×	1×	1×	-	**1.43×**	**1.95×**	**2.56×**	**1.07×**	**0.10×**	**0.09×**

* Vitis HLS automatically infers an II of 1 and fails the simulation. Therefore, we provide the result of sequential hardware for `dist_itr` here.

D. Boogie Program Generation

In a loop, the dependence distance of a non-trivial data recurrence is always 1, which is well-handled by existing schedulers. Only memory statements can have variable dependence distances, so the memory dependence needs to be analysed. Our tool automatically translates these memory statements into a Boogie program, following three steps:

1) Program slicing: Instructions that do not affect the memory accesses are not translated into Boogie, such as function `f`, because only the values of memory addresses are required for dependence analysis.

2) Behaviour/Dependence Description: The sliced program is then transformed into a Boogie program to prove Eq. 7. Fig. 2 illustrates part of Boogie description for the motivating example in Fig. 1. The procedure `pickOneMemStmtFromExample` in Fig. 2a returns an arbitrary (k, n) from a loop where statement k is always a memory statement. We reformulate each (k, n) as a 6-tuple for modelling D'':

valid: validity of (k, n). It is invalid after the loop finishes.
k & n: the value of k & the value of n.
addr: the array address.
mode: whether the statement is a load or a store.
array: the array accessed by this statement.

The loop structure is modelled using an existing tool named EASY [17] based on the formulation by Chong [19], which can capture the behaviour of any arbitrary loop iteration. For simplicity, we only show the loop body as one of our contributions. Each data-dependent condition is approximated into a non-deterministic condition `if(*)` like line 8 so the verifier tries to prove Eq. 7 for both cases. Each memory statement is also formulated as a 6-tuple and arbitrarily returned using `if(*)`. Therefore, the procedure `pickOneMemStmtFromExample` arbitrarily returns one memory access in the loop across all the iterations.

3) Assertion Description: The other part of the Boogie program in Fig. 2b proves Eq. 7 for two different (k, n). Firstly, two `call` instructions return two arbitrarily (k, n) in 6-tuples. Then the Boogie program ignores the cases of the independent set $\{(k_1, k_2, n_1, n_2)\} \setminus D''$ with three conditions: 1) Any two (k, n) must be both valid; 2) they should access the same array at the same address; 3) have at most one write so the dependence exists. We assume (k_2, n_2) is after (k_1, n_1) so given the offset time α and latency L' of these two accesses, the assertion on line 18 which describes Eq. 7 *always* holds if the given II is reachable.

E. Tool Flow

The tool flow of our work that takes Vitis HLS for prototyping is shown in Fig. 3. Our tool only uses Vitis HLS for obtaining offsets and latencies in Eq. 3. Then the Boogie verifier proves whether the schedule satisfies the constraints with the given II. If failed, the II is relaxed and the verification is repeated with the corresponding offsets and latencies until an successfully proved II is found. Our tool uses a search sequence that checks II < 5 first and then binary search, and also allows users to customise the II search region. The iterative process uses Vitis HLS only for getting the offsets and latencies instead of synthesis.

V. EXPERIMENTS

Our approach is beneficial for a certain class of applications that contains control flows in a loop. Most loops in existing benchmark sets, such as Polybench [20] and CHStone [21], are amenable for existing HLS tool flow because of the absence of Eq. 8, such that Vitis HLS can already generate optimal

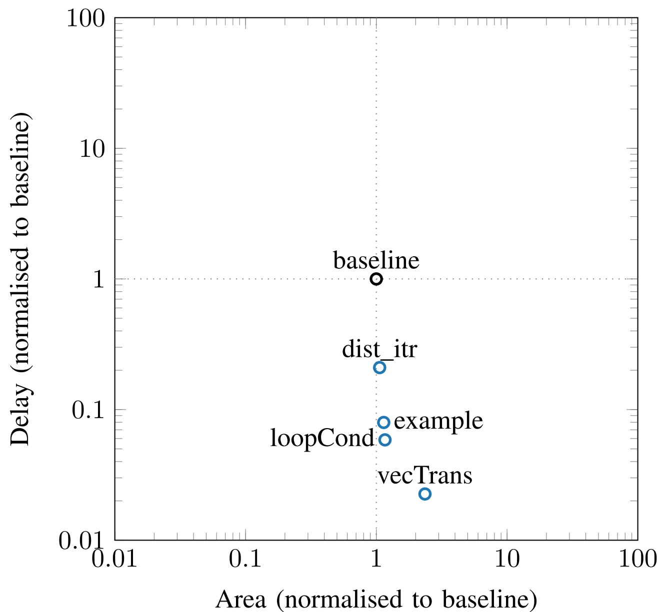

Fig. 4: Relative area and delay of our designs compared to the baseline.

