Scheduling Information-Guided Efficient High-Level Synthesis Design Space Exploration

Xingyue Qian[1], Jian Shi[1], Li Shi[1], Haoyang Zhang[1], Lijian Bian[2], and Weikang Qian[1,3]
[1]University of Michigan-SJTU Joint Institute and [3]MoE Key Lab of AI, Shanghai Jiao Tong University, Shanghai, China
[2]Shanghai AnLogic Infotech Co., Ltd.
Emails: {qianxingyue, timeshi, shili2017, zhy-sjtu-jc}@sjtu.edu.cn, lijian.bian@anlogic.com, qianwk@sjtu.edu.cn

***Abstract*—High-level synthesis (HLS) transforms designs specified by high-level programming language into RTL designs. In order to get the optimal designs, many design space exploration (DSE) methods are proposed. However, most of them consider the HLS tool as a black box, ignoring crucial information from the synthesis process, particularly the scheduling step. In this work, we propose to extract some useful information from scheduling to guide the DSE and develop a genetic algorithm (GA)-based DSE method based on our in-house HLS tool. The experimental results show that our method can obtain more Pareto-optimal points than the counterpart without using the scheduling information. It also outperforms a traditional GA-based HLS DSE method by using only a quarter of the total run time. For a large benchmark, our method finds 95.7% Pareto-optimal designs by visiting only 0.18% total promising design points.**

I. Introduction

Field-programmable gate array (FPGA) has shown its remarkable ability as accelerator for various applications such as deep learning and image processing. High-level synthesis (HLS) methodology is proposed to automatically transform a high-level source code into an register-transfer level (RTL) design that is used to program FPGA.

Nowadays, several HLS tools such as Vivado HLS [1] and LegUp [2] are available. They have many synthesis options that can be controlled by users. These options are also called *knobs*. We call a combination of knobs a *design point*, and all the possible points form a *design space*. The design quality is usually measured by multiple conflicting metrics, such as area and latency. For such a multi-objective optimization problem, we typically look for a set of design points that are *Pareto-optimal*, which is achieved by design space exploration (DSE).

Various HLS DSE methods have been proposed in recent years. For example, simulated annealing is used to generate new designs incrementally from old ones based on a global cost function [3]. A genetic algorithm (GA) is applied to do DSE, which also uses machine learning (ML) to speed up the process [4]. In another set of works, *e.g.*, [5], analytical models are proposed to replace the actual HLS process, and the design space is explored heuristically. However, most methods have no access to the internal information in HLS process that can guide the DSE; they either use an HLS tool as a black-box or an analytical model that needs to be carefully tuned to obtain the quality of result to guide the DSE.

There are few works where the HLS tool is more integrated with DSE. In [6], design space of a single loop is explored within an HLS process by optimizing resource usage for each candidate initiation interval. Another work explores the design space of multi-clock dataflow designs based on the bottleneck model reported by their own HLS tool [7].

However, the potential of exploiting the information that can be easily obtained from the scheduling step to efficiently guide the DSE has not been well studied yet.

In this work, we propose a novel and efficient DSE method that exploits the scheduling information. It is adapted from the traditional GA and leverages our own HLS tool. The main contributions of our work are listed as follows.

- We identify some useful information from scheduling that can lead to a more efficient DSE. Such information can give hints on the potential benefit of changing a knob, so it can guide the DSE to search more promising points.
- We propose efficient methods to extract the above information from the scheduling step. The extraction is done during the synthesis process with almost no extra effort.
- We propose a novel DSE method adapted from GA, which exploits the extracted scheduling information.

The experimental results show that our method is far better than the counterpart without using the scheduling information. Compared to a traditional GA-based method, our method achieves $4\times$ speed-up, while finding better Pareto set. For a large benchmark, our method can find 95.7% Pareto-optimal points by searching 0.18% of the promising design space.

