Subgraph Extraction-based Feedback-guided Iterative Scheduling for HLS

Hanchen Ye*¶, David Z. Pan†§, Chris Leary‡, Deming Chen*, Xiaoqing Xu§
*University of Illinois Urbana-Champaign, †The University of Texas at Austin, ‡Google, §X, the moonshot factory
hanchen8@illinois.edu, dpan@ece.utexas.edu, leary@google.com, dchen@illinois.edu, xiaoqingxu@x.team

***Abstract*—This paper proposes ISDC, a novel feedback-guided iterative system of difference constraints (SDC) scheduling algorithm for high-level synthesis (HLS). ISDC leverages subgraph extraction-based low-level feedback from downstream tools like logic synthesizers to iteratively refine HLS scheduling. Technical innovations include: (1) An enhanced SDC formulation that effectively integrates low-level feedback into the linear-programming (LP) problem; (2) A fanout and window-based subgraph extraction mechanism driving the feedback cycle; (3) A no-human-in-loop ISDC flow compatible with a wide range of downstream tools and process design kits (PDKs). Evaluation shows that ISDC reduces register usage by 28.5% against an industrial-strength open-source HLS tool.**

I. Introduction

Scheduling is one of the most important problems in high-level synthesis (HLS) that partitions a computation graph into multiple clock cycles under the given timing and resource constraints. In 2006, Cong and Zhang [1] proposed a scheduling algorithm based on a system of difference constraints (SDC) formulation, converting the scheduling problem into a linear programming (LP) problem that can be solved optimally in polynomial time. SDC scheduling marked an important milestone for HLS and has been widely adopted in HLS tools, including AMD Vitis HLS [2], LegUp [3], and Google XLS [4].

Over the years, both industrial and academic HLS tools [2]–[4] have been relying on high-level intermediate representation (IR), such as LLVM IR [5], for timing analysis, area/resource analysis, and scheduling. In this context, the IR operations, such as integer additions and multiplications, are viewed as the fundamental elements to schedule against. Their delays and resources are pre-characterized in isolation through downstream tools, such as logic synthesizer [6], [7], for the target technology library. While this can capture some low-level characteristics of individual operations, it does not model further optimizations in downstream tools, such as logic resubstitution and rewriting, leading to estimations that are substantially different from the actual quality of results (QoR) [8].

To study this phenomenon, we generated 6912 different design points of an HLS design with Google XLS [4] and profiled their post-synthesis static timing analysis (STA) and XLS-estimated critical path delays. We used Yosys [6] and OpenSTA [9] for logic synthesis and STA. We used SKY130 [10] as the target technology library. Fig. 1 shows the profiling results. We can observe the XLS-estimated delays (blue dots) exhibit significant deviation from the STA delays (the green line),

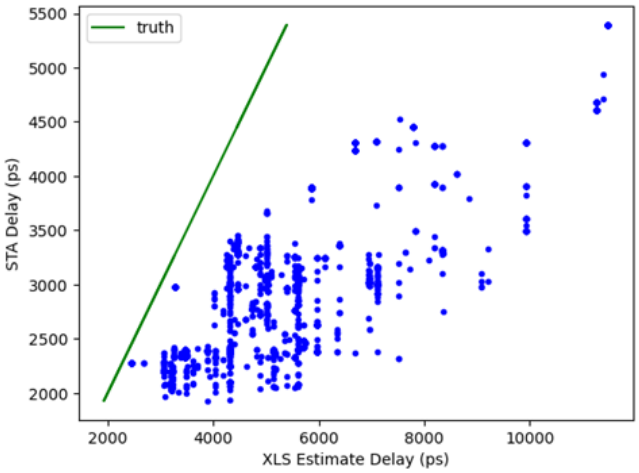


Fig. 1: Post-synthesis STA vs. XLS-estimated critical path delay of 6912 different HLS design points.

which are treated as the ground truth for this experiment. These deviations create unused slack and present numerous opportunities to refine scheduling quality, such as reducing register usage. However, without access to low-level information, HLS tools cannot effectively capitalize on these opportunities.

Feedback-directed optimization (FDO) has been widely adopted in software compilation and shown significant benefits [11]. However, this idea has not been thoroughly studied in the HLS domain. Lucas et al. [12] presented a process variation and layout-aware HLS binding algorithm, which aimed to improve the performance yield of generated designs by enhancing the HLS binding process. Zheng et al. [13] introduced a placement and routing (PAR) directed HLS flow, which heavily depends on back annotations from a specific proprietary synthesis and PAR tool, limiting its adaptability to other scenarios. Tan et al. [14] presented a mixed integer linear programming (MILP) formulation for technology mapping-aware HLS scheduling. While it considers the mapping of original operations into look-up tables (LUTs), it cannot capture inter-operation optimizations present in logic synthesis and beyond. Rizzi et al. [15] introduced an iterative technology mapping-aware register placement algorithm. Nonetheless, its focus remains primarily on the LUT-mapping of FPGA targets and dynamically scheduled dataflow circuits.