schedules. Instead we investigate four benchmarks that have the code patterns in realistic applications[1] where Eq. 8 exists.

vecTrans performs conditional matrix transformation on a partitioned array.

loopCond contains a loop pattern with an `if` condition that performs different operations depending on the loop iterator, which can be found in stencil computation [23].

dist_itr conditionally assigns a polynomial function of array data to another affine address of the same array, which can be seen in matrix decomposition and triangular matrix computation [8], [24].

quard as part of `tramp3d-v4` benchmark has a non-linear memory access pattern that increments the array index in a quadratic form [25].

Existing approaches, *e.g.* polyhedral techniques including loop splitting, can only benefit restricted loops that have simple control flow, which produced the same results as Vitis HLS for these benchmarks. We compare our results with the corresponding design automatically inferred from Vitis HLS. We assess our work on both the circuit area and the wall clock time from Vitis. The FPGA family we used for measurements is xc7z020clg484, and the version of Vitis software is 2020.1.

Fig. 4 illustrates the relative area and delay of our designs compared to the baseline. All the benchmarks are on the bottom right of the baseline point, indicating our approach achieves higher performance with affordable area overhead. Tab. I shows the detailed results of the four benchmarks. All the benchmarks either cannot be efficiently pipelined by Vitis HLS automatically, or gives *wrong* results as the scheduler gives an over-optimistic II. The cycle counts are significantly reduced by our work due to smaller IIs with area overheads. The maximum frequency does not change since both approaches use Vitis HLS for retiming. On average, we achieve $11.1\times$ speedup with $1.95\times$ area. If the condition does not exist in the input code, our approach can still give the same II as the II automatically determined by Vitis HLS, without losing any performance or area. The time overhead in our tool is neglectable compared to the synthesis time in Vitis HLS, which are for generation and verification of a Boogie program and interfacing the scheduling process in Vitis HLS. The average additional time is 2 minutes in our experiments.

[1]The source of our tool and benchmarks are available at [22].

VI. Conclusions

In HLS tools, existing static scheduling approaches approximate the distance distances and iteration latencies independently for loop pipelining. We show in a certain class of applications, correlating these two constraints in scheduling can significantly improve the hardware performance. Our work supports automatic translation for arbitrary code into Boogie programs, and formally proves the correctness of the schedule with a smaller II. Our future work is to explore the limits of static analysis in dynamic scheduling.

Acknowledgements

This work is supported by the EPSRC (EP/P010040/1, EP/R006865/1).

References

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

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

[3] Catapult High-Level Synthesis, 2020. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[4] Intel HLS Compiler, 2020. [Online]. Available: https://www.altera.com/

[5] Xilinx Vitis HLS, 2020. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/index.html

[6] K. R. M. Leino, “This is boogie 2,” June 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

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

[8] 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)*, 2016, pp. 72–79.

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

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

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

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

[13] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. Monterey, CA: ACM, 2018, pp. 127–136.

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

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

[16] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A new approach to automatic memory banking using trace-based address mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 179–188. [Online]. Available: https://doi.org/10.1145/3020078.3021734

[17] J. Cheng, S. T. Fleming, Y. T. Chen, J. H. Anderson, and G. A. Constantinides, "EASY: Efficient Arbiter SYnthesis from Multi-threaded Code," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. Seaside, CA, USA: ACM, 2019, pp. 142–151.

[18] 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: Association for Computing Machinery, 1994, p. 63–74. [Online]. Available: https://doi.org/10.1145/192724.192731

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

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

[21] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*, 2008, pp. 1192–1195.

[22] II Prover, 2021. [Online]. Available: https://github.com/JianyiCheng/iiProver

[23] W. Altoyan and J. J. Alonso, "Investigating performance losses in high-level synthesis for stencil computations," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 195–203.

[24] T. A. Davis, "Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.

[25] Test Suites, 2021. [Online]. Available: https://github.com/microsoft/test-suite