Design Space Exploration of Multiple Loops on FPGAs using High Level Synthesis

Guanwen Zhong*, Vanchinathan Venkataramani*,Yun Liang†, Tulika Mitra* and Smail Niar‡
*School of Computing, National University of Singapore
†Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China
‡LAMIH, University of Valenciennes, France
Email: {guanwen,vvanchi,tulika}@comp.nus.edu.sg, ericlyun@pku.edu.cn, smail.niar@univ-valenciennes.fr

***Abstract*—Real-world applications such as image processing, signal processing, and others often contain a sequence of computation intensive kernels, each represented in the form of a nested loop. High-level synthesis (HLS) enables efficient hardware implementation of these loops using high-level programming languages. HLS tools also allow the designers to evaluate design choices with different trade-offs through pragmas/directives. Prior design space exploration techniques for HLS primarily focus on either single nested loop or multiple loops without consideration to the data dependencies among them. In this paper, we propose efficient design space exploration techniques for applications that consist of multiple nested loops with or without data dependencies. In particular, we develop an algorithm to derive the Pareto-optimal curve (performance versus area) of the application when mapped onto FPGAs using HLS. Our algorithm is efficient as it effectively prunes the dominated points in the design space. We also develop accurate performance and area models to assist the design space exploration process. Experiments on various scientific kernels and real-world applications demonstrate that our design space exploration technique is accurate and efficient.**

I. Introduction

Current- and next-generation applications in many embedded system domains demand high performance that cannot be satisfied with general-purpose processors. The ASICs can provide the best performance at lowest power budget; but they suffer from huge design efforts and lack of flexibility. Compared to ASICs, Field Programmable Gate Array (FPGA) devices have the advantages of re-programbility and much lower cost. In addition, high spatial parallelism and substantially increased capacity make FPGAs amenable to tailoring according to individual applications. Thus FPGAs have become attractive to the designers and have gained market traction for the past two decades. However, the complex hardware programming models such as low-level hardware description languages (Verilog/VHDL) and synthesis flow make FPGAs inaccessible to average developers, which hinders its acceptability. This productivity gap has led to the emergence of high-level synthesis (HLS) that allows designers to focus on high-level specifications such as C/C++, SystemC, Matlab etc. and automatically transforms such high-level specifications into low-level implementations in the form of Register-transfer level (RTL) circuits or gate-level netlists.

After decades of sustained endeavour, both academic [7][8][9][24] and industrial [1][4][23][25][26] tools have emerged as mature solutions for high-level synthesis. Not only these tools can generate hardware implementation from high-level programming language specifications, but they also give the designers multiple implementation choices (e.g., loop unrolling factors) in the form of pragmas/directives. This allows the designers to perform in-depth design-space exploration (DSE) that evaluates numerous hardware implementations through HLS tools and returns a set of Pareto-optimal points in the multi-objective design space optimizing latency, power, throughput and area. However, the huge complexity of the design space coupled with the non-negligible runtime of HLS tools renders it impossible to perform exhaustive DSE for relatively complex applications. This makes the dream of a push-button solution for design space exploration of entire application un-achievable at this point.

To address this challenge, we propose an efficient DSE technique to obtain the Pareto-optimal curve (performance vs. area) for an application mapped onto FPGAs using HLS. Our DSE algorithm prunes the design space by effectively eliminating dominated configurations instead of evaluating all the possible configurations. This pruning can significantly reduce the number of invocations of the HLS tool. We also develop accurate performance and area prediction models for our DSE algorithm, further scaling down the need to invoke HLS for the remaining design points.

More concretely, as real-world applications often contain a sequence of computation intensive kernels represented by nested loops, we target automated design-space exploration to map multiple nested loops onto FPGAs using HLS. Existing works [3][5][17][20] that target a single nested loop or multiple loops often ignore interactions and data dependencies among the loops. In this paper, we consider the dataflow dependencies among the loops as our experimental evaluation reveals that such interactions among multiple loops can not be neglected. Experimental evaluation with real-world benchmarks show that the set of Pareto-optimal points predicted by our solution is very close to the Pareto-optimal points identified through exhaustive search, whereas the runtime of our DSE algorithm is 235X faster, on an average, compared to exhaustive search.

II. Background and Motivation

Real-world applications in image processing, signal processing, etc., often contain multiple (single or nested) loops.