In this paper, we introduce low-level feedback into the HLS scheduling problem and propose ISDC, a novel iterative SDC scheduling method. The main contributions are:

¶Work was done when interning at X, the moonshot factory.

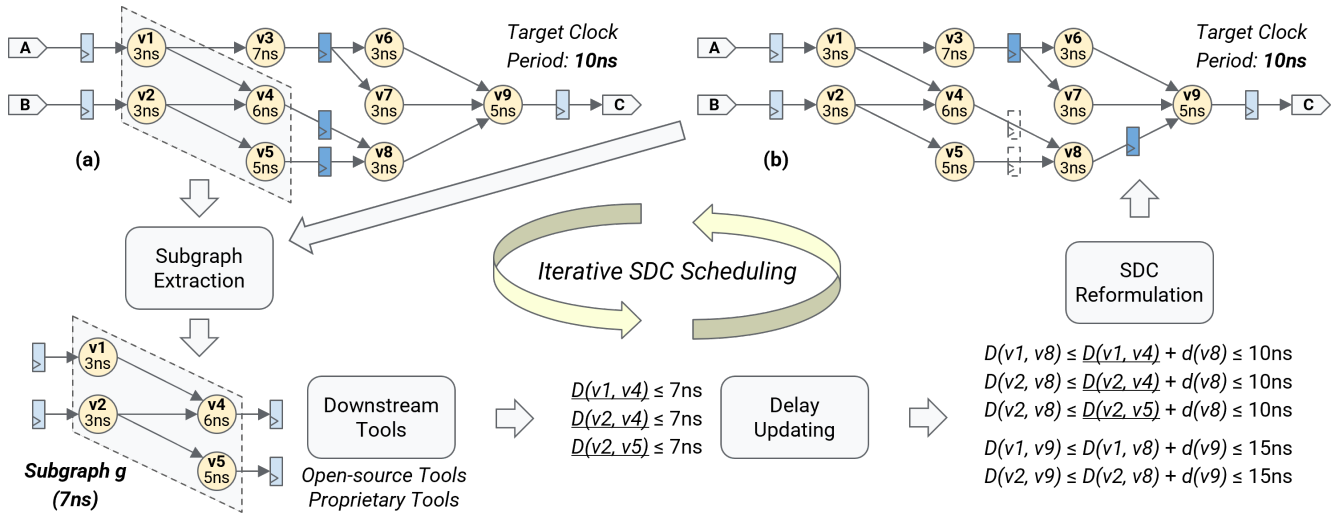


Fig. 2: Overall flow of ISDC scheduling algorithm.

- A scheduling algorithm that leverages feedback from downstream tools to refine the scheduling result iteratively and reduce register usage.
- An enhanced SDC formulation that effectively integrates low-level feedback into the LP problem.
- A fanout-driven and window-based subgraph extraction method that improves the quality and convergence speed of iterative scheduling.
- A no-human-in-loop ISDC workflow compatible with a wide range of downstream tools and PDK. ISDC is fully open-sourced at https://github.com/google/xls.

II. Preliminaries

The HLS IR to be scheduled is typically represented as a directed graph G. For each operation node v in graph G, SDC scheduling [1] defines a variable s_v to represent the time step in which the operation is scheduled into. By ensuring constraints in integer-difference form, such as:

$$s_u - s_v \leq d_{u,v} \tag{1}$$

where $d_{u,v}$ is an integer, a totally unimodular constraint matrix is derived, which is guaranteed to have integral solutions [16]. A set of common HLS constraints can be expressed in the form of integer-difference constraints [16]. Specifically, to meet the target clock frequency, a timing constraint is used to constrain the maximum combinational delay within a clock cycle. For the *critical combinational path (CCP)* connecting v_{i_1} and v_{i_k} with the largest delay, we can calculate its delay $D(ccp(v_{i_1}, v_{i_k}))$ as $\sum_{s=1}^{k} d(v_{i_s})$, where $d(v)$ is the individual delay of v. For each operation pair v_i and v_j with $D(ccp(v_i, v_j)) > T_{clk}$, where T_{clk} is the target clock period, we construct a constraint as:

$$s_{v_i} - s_{v_j} \leq -\left(\left\lceil \frac{D(ccp(v_i, v_j))}{T_{clk}} \right\rceil - 1\right) \tag{2}$$

Eq. 2 states that the combinational path with total delay exceeding the target clock period T_{clk} must be partitioned into at least $\lceil D(ccp(v_i, v_j))/T_{clk} \rceil$ number of clock cycles.