II. Background

A. High-Level Synthesis

The input of a typical HLS flow is an intermediate representation (IR) compiled from a high-level source code and the output is the corresponding RTL code. Four main steps of HLS are allocation, scheduling, binding, and RTL generation [2]. The step most relevant to our work is scheduling. It assigns each IR instruction to specific state(s), *i.e.*, clock cycle(s).

To support the IR instructions, some types of functional units (FUs), *e.g.*, 64-bit signed multiplier, are available. The FU types can be *resource-constrained* or *resource-unconstrained*. The number of available hardware instances for the former is specified and limited, while that for the latter is not. In this work, three kinds of FU types, *i.e.*, multipliers, dividers, and remainders, are considered to be resource-constrained.

B. Area and Latency Estimation

In this work, we measure the quality of a hardware design by two metrics: area and latency.

This work is supported by the National Key R&D Program of China under grant number 2021ZD0114701. Corresponding author: Weikang Qian.

For area estimation, The total area is calculated as a weighted sum given as $\mathit{Area} = \lambda_1 N_{\mathit{LUT}} + \lambda_2 N_{\mathit{FF}} + \lambda_3 N_{\mathit{DSP}} + \lambda_4 N_{\mathit{BRAM}}$, where λ_i's are hardware-dependent coefficients and N_{LUT}, N_{FF}, N_{DSP}, and N_{BRAM} are the total numbers of look-up tables, flip-flops, digital signal processing modules, and block random-access memories, respectively [8].

The total latency is the product of the clock period and the number of clock cycles spent in running the design. In this work, we assume that the clock period is a user-given input. Thus, the latency is just measured by clock cycles obtained from a one-time IR simulation and the state tables generated for each design point. Note that the correctness of our latency estimation is verified against the RTL simulation result.

C. Pareto-Optimal Design Point

A design point x in the whole design space S is said to be *Pareto-optimal* if there does not exist another point y in S with both less area and less latency than x. The set of Pareto-optimal points is also called the *Pareto-optimal set*, or simply *Pareto set*. The target of the DSE is to find the Pareto set. A concept related to Pareto-optimality is *rank*, which is widely used in GA. Rank-1 points are the Pareto-optimal points. For $i \geq 2$, rank-i points are the Pareto-optimal points in the space with points in the first $(i-1)$ ranks removed.

III. Methodology

In this work, we consider the following problem: *find as many Pareto-optimal design points as possible in the area-latency plane by visiting as few points as possible.* For simplicity, the types of knobs are limited to two in our methodology elaboration. They are *resource number* and *loop pipelining*, where resource number specifies the number of available hardware instances for each resource-constrained FU type and loop pipelining is applied to the inner-most loops. However, the proposed DSE method can be generalized to include other types of knobs such as loop unrolling and array partitioning. We also consider loop unrolling in experiments.

A. Information from Scheduling for Guiding DSE

In HLS, the step most relevant to DSE is scheduling. Several popular scheduling methods exist, including list scheduling [9] and SDC scheduling [10]. Our HLS tool uses the former.

Based on the list scheduling procedure and the pipelining procedure, we propose to extract two pieces of information from the scheduling step to guide the DSE: conflict number and pipeline gain. They correspond to the two types of available knobs, resource number and loop pipelining, respectively. They are extracted by our HLS tool during the synthesis of a design point x and give hints on the benefit of changing a knob of x to reach a new design point.