978-1-4799-6492-5/14/$31.00 ©2014 IEEE

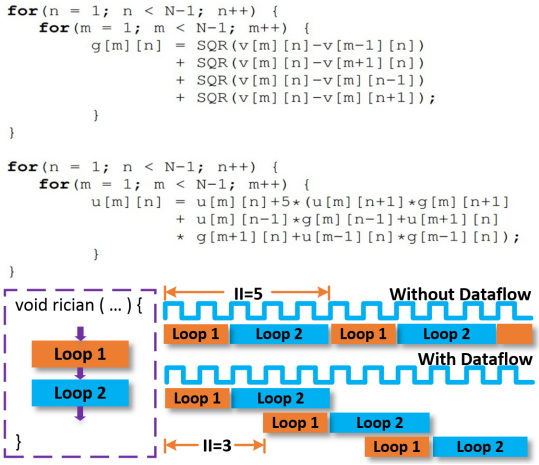


Fig. 1: Rician Deconvolution with dataflow feature.

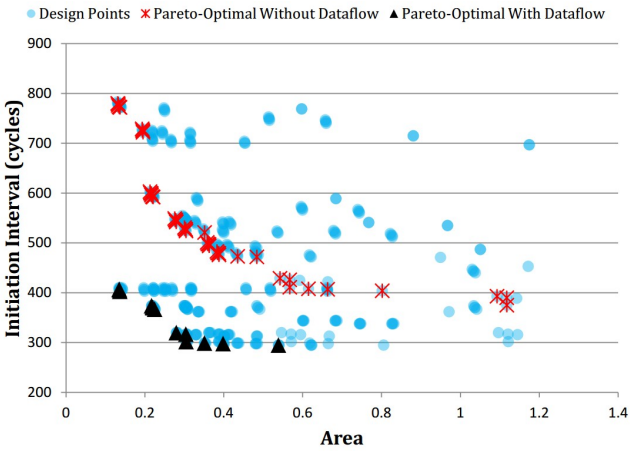


Fig. 2: Design space of Rician Deconvolution example.

Using HLS tools, we can convert these loops described in high-level languages such as C/C++ or SystemC into efficient FPGA-based hardware implementations. Moreover, modern HLS tools such as Xilinx Vivado [26] feature with a variety of pragmas/directives such as loop unrolling, pipelining, etc., for loop performance optimizations. By using the pragmas differently, we can have multiple different implementations with performance and area tradeoff.

The loops contained in an application are often related via dataflow dependencies. Fig. 1 illustrates the dataflow feature using the *Rician Deconvolution* [19] application in medical imaging domain. There are two loops in the application. The two loops are dependent as the first loop produces the array g, which is consumed by the second loop. Dataflow optimization aims to minimize the *initiation interval* (II) of the top-level function (e.g., *rician* function) containing the loops, where the II is defined as the number of cycles between consecutive initiations of the function [11][15][26]. In this example, without dataflow optimization, each loop executes in isolation and the II of the function is the sum of the execution time of the two loops. With dataflow optimization, loop instances from different iterations of the function can execute in a pipelined fashion thereby reducing the II. As throughput is the inverse of II [26], minimizing II leads to improved throughput.

A. Motivating Example

For each individual loop in Figure 1, we optimize it through loop unrolling. For each loop, we set the loop bound to be 6 and vary the unroll factor ({1, 2, 3, 6}). We also consider to enable or disable the dataflow between the two loops. Thus, there are 512 (2*4*4*4*4) design points in total. Fig. 2 presents the *initiation interval* versus *area* design space using Vivado HLS. We define the *area* in terms of Weighted Area (*WA*) and Area Efficiency (*AE*). *WA* and *AE* are defined using the following equations:

$$WA = bram/BRAM + dsp/DSP + ff/FF + lut/LUT \quad (1a)$$

$$AE = MAX(bram/BRAM, dsp/DSP, ff/FF, lut/LUT) \quad (1b)$$

where *BRAM*, *DSP*, *FF* and *LUT* represent the available BRAM, DSP, Flip-Flop, and LUT resources of a given FPGA platform, while *bram*, *dsp*, *ff* and *lut* are FPGA resources utilized by the current pragma configuration. We define *area* as,

$$area = \begin{cases} WA & \text{if } AE \leq 1, \\ \alpha & \text{if } AE > 1. \end{cases} \quad (2)$$

That is, when *AE* is less than or equal to 1, *area* is the sum of the resource usage utilization ratio of different resource types; otherwise, the design has already exceeded the FPGA resource budget, and we set *area* to α. We use 4 for α in this paper.

In Fig. 2, the red stars represent the Pareto-optimal curve without dataflow directive, while the black triangles represent the Pareto-optimal curve with dataflow directive. The results clearly demonstrate that data flow optimization plays a critical role. In terms of II, the best design point on the Pareto-optimal curve with dataflow is 30% better than that without dataflow. However, the current DSE techniques using HLS tools [2][3][17][20][27] primarily focus on optimizing individual loops and ignore the dataflow feature between loops.