III. Iterative SDC Scheduling

A. Overall Flow

Fig. 2 shows the overall flow of the proposed ISDC scheduling algorithm. ISDC starts from an initial pipeline, as depicted in Fig. 2(a), scheduled with the original SDC scheduling algorithm [1]. Note that each node in Fig. 2(a) represents an operation of the HLS IR, such as integer additions and multiplications. On top of this initial schedule, a set of combinational subgraphs, such as subgraph g at the lower-left of Fig. 2, are extracted and passed to downstream tools for subgraph logic synthesis and beyond. Subsequently, the subgraph delays fed back from downstream tools are integrated into an enhanced SDC formulation to construct an updated LP problem. Upon solving this LP problem, a new pipeline schedule is generated as depicted in Fig 2(b). This procedure is then iteratively applied to the new pipeline schedule until a stable scheduling result is achieved, exemplified by metrics such as register usage.

1) Why low-level feedback helps: As shown in Fig. 2(a), the initial estimation of $D(ccp(v_2, v_8))$ is calculated as $d(v_2) + d(v_4) + d(v_8)$, which totals to 12ns. Given the target clock period of 10ns, v_2 and v_8 must be scheduled into separate clock cycles. However, suppose the delay of subgraph g reported by downstream tools is 7ns, $D(ccp(v_2, v_8))$ can be recalculated as $D(g) + d(v_8)$, equaling to 10ns. As a result, v_8 can now be merged into the same clock cycle as v_2, leading to a decrease in register usage as depicted in Fig. 2(b). This underscores the significance of low-level feedback in refining scheduling result. Such feedback empowers ISDC to identify better design points that might have been erroneously overlooked by the original SDC scheduling algorithm.

2) Why an iterative approach helps: Considering the real-world constraints of computational resources, it is infeasible to evaluate every subgraph in an HLS design for feedback, especially given the exponential increase in complexity as the HLS design grows. By using an iterative approach, ISDC can capitalize on knowledge from prior iterations, substantially reducing the search space of subgraph extraction by focusing

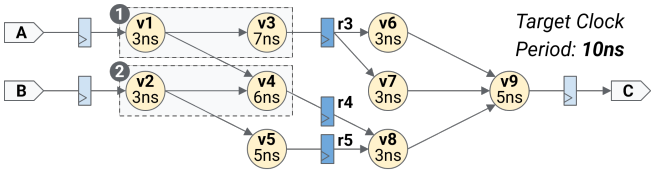


Fig. 3: Delay-based vs. fanout-based subgraph extraction.

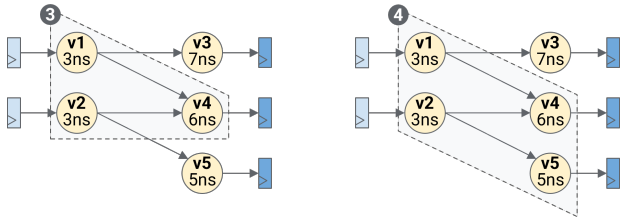


Fig. 4: Cone-based vs. window-based subgraph extraction.

on *combinational* subgraphs from the previous schedule. This approach helps ISDC incrementally refine the scheduling result, maintaining manageable computational complexity throughout each iteration.

B. Subgraph Extraction Strategy

Despite using an iterative approach, the number of subgraph candidates remains vast, which can readily result in slow convergence. In this section, we introduce two orthogonal strategies to address this problem. Their ablation studies are presented in Section IV-A.

1) Fanout-driven strategy: A direct and intuitive extraction strategy is delay-driven: focusing on the longest paths from the previous schedule because of their impact on the achievable clock frequency. Nonetheless, we argue that relying solely on delay is not the most effective strategy. As illustrated in Fig. 3, path ① is the longest combinational path with a delay of 10ns. But the register associated with path ①, specifically r_3, is utilized by two consumer nodes, v_6 and v_7. Merging the two nodes into the first clock cycle would increase register usage, being not beneficial. In comparison, although path ② has a shorter delay of 9ns, its associated register r_4 only has a single consumer, offering greater flexibility in its positioning.