1) Conflict Number: When an instruction is ready (*i.e.*, all of its predecessors have been finished) but cannot be scheduled due to resource limit of the corresponding FU type T, a *conflict* occurs for the FU type T. In this case, if the resource number of T is increased, then *the conflict is resolved* and the instruction can be scheduled, leading to a potential decrease of the total latency. The *conflict number* of an FU type T is the total number of conflicts occurred on T during the scheduling, calculated as follows:

$$\mathit{Conflict}(T) = \sum_{b \in B} \mathit{TotExTimes}(b) \cdot \mathit{Conflict}_b(T), \quad (1)$$

where B is the set of all basic blocks, $\mathit{TotExTimes}(b)$ is the total number of times a basic block $b \in B$ is executed, and $\mathit{Conflict}_b(T)$ is the number of conflicts occurred on T during the scheduling of basic block b. Generally speaking, the more conflicts an FU type has, the more critical it is to increase its resource number, in order to reduce the latency. Thus, in DSE, if we want to reach a new design point by increasing the resource number of an FU type of an already-visited design point x, we favor the FU type with more conflicts.

2) Pipeline Gain: If a loop has not been pipelined, the *pipeline gain* of the loop is the expected reduction of the total number of clock cycles when the loop is pipelined; otherwise, it is 0. In DSE, if we want to reach a new design point by pipelining a loop of an already-visited design point x, we favor the loop with a larger pipeline gain. Consider a basic block b that is a loop. We denote the number of times the entire loop b is executed as $\mathit{ExTimes}(b)$, the number of iterations of the i-th ($1 \leq i \leq \mathit{ExTimes}(b)$) time the loop is executed as $\mathit{Iters}_i(b)$, the number of clock cycles spent on one iteration of b as $\mathit{Cycles}(b)$, and the minimal initiation interval [11] as $\mathit{II}(b)$. By definition, the pipeline gain can be calculated as

$$\mathit{Gain}(b) = \sum_{i=1}^{\mathit{ExTimes}(b)} (\mathit{Cycles}(b) - \mathit{II}(b)) \cdot (\mathit{Iters}_i(b) - 1), \quad (2)$$

where $\mathit{ExTimes}(b)$ and $\mathit{Iters}_i(b)$ are obtained from a one-time IR simulation. $\mathit{Cycles}(b)$ is obtained just as the length of state table without pipelining. The only extra effort to obtain the gain is to estimate $\mathit{II}(b)$ under the resource constraint, which can be obtained efficiently by calling part of the pipelining procedure during the synthesis of design point x.

B. Proposed DSE Flow

In this section, we describe the proposed DSE flow using the extracted information from scheduling. We first describe a function $\mathit{HLS}(x)$ used in the flow. Then, we present the entire flow. Unless otherwise specified, FU type refers to resource-constrained FU type and loop refers to inner-most loop.

1) $\mathit{HLS}(x)$: This function calls our HLS tool to synthesize a design point x with its knob setting $x.\mathit{num}$ storing the resource numbers of the FU types and $x.\mathit{ppl}$ storing the pipelining options of the loops. Our tool evaluates the quality of x and gives $x.\mathit{area}$ and $x.\mathit{latency}$. As a byproduct, our tool also outputs the scheduling information extracted during the process and stores the conflict numbers of FU types and the pipeline gains of loops into vectors $x.\mathit{conflict}$ and $x.\mathit{gain}$.

Example 1 *Table I shows a design point x of an example design. By the table, Rem32 has no conflict, so its resource number will not be increased. Mul32 has more conflicts than Div32, so increasing the resource number of Mul32 is preferred over increasing that of Div32. Loop2 has already been pipelined, so its pipeline gain is 0 by definition. For*

the unpipelined loops, Loop3 has a larger pipeline gain than Loop1, so pipelining Loop3 is preferred over pipelining Loop1.

Table I. The conflict numbers and the pipeline gains extracted from the scheduling step of an example design under a knob setting.

FU type	Mul32	Div32	Rem32
$x.num$	2	3	1
$x.conflict$	5	1	0
Loop	Loop1	Loop2	Loop3
$x.ppl$	False	True	False
$x.gain$	32	0	76

Algorithm 1: The proposed DSE flow.