Table I presents the detailed II results for a subspace of the entire design space. *Dataflow = 1* means that we enable dataflow and vice versa. The unrolling factor "2_1_1_1" represents the unroll factor for each loop, where the first 2 and 1 are unroll factors for the outer and inter loops of loop 1. For this subspace, we do not optimize loop 2. Thus, the unroll factor for its outer and inner loop are 1. From Table I, without dataflow optimization, II is equal to the sum of execution time of loops in the application. When dataflow feature is enabled, II is equal to the execution time of the longest loop. This is because the two consecutive instances of the same loop can not be executed concurrently. Thus, to minimize the II for an application with data dependent loops, we have to optimize the longest loop. The DSE in this case is more challenging as a different loop can become the longest loop after the original longest loop is optimized. In Section III, we develop our efficient DSE algorithm to overcome this challenge.

Configuration		Loop1	Loop2	Initiation Interval
Dataflow	Unrolling Factor			
0	2_1_1_1	405	372	780
1		405	372	407
0	2_2_1_1	225	372	600
1		225	372	374
0	2_6_1_1	72	372	447
1		72	372	374

TABLE I: Latency results of loop1 for the motivating example.

Vivado HLS takes from seconds to minutes to synthesize each configuration. The total design space exploration time for this simple example is about 90 minutes. However, when the complexity of the applications continues to grow, exhaustive design space exploration becomes infeasible. Thus, we urgently need a design space exploration technique to efficiently and accurately obtain the pareto-optimal curve.

B. Problem Formulation

Let us consider an application kernel $\mathcal{K}$ that consists of n loops $\{L_1, L_2, ..., L_n\}$. The kernel configuration $\mathcal{C}_\mathcal{K}$ is a set of configurations $\{KC^1, KC^2, ..., KC^t, ..., KC^{N_\mathcal{K}}\}$, where $N_\mathcal{K}$ is the total number of kernel configurations. For a kernel configuration KC^t, it is represented by $\langle lc_1, lc_2, ..., lc_i, ..., lc_n\rangle$, where lc_i is a configuration of loop L_i.

Loop L_i contains m_i levels of nested loops $\langle L_{i1}, L_{i2}, ..., L_{ij}, ..., L_{im_i}\rangle$. The execution time of a loop L_i is represented by E_i. In this work, for each loop, we consider loop unrolling for performance optimization. We do not consider loop pipelining. Thus, the initiation interval II_i of a loop L_i is E_i. A_i is the area of L_i. The configuration set C_i of L_i is defined by $\{c_i^1, c_i^2, ..., c_i^r, ..., c_i^{S_i}\}$, where S_i is the total number of configurations of loop L_i. Each configuration c_i^r of loop L_i is represented by $\langle u_{i1}, u_{i2}, ..., u_{ij}, ..., u_{im_i}\rangle$, where u_{ij} is the loop unrolling factor configuration for the j^{th}-level loop L_{ij}. UF_{ij} consists of all loop unrolling configurations $\{uf_{ij}^1, uf_{ij}^2, ..., uf_{ij}^{s_{ij}}\}$ for L_{ij} and $u_{ij} \in UF_{ij}$. s_{ij} specifies the number of configurations for j^{th}-level loop L_{ij} in loop L_i. Then S_i, the number of configurations of a loop L_i, is defined as

$$S_i = \prod_{j=1}^{m_i} s_{ij} \tag{3}$$

The total configurations for a kernel $\mathcal{K}$ is calculated by,

$$N_\mathcal{K} = \prod_{i=1}^{n} S_i \tag{4}$$

Finally, the **design space** is doubled if we consider dataflow feature.

$$2 \cdot N_\mathcal{K} = 2\prod_{i=1}^{n}\Big(\prod_{j=1}^{m_i} s_{ij}\Big) \tag{5}$$

The area $\mathcal{A}_\mathcal{K}$ of $\mathcal{K}$ is calculated as

$$\mathcal{A}_\mathcal{K} = \sum_{i=1}^{n} A_i \tag{6}$$

The initiation interval $II_\mathcal{K}$ of $\mathcal{K}$ is defined below,

$$II_\mathcal{K} = \begin{cases} \sum_{i=1}^{n} E_i & \text{if } \mathcal{K} \text{ has no dataflow feature,} \quad (7a) \\ \max\limits_{i=1,...,n} E_i & \text{if } \mathcal{K} \text{ has dataflow feature} \quad (7b) \end{cases}$$

Our goal is to derive the Pareto-optimal curve with initiation interval and area trade-off.