Essentially, the more a register is being utilized, the more critical it becomes, reducing the benefit of repositioning it. Therefore, we introduce the following metric to drive the subgraph extraction process:

$$S(v_i, v_j) = \sum_{s=1}^{k} \frac{bit\_count(r_s(v_j)) + \frac{D(ccp(v_i,v_j))}{T_{clk}}}{num\_users(r_s(v_j)) + 1} \quad (3)$$

Assuming v_j produces a total of k results, $r_s(v_j)$ denotes the s-th result of v_j. The function bit_count quantifies the significance of $r_s(v_j)$, while num_users captures the degree to which $r_s(v_j)$ is utilized. $D(ccp(v_i, v_j))/T_{clk}$ serves as a tie-breaker, and is ensured to be less than 1.0 in any valid schedule. Suppose m subgraphs are extracted in each iteration, ISDC sorts all combinational paths from the previous schedule in descending order of $S(v_i, v_j)$ and extract the top m paths. Given that num_users can be viewed as the HLS IR level fanout, we term this approach the *fanout-driven* strategy.

Algorithm 1 Pseudo code of delay updating

Require: G, the graph to be scheduled
Require: $S[m]$, all evaluated subgraphs
Require: $D[n][n]$, the past critical path delay of all node pairs
Ensure: Updated $D[n][n]$, the critical path delay of all node pairs

```
1:  if is_first_iteration() then                          ▷ Initialize D
2:      for u in get_nodes(G) do
3:          for v in get_nodes(G) do
4:              if u == v then
5:                  D[u][v] ← d(v)                      ▷ Get individual delay
6:              else if is_connected(u, v) then
7:                  D[u][v] ← D(ccp(u, v))              ▷ Get critical path delay
8:              else
9:                  D[u][v] ← −1                        ▷ Annotate as not connected
10: for g in S do                                         ▷ Traverse all evaluated subgraphs
11:     for u in get_nodes(g) do
12:         for v in get_nodes(g) do
13:             if D[u][v] > get_delay(g) and D[u][v] ≠ −1 then
14:                 D[u][v] ← get_delay(g)                ▷ Update critical delay
```

2) Window-based strategy: The motivation of introducing feedback is to capture the low-level optimizations in downstream tools. To better capture inter-node optimizations, ISDC expands the paths identified in Section III-B1 to *cone*s and *window*s. Here, a cone is defined as a set of nodes at the HLS IR level with multiple input nodes (*leaves*) and a single output node (*root*). A cone must adhere to the following properties: (1) Each path from any primary input (PI) of graph G to $root$ passes though a leaf; (2) For each leaf, there exists a path from a PI to $root$ that passes though that specific leaf and bypasses any other leaves. To expand a given path between nodes v_i and v_j into a combinational cone, ISDC uses a depth-first search (DFS) algorithm that recursively identifies the preceding nodes of v_j until it encounters the boundary nodes of clock cycles or the PI of the entire graph G.

A window is derived by merging multiple cones that have different roots but share an identical or overlapping set of leaves. While a window still adheres to the properties above, it extends them to the case of multiple output nodes. Fig. 4 shows an example of expanding path ② in Fig. 3 to a cone (subgraph ③) and a window (subgraph ④). Given that cone/window-based optimizations are prevalent in logic synthesis [7], the cone/window subgraphs can capture the most relevant inter-operation optimizations, while also being sufficiently self-contained to mitigate the potential of over-optimization.

C. Delay Updating

In the initial SDC scheduling phase, ISDC employs the method outlined in Section II to calculate the critical path delay for every node pair and set timing constraints. To integrate the low-level feedback into the subsequent SDC formulations, ISDC maintains a matrix $D[n][n]$ that holds the estimated critical path delay of all node pairs, where n denotes the total node count. In each iteration, ISDC updates $D[n][n]$ with Alg. 1 once the subgraph delays are fed back from downstream tools. Lines 1 to 9 initialize $D[n][n]$ with the naive delay estimations. Subsequently, lines 10 to 14 traverse all evaluated subgraphs. For each subgraph g, the delay of all node pairs covered by g is updated with $D(g)$, but only if $D(g)$ is shorter than the original