Input: High-level source code; hardware configuration file; N_{gen}: the number of generations; N_{mut}: the number of mutations per generation; R: the number of ranks in a population;
Output: Pareto set $\mathcal{X}$;

```
1  Initial point x: x.num ← (1,...,1); x.ppl ← (false,...,false);
2  x ← HLS(x); Pareto set X ← {x};
3  Set of all visited points S ← {x}; Population X_rank ← ∅;
4  count_FU ← 0; count_loop ← 0;
5  for g ← 1 to N_gen do
6      Pick design points in the first R ranks of S to form X_rank;
7      for m ← 1 to N_mut do
8          Randomly pick a point parent from X_rank;
9          new_point ← parent;
10         if rand(0,1) < BiasProb(count_FU, count_loop) then
11             selFU ← true;
12             i ← RandPick(parent.conflict);
13             new_point.num_i ← new_point.num_i + 1;
14         else
15             selFU ← false;
16             i ← RandPick(parent.gain);
17             new_point.ppl_i ← true;
18         if new_point ∈ S then continue;
19         new_point ← HLS(new_point); S ← S ∪ {new_point};
20         Update X by considering new_point;
21         if new_point ∈ X then
22             if selFU = true then count_FU ← count_FU + 1;
23             else count_loop ← count_loop + 1;
24 return X;
```

2) Details of the DSE Flow: Our proposed DSE flow is adapted from the traditional GA. Algorithm 1 shows its details. Lines 1–4 are the initialization steps. Line 1 generates an initial design point with a single instance for all the FU types and no pipelining for all the loops. After the HLS procedure is applied, the point is added into the Pareto set $\mathcal{X}$ and the set of all the visited points, $\mathcal{S}$ (Lines 2–3). Line 4 sets two variables $count_{FU}$ and $count_{loop}$ to zero. Their meanings will be introduced later.

The main flow iterates for N_{gen} generations (Line 5). In each generation, Line 6 first picks the first R ranks of the set of all the visited points, $\mathcal{S}$, to form the present population $\mathcal{X}_{rank}$. Then, N_{mut} mutations (Lines 7–23) are tried.

In each round of the mutation loop, Line 8 first randomly picks a parent point from the present population $\mathcal{X}_{rank}$. For traditional GA, the mutation on a point x is performed by randomly changing the setting of one knob. In our case, we bias the mutation with the scheduling information. Our mutation creates a new point by either increasing the resource number of a selected FU type or pipelining a selected loop of x. The two types of mutations are chosen randomly. However, the probability of selecting which type of mutation to be applied is biased towards the one that has produced more Pareto-optimal points. The probability is decided by the function $BiasProb(a, b)$ shown at Line 10, defined as $BiasProb(a, b) = \frac{base^a}{base^a + base^b}$, where $base \geq 1$ is a parameter. Its two inputs are $count_{FU}$ and $count_{loop}$, which record the numbers of Pareto-optimal points generated by the two types of mutations, respectively, and are updated in Lines 21–23. Thus, the more Pareto-optimal points are generated by a type of mutation, the more likely it will be picked again.

After the mutation type is chosen, the FU type or the loop corresponding to the mutation type is picked according to the conflict numbers or the pipeline gains using the function *RandPick* (Lines 12 and 16). The function takes a vector v as input and randomly picks an index i in the vector with probability $\frac{v_i}{\sum_j v_j}$, where v_i is the i-th entry in v.

After a new point is created by mutation, Line 18 checks whether it has already been visited. If not, Line 19 applies the HLS procedure to it and adds it into the visited point set $\mathcal{S}$. Line 20 updates the Pareto set $\mathcal{X}$ by considering the new point. Finally, if the new point is added into the Pareto set, either $count_{FU}$ or $count_{loop}$ is incremented by 1, depending on the selected mutation type (Lines 21–23).

Note that in traditional GA, there is also a crossover step, but it is not applicable here since in crossover step, multiple knobs are changed at the same time while our scheduling information specializes in detecting the benefit of changing a single knob.