Pareto-optimal Curve: Let $\mathcal{D}$ be the design space of a kernel $\mathcal{K}$ consisting of all design points. Let $(KC^t, II_\mathcal{K}^t, \mathcal{A}_\mathcal{K}^t)$ denote the corresponding initiation interval $II_\mathcal{K}^t$ of a kernel $\mathcal{K}$ and area $\mathcal{A}_\mathcal{K}^t$ under the kernel configuration KC^t. We are interested in identifying a curve that consists of all possible Pareto-optimal solutions $\mathcal{P} = \{(KC^1, II_\mathcal{K}^1, \mathcal{A}_\mathcal{K}^1), ..., (KC^q, II_\mathcal{K}^q, \mathcal{A}_\mathcal{K}^q), ..., (KC^Q, II_\mathcal{K}^Q, \mathcal{A}_\mathcal{K}^Q)\}$. Q denotes the number of Pareto-optimal solutions for a kernel $\mathcal{K}$. The Pareto-optimal curve captures the different performance-area tradeoffs [6][16]. Each $(KC^q, II_\mathcal{K}^q, \mathcal{A}_\mathcal{K}^q)$ in this curve has the property that there does not exist any configuration with a triple $(KC^t, II_\mathcal{K}^t, \mathcal{A}_\mathcal{K}^t)$ such that $II_\mathcal{K}^t \leq II_\mathcal{K}^q$ and $\mathcal{A}_\mathcal{K}^t \leq \mathcal{A}_\mathcal{K}^q$, with at least one of the inequalities being strict. Thus, for any $(KC^t, II_\mathcal{K}^t, \mathcal{A}_\mathcal{K}^t) \in \mathcal{D}-\mathcal{P}$, there exists a $(KC_\mathcal{K}^q, II_\mathcal{K}^q, \mathcal{A}_\mathcal{K}^q) \in \mathcal{P}$ such that $II_\mathcal{K}^q \leq II_\mathcal{K}^t$ and $\mathcal{A}_\mathcal{K}^q \leq \mathcal{A}_\mathcal{K}^t$, with at least one of the inequalities being strict. The set $\mathcal{D} - \mathcal{P}$ consists of all the *dominated solutions*, dominated by elements in the Pareto-optimal set $\mathcal{P}$.

III. Automated Design Space Exploration

The main goal of our automated DSE problem is to efficiently explore the design space and provide an approximate Pareto-optimal curve with performance (II) and area tradeoff. The exhaustive search that evaluates each configuration using HLS tools and then builds the pareto-optimal curve is infeasible for large applications. In contrast, we improve the exploration time through performance estimation and efficient search algorithms.

A. Framework Overview

Our automated DSE framework is shown in Fig. 3. The input to our DSE is the synthesizable C code of an application. The DSE framework consists of *Code Detection*, *Area/Performance Prediction*, *Dataflow Detection*, *Search Algorithm*.

The *Code Detection* component detects the loops, number of loop levels in a nested loop, and loop bound information. *Area/Performance Prediction* component predicts the area and initiation interval of different configurations. *Dataflow Detection* component checks whether the dataflow can be enabled or not. With dataflow enabled, we use *Search With Dataflow* algorithm to estimate the Pareto-optimal curve; othewise, it will use *Search Without Dataflow* algorithm. The output of the search algorithm is the pareto-optimal curve.

B. Area and Performance Prediction Models

In order to reduce the number of invocations of HLS tools, we need to perform DSE based on area and performance estimates. To develop accurate area/performance models, we

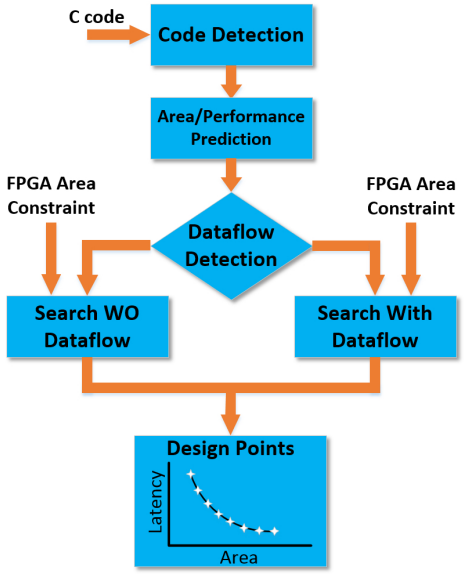


Fig. 3: The Automated Design Space Exploration Framework

pre-invoke HLS tools for some sample design points and then perform estimation for the remaining points.