Algorithm 2 Pseudo code of SDC reformulation

Require: G, the graph to be scheduled
Require: T_{clk}, target clock period
Require: Updated $D[n][n]$, the critical path delay of all node pairs
Ensure: M, the reformulated SDC model

```
1: V ← get_nodes(G)
2: for v in topo_sort(V) do
3:     D_v[n] ← new([−1] × n)
4:     for p in get_operands(v) do          ▷ Traverse all operands of v
5:         for u in V do
6:             if D[u][p] ≠ −1 then
7:                 if D_v[u] < D[u][p] + D[v][v] then
8:                     D_v[u] ← D[u][p] + D[v][v]
9:     for u in V do
10:        if D_v[n] ≠ −1 then
11:            if D[u][v] > D_v[n] or D[u][v] == −1 then
12:                D[u][v] ← D_v[n]          ▷ Update critical path delay
13: for u in reverse_topo_sort(V) do
14:    D_u[n] ← new([−1] × n)
15:    for c in get_users(u) do              ▷ Traverse all users of u
16:    ... ...                               ▷ Reversed delay propagation
17: M ← initialize_sdc(V)
18: for u in V do
19:    for v in V do
20:        if D[u][v] > T_clk then
21:            add_timing_constraint(M)      ▷ Set Eq. 2 to M
22: add_other_constraint(M)
```

delay estimation. Consequently, ISDC maximally leverages the information obtained from each subgraph evaluation, thereby accelerating the iterative convergence.

D. SDC Reformulation

Upon the updated delay matrix $D[n][n]$, all the timing constraints discussed in Section II are reformulated to construct an updated LP problem. Essentially, this reformulation can be viewed as an all-pairs shortest path problem, optimally solved by Floyd-Warshall algorithm with a complexity of $O(n^3)$. To mitigate this cubic complexity, we propose an $O(n^2)$ algorithm as Alg. 2, which provides a sufficiently accurate delay estimation for our purposes. The estimation accuracy study is presented in Section IV-B. Lines 2 to 12 traverse all nodes of graph G in a topological order, ensuring that a node is processed only after all its operand nodes. For a specific node v, lines 4 to 8 calculate the delay from all nodes to v by adding v's individual delay to the delay from all nodes to v's operand nodes. Lines 7 to 8 ensure only the critical path delay is recorded. Subsequently, lines 9 to 12 update $D[n][n]$ only if the newly calculated delay is shorter.

After this topological order traversal, lines 13 to 16 of Alg. 2 reprocess all nodes, but in a reversed topological order. This step aims to identify the complementary paths that cannot be identified by the initial topological order traversal. Finally, lines 18 to 21 set the timing constraints for the LP problem based on the recalculated $D[n][n]$. Intuitively, by reformulating the SDC problem, ISDC prunes the over-conservative timing constraints that were erroneously set in the previous SDC scheduling. This enlarges the updated LP problem's search space, naturally leading to a refined scheduling result.

IV. Evaluation

We implemented the proposed ISDC algorithm on top of an industrial-strength open-source HLS tool, Google XLS [4], which uses SDC scheduling [1] as the default scheduling algorithm. We used Yosys [6] and OpenSTA [9] for logic synthesis and STA. We used open-source SKY130 [10] as the target technology library.

A. Ablation Study

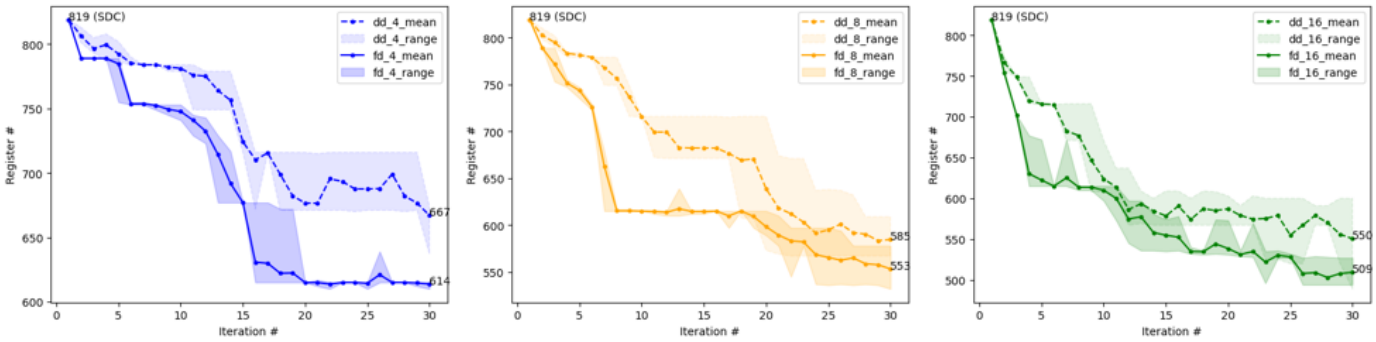
We performed a set of ablation studies on an XLS-based HLS design to demonstrate the efficacy of the proposed fanout-driven and window-based subgraph extraction strategy.