IV. Experimental Results

Our HLS tool is developed based on LLVM 10.0, using Clang in the front end. Its generated RTLs can be successfully synthesized and pass the on-board FPGA test, so its correctness is verified.

To study the performance of our proposed DSE method, a baseline brute-force method that explores all the *promising* points in the design space is also implemented. This means that the design points that cannot be Pareto-optimal are pruned and the brute-force method only visits the remaining points.

We use 7 benchmarks selected from CHStone [12], Polybench [13], and LegUp [2] to test our DSE method. Their basic information is listed in Table II, where the row 'size' lists the size of the promising design space when two types of knobs, resource number and loop pipelining, are considered, and the row 'size_ur' lists the size after including unrolling knob. Only these benchmarks are selected, since 1) they contain both resource-constrained FU types and loops that can be pipelined, and 2) their promising design spaces are large enough to test the efficiency of our proposed DSE method.

Table II. Information of the benchmarks.

benchmark	mandelbrot	gemver	aes	adi	adpcm	gsm	jpeg
#FU types	1	3	5	3	1	1	3
#loops	3	5	10	8	12	11	15
size	96	256	1024	3072	20480	73728	4.2E6
size_ur	384	6144	32768	7.9E5	2.5E7	8.4E6	4.3E9

To measure the performance of our method, two metrics are used: dominance percentage, which denotes the percentage of real Pareto-optimal points found (D%), and percentage of visited points in the promising design space (Pt%) [14].

A. Performance with Two Types of Knobs

This section studies the performance of our method when two types of knobs, resource number and loop pipelining, are considered. To demonstrate the benefit of using scheduling information, we implement a corresponding black-box counterpart of our method where no conflict number or pipeline gain is available, so the FU type or loop choice is done randomly. Fig. 1 shows the performance of three versions of our method with different choices of the parameters *base* and R in Algorithm 1 and their corresponding black-box counterparts (*i.e.*, BK_*), using the D%-Pt% plot. Note that different points on the same curve have different choices for the parameters N_{gen} and N_{mut} but the same choice of *base* and R. The values D% and Pt% of each point is the average value over all benchmarks except `jpeg` in 100 runs, where `jpeg` is excluded since its promising design space is too large for the brute-force algorithm. From Fig. 1, we can see that when visiting the same number of points, our method can obtain Pareto sets with larger dominance than the black-box method.

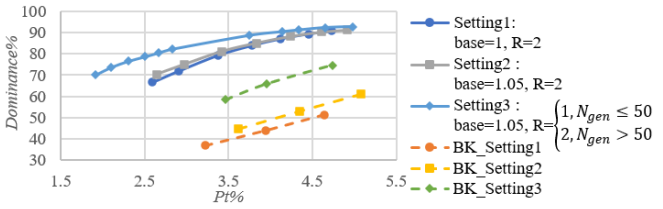

Fig. 1. Comparison of our proposed DSE method and the black-box DSE method, a counterpart of ours without using the scheduling information.

For a large benchmark `gsm` from CHStone benchmark suite, our method can find 95.7% Pareto-optimal designs by visiting only 0.18% total promising design points.

B. Generalization to Loop Unrolling Knob

To show the generalization ability of our proposed DSE method, we extend our method by also including the loop unrolling knob. The related scheduling information is the expected number of cycles that can be reduced by unrolling the loop divided by a rough estimation of the area increase.

The design space becomes too large for the brute-force method for some benchmarks, so we study the performance of our method by comparing it to a traditional GA-based method denoted as *TGA*, which is used as the baseline in [4]. To make the comparison more comprehensive, we add two more criteria widely used in evaluating the DSE quality: *average distance from reference set (ADRS)* and *cardinality* [14]. ADRS indicates how close the obtained Pareto set is to the reference set. Cardinality is just the size of the obtained Pareto set. A better DSE method usually has a smaller ADRS and a larger cardinality.