1) Area Prediction Model: Through empirical study, we observe that when the configuration u_{ij} of all the nested loop L_{ij} except for the innermost level in a given nested (m_i levels) loop L_i is fixed, increasing unrolling factors u_{i1} for the innermost loop L_{i1} incurs linear increase in area A_i of loop L_i. Let the configuration with no optimization on the innermost-level loop L_{i1} of loop L_i be $c_i^1 = \langle 1, u_{i2}, ..., u_{im_i} \rangle$ and the configuration with the innermost-level loop completely unrolled in loop L_i be $c_i^{s_{i1}} = \langle uf_{i1}^{s_{i1}}, u_{i2}, ..., u_{im_i} \rangle$. Based on this observation, we can predict area $A_i(c_i^r)$ as follows,

$$A_i(c_i^r) = \frac{u_{i1} - 1}{uf_{i1}^{s_{i1}} - 1}(A_i(c_i^{s_{i1}}) - A_i(c_i^1)) + A_i(c_i^1). \quad (8)$$

where $uf_{i1}^{s_{i1}}$ is the unrolling factor for the innermost-level loop L_{i1} in loop L_i, and $c_i^r = \langle u_{i1}, u_{i2}, ..., u_{im_i} \rangle$ is the unrolling configuration for all the nested levels of loop L_i. We obtain $A_i(c_i^1)$ and $A_i(c_i^{s_{i1}})$ through HLS tools. To estimate the area for the entire application, we sum the area of the individual loops in the kernel.

2) Performance Prediction Model: We have observed that the execution time E_i of a loop L_i can be estimated using *iteration latency*[1] of its innermost-level loop L_{i1}. We define $IL = \{IL_{i1}^{uf_{i1}^1}, IL_{i1}^{uf_{i1}^2}, ..., IL_{i1}^{uf_{i1}^v}, ..., IL_{i1}^{uf_{i1}^{s_{i1}}}\}$ as a set containing iteration latency with all available unrolling factors UF_{i1} of L_{i1}. The loop bound set of loop L_i is $B_i = \{B_{i1}, B_{i2}, ..., B_{ij}, ..., B_{im_i}\}$, where B_{ij} is the loop bound of the j^{th}-level loop L_{ij}. Then, execution time E_i of a loop L_i with configuration $\langle u_{i1}, u_{i2}, ..., u_{im_i} \rangle$ can be estimated as follows,

$$E_i = \begin{cases} IL_{i1}^{u_{i1}} \cdot \dfrac{B_{i1}}{u_{i1}} + c & \text{if } m_i{=}1, \quad (9a) \\ \left[\left(\left(IL_{i1}^{u_{i1}} \cdot \dfrac{B_{i1}}{u_{i1}} + c\right)\dfrac{B_{i2}}{u_{i2}} + c\right)\cdots\right]\dfrac{B_{im_i}}{u_{im_i}} + c & \text{if } m_i{\geq}2 \quad (9b) \end{cases}$$

where, c is a constant representing extra cost. We set c as 2 in our work to account for the additional cycles in entering and exiting the loop. For a single-level loop L_i ($m_i = 1$), its execution time E_i is calculated as the iteration latency $IL_{i1}^{u_{i1}}$ with unrolling configuration u_{i1} multiplied by the number of iterations after unrolling $\frac{B_{i1}}{u_{i1}}$. The execution time E_i with $m_i \geq 2$ is calculated similarly.

For our performance and area estimation, we need to sample a few configurations by invoking the HLS tools. For example, the benchmark MTT2 has a design space with 15,552 design points, whereas the number of pre-invocation of HLS is only 30.

For a kernel $\mathcal{K}$ containing multiple loops without dataflow feature, the initiation interval $II_{\mathcal{K}}$ is calculated by Equation 7a; for a kernel $\mathcal{K}$ with dataflow feature, the initiation interval $II_{\mathcal{K}}$ of the kernel is constrained by the most time-consuming (worst-case execution time) loop as shown in Equation 7b.

C. Algorithm Description

The *Dataflow Detection* component checks whether dataflow feature can be enabled for the application. Depending on the outcome, we use different search algorithms.

1) Search Algorithm with Dataflow Feature

Algorithm 1 is used to select the Pareto-optimal design points $\vec{DP}$ for a kernel $\mathcal{K}$ with dataflow feature. The general idea of this algorithm is that we always focus on optimizing the longest loop to minimize the II of the kernel. Once the original longest loop is optimized and is no longer the bottleneck, we switch to