1) Fanout-driven strategy: Fig. 5 shows the comparisons between the delay-driven (dd) and fanout-driven (fd) strategies. We performed 30 iterations of scheduling under 400MHz clock frequency, where 4, 8, or 16 subgraphs were extracted per iteration. The results indicate that the fanout-driven strategy converges notably faster than its delay-driven counterpart, particularly in the initial iterations. Furthermore, it consistently achieves lower register usage across all three cases.

2) Window-based strategy: Fig. 6 shows the comparisons among the path, cone, and window-based strategies. Notably, the cone/window-based strategies demonstrate faster convergence than the path-based approach, achieving a reduced register usage. Path-based strategy is often trapped in local minima. In contrast, the cone/window-based strategy can overcome those points, achieving further improvements in subsequent iterations. While the cone and window-based strategies exhibit similar performance, the results suggest a slight edge for the window-based approach. Meanwhile, as expected, ISDC converges faster with the extraction and evaluation of more subgraphs per iteration.

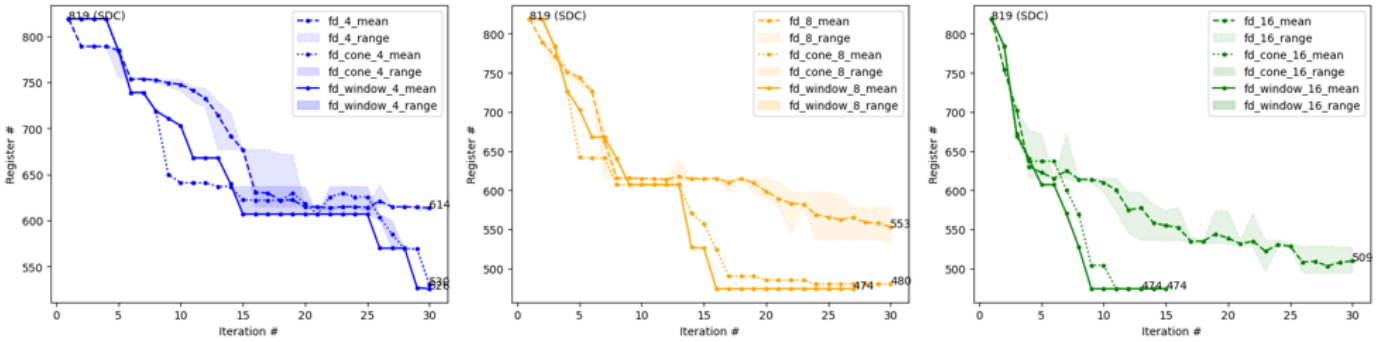
B. Benchmarking Results

We performed benchmarking on 17 XLS-based HLS designs to evaluate ISDC. The benchmarks encompass common algorithms like *crc32*, as well as datapaths from industrial SoCs, including an machine learning processor (*ML-core*) and a video processor (*video-core*). In the evaluation, we used the fanout-driven and window-based strategy, evaluating 16 subgraphs per iteration in parallel. A total of 15 iterations were performed on each benchmark. Tab. I shows the evaluation results, including metrics such as the target clock period, post-synthesis slack, number of pipeline stages, number of registers, and scheduling runtime. By default, we set the target clock period to 2500ps to constrain the scheduling. If an operation in a benchmark exhibited an individual delay exceeding 2500ps, we adjusted the target clock period to 5000ps. On average, ISDC achieves a 28.5% lower register usage compared to the original SDC scheduling. This comes at the cost of an average 40.8× increase in scheduling runtime. For instance, for the largest benchmark, *sha256*, ISDC spends around 54.7 minutes to converge, which is 11.5× longer than the original SDC's 4.7 minutes. Among all benchmarks, ISDC utilized 39.1% additional slack in average to make room for the reduction in register usage. However, there are several counter examples, such as *ML-core datapath0*



(a) 4 subgraphs per iteration. (b) 8 subgraphs per iteration. (c) 16 subgraphs per iteration.

Fig. 5: Ablation study of delay-driven and fanout-driven subgraph extraction. Path-based strategy is used.



(a) 4 subgraphs per iteration. (b) 8 subgraphs per iteration. (c) 16 subgraphs per iteration.

Fig. 6: Ablation study of path, cone, and window-based subgraph extraction. Fanout-driven strategy is used because it has produced better results than the delay-driven strategy as shown in Fig. 5.