We compare our method and TGA on all the 7 benchmarks. The parameters *base* and R are selected as Setting3 in Fig. 1. The parameters N_{gen} and N_{mut} are tuned so that the number of visited points of our method is roughly $1/4$ of that of TGA. The reference Pareto set is obtained from the combination of the Pareto sets found by the two methods. Due to randomness, the average of each measure over the 5 runs is obtained for each method together with the average run time.

Table III. Performance comparison between our method and a traditional GA-based HLS DSE method [4] after including the loop unrolling knob.

benchmark	Dominance%		ADRS%		Cardinality		Run time (s)	
	TGA	Our	TGA	Our	TGA	Our	TGA	Our
`mandelbrot`	47.0	71.2	9.22	3.26	13	19	315	92
`gemver`	76.9	81.8	1.39	0.83	23	24	1113	260
`aes`	71.9	78.8	0.24	0.06	29	29	1816	397
`adi`	49.0	63.3	1.49	0.96	38	40	1553	485
`adpcm`	8.90	96.1	1.30	0.01	29	34	1578	341
`gsm`	21.3	80.8	0.61	0.08	37	55	1629	346
`jpeg`	35.5	64.5	2.18	1.86	24	33	1482	365
Geomean	36.5	75.9	1.34	0.32	26	32	1208	296

Table III compares the performance of our DSE method and TGA. It can be seen that the quality of the Pareto set obtained by our method is far better than that obtained by TGA under all three measures. Furthermore, our method is $4.1\times$ faster. In [4], an advanced ML-accelerated DSE method was also proposed. It was reported that compared to TGA, the ML-accelerated method only enjoys $2\times$ acceleration while having some DSE quality drop. Thus, our method is also better than the ML-accelerated method.

V. Conclusion

In this work, we propose to extract useful information from the scheduling step in HLS process. Then, an efficient DSE method guided by the scheduling information is implemented. The experimental results show that the proposed DSE method outperforms its counterpart without using the scheduling information and a traditional GA-based DSE method under various criteria. In future work, we plan to support more design knobs by exploiting other useful information from scheduling.

References

[1] *Introduction to FPGA Design with Vivado High-Level Synthesis*, Xilinx Inc., 2019.

[2] A. Canis *et al.*, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," in *FPGA*, 2011, pp. 33–36.

[3] B. Schafer *et al.*, "Adaptive simulated annealer for high level synthesis design space exploration," in *VLSI-DAT*, 2009, pp. 106–109.

[4] B. Schafer and K. Wakabayashi, "Machine learning predictive modelling high-level synthesis design space exploration," *IET Computers & Digital Techniques*, vol. 6, pp. 153–159, 2012.

[5] J. Zhao *et al.*, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," in *ICCAD*, 2017, pp. 430–437.

[6] J. Oppermann *et al.*, "Design-space exploration with multi-objective resource-aware modulo scheduling," in *Euro-Par*, 2019, pp. 170–183.

[7] T. Liang *et al.*, "Hi-ClockFlow: Multi-clock dataflow automation and throughput optimization in high-level synthesis," in *ICCAD*, 2019, pp. 1–6.

[8] N. Pham *et al.*, "Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis," in *DATE*, 2015, pp. 157–162.

[9] J. Zhu and D. Nikil, "Electronic system-level design and high-level synthesis," in *Electronic Design Automation*. Morgan Kaufmann, 2009, ch. 5, pp. 235–297.

[10] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *DAC*, 2006, pp. 433–438.

[11] M. Fingeroff, *High-level synthesis: Blue book*. Xlibris Corporation, 2010.

[12] Y. Hara *et al.*, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *ISCAS*, 2008, pp. 1192–1195.

[13] L. Pouchet *et al.*, "PolyBench/C the polyhedral benchmark suite," 2015. [Online]. Available: https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/

[14] B. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE TCAD*, vol. 39, no. 10, pp. 2628–2639, 2020.