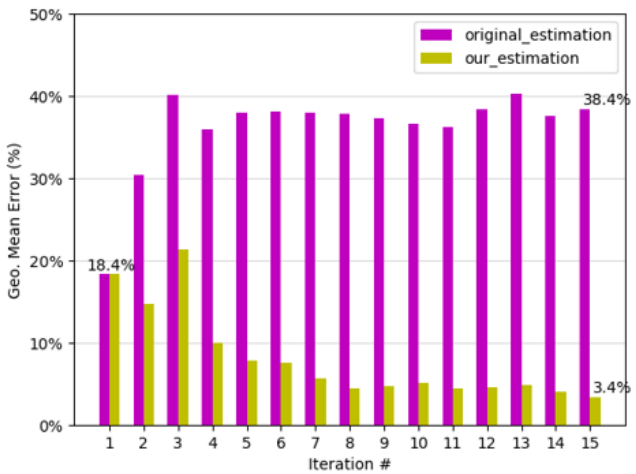


Fig. 7: Delay estimation accuracy comparison.

opcode0, which exhibits a slight increase in slack but still achieves a register usage reduction.

To evaluate ISDC's delay estimation accuracy, we analyzed its estimation across the 17 benchmarks and compared with the original SDC. Fig. 7 shows the comparison results. In the first iteration, without low-level feedback, ISDC exhibits the same estimation error as the original SDC. However, as the iterations progress, ISDC gradually reduces its estimation error, ultimately reaching an error of 3.4%. In contrast, the original SDC's estimation error increases. We attribute this to the fact that as the scheduling results are refined, more low-level optimizations are overlooked by the original SDC.

V. Discussion

1) Process node: Though we used real-world benchmarks for evaluation, we evaluated them down-clocked and on an older open-source industry process node (SKY130) to pioneer the methodology. We expect that the improvements should apply as effectively to more advanced process nodes and proprietary tools that offer similar STA report facilities.

2) Retiming: Retiming [17] is a method that repositions registers in gate-level sequential circuits to optimize performance or reduce resource usage without altering the overall functionality. On the other hand, HLS scheduling operates at higher-level IRs composed with algebraic operations and explicit control flows. This provides HLS scheduling with greater flexibility and larger design space to find more optimized designs. Furthermore, HLS scheduling preserves the algebraic attributes in the generated circuits, paving the way for robust verification processes, such as logic equivalence checking. This mitigates the limitations inherent in the retiming technique.

3) Runtime: A common concern of feedback-guided approaches is runtime. While the results in Tab. I demonstrate that ISDC converges at a practical pace, we have also explored a more aggressive strategy using the and-inverter-graph (AIG) to guide the scheduling. AIG is a widely adopted representation

TABLE I: Benchmarking results on 17 XLS-based HLS designs.

Benchmark	Clock Period (ps)	XLS [4] (SDC Scheduling)				Ours (Iterative SDC Scheduling)				
		Slack (ps)	Stage Num.	Register Num.	Schedule Time (s)	Slack (ps)	Stage Num.	Register Num.	Schedule Time (s)	Iteration Num.
ML-core datapath1	2500	1161.65	2	99	0.14	729.72	1	50	6.73	3
ML-core datapath0 opcode4	5000	943.93	2	109	0.11	943.93	2	109	0.10	1
rrot	2500	866.23	2	192	0.08	499.33	1	96	2.98	2
ML-core datapath0 opcode3	5000	1440.65	3	138	0.13	772.87	2	101	23.90	6
binary_divide	2500	518.66	3	71	0.12	436.18	3	70	7.56	4
hsv2rgb	5000	1450.73	3	134	0.11	1149.73	2	102	10.64	3
ML-core datapath0 opcode0	5000	1140.9	3	162	0.12	1162.66	2	108	19.26	4
crc32	2500	1744.35	3	75	0.11	1686.49	1	38	4.76	3
ML-core datapath0 opcode1	5000	1235.58	5	298	0.15	1519.2	4	234	21.28	4
ML-core datapath0 opcode2	5000	1331.25	6	480	0.44	1030.73	3	209	94.30	14
ML-core datapath0 (all opcodes)	5000	1834.68	8	1214	1.62	951.24	5	729	101.61	13
ML-core datapath2	2500	220.14	10	819	0.43	36.71	6	474	27.62	9
float32_fast_rsqrt	5000	1202.02	10	1055	1.79	144.91	8	797	118.47	14
video-core datapath	2500	26.86	12	1756	24.28	166.31	12	1732	316.62	11
internal datapath	2500	371.22	26	3095	13.73	60.42	25	2976	167.04	10
sha256	2500	232.66	112	85545	284.47	74.11	97	73990	3280.88	11
fpexp_32	5000	442.75	121	30569	240.90	236.97	114	29242	3441.08	13
Geo. Mean		686.74	6.93	569.86	0.84	418.16	4.85	407.19	34.46	
Ratio		100.0%	100.0%	100.0%	100.0%	**60.9%**	**70.0%**	**71.5%**	**4080.5%**	

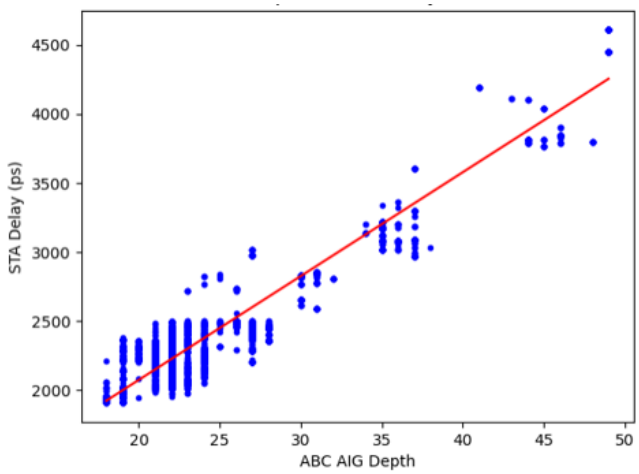


Fig. 8: Post-synthesis STA vs. ABC [7] AIG depth of 6912 different HLS design points.

for logic optimizations in tools like ABC [7]. As shown in Fig. 8, there is a compelling linear correlation between post-synthesis STA delay and AIG depth within ABC. This suggests a future research direction of bypassing the time-consuming technology mapping and post-synthesis STA, and directly using AIG depth as feedback.

4) Simultaneous HLS and logic optimization: In ISDC, we consciously bypass the back annotation technique used in [13], [15] due to its backend-specific nature and lack of generalizability. However, to squeeze out the extra bit of performance from digital circuits in the post-Dennard-scaling era, it is possible to blur the lines between HLS and downstream processes, such as logic synthesis. Future endeavors might see a co-optimization of the two design spaces, such as simultaneous HLS scheduling and logic optimization.

VI. Conclusion

In this paper, we proposed ISDC, a feedback-guided iterative scheduling algorithm for HLS. Building upon the traditional SDC approach, ISDC integrates iterative refinements and downstream tool feedback, showing a notable reduction in register usage. ISDC offers insights into feedback-guided optimization for HLS and highlights avenues for future exploration.

References

[1] J. Cong *et al.*, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proc. of DAC*, 2006.

[2] Advanced Micro Devices Inc, *Vitis High-Level Synthesis User Guide UG1399 (v2022.2)*, 2022.

[3] Microchip Technology Inc, *LegUp 2021.1 Documentation*, 2021.

[4] The XLS Authors, "XLS: Accelerated HW synthesis," https://github.com/google/xls, 2023, accessed on: 2023-09-17.

[5] C. Lattner *et al.*, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of CGO*, 2004.

[6] C. Wolf, "Yosys open synthesis suite," 2016.

[7] R. Brayton *et al.*, "ABC: An academic industrial-strength verification tool," in *Proc. of CAV*, 2010.

[8] S. Dai *et al.*, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *Proc. of FCCM*, 2018.

[9] The OpenSTA Authors, "OpenSTA: Parallax static timing analyzer," https://github.com/parallaxsw/OpenSTA, 2023, accessed on: 2023-09-17.

[10] The SkyWater PDK Authors, "SkyWater open source PDK," https://github.com/google/skywater-pdk, 2023, accessed on: 2023-09-17.

[11] D. Chen *et al.*, "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications," in *Proc. of CGO*, 2016.

[12] G. Lucas *et al.*, "FastYield: Variation-aware, layout-driven simultaneous binding and module selection for performance yield optimization," in *Proc. of ASPDAC*, 2009.

[13] H. Zheng *et al.*, "Fast and effective placement and routing directed high-level synthesis for FPGAs," in *Proc. of FPGA*, 2014.

[14] M. Tan *et al.*, "Mapping-aware constrained scheduling for LUT-based FPGAs," in *Proc. of FPGA*, 2015.

[15] C. Rizzi *et al.*, "An iterative method for mapping-aware frequency regulation in dataflow circuits," in *Proc. of DAC*, 2023.

[16] Z. Zhang *et al.*, "SDC-based modulo scheduling for pipeline synthesis," in *Proc. of ICCAD*, 2013.

[17] C. E. Leiserson *et al.*, "Retiming synchronous circuitry," *Algorithmica*, 1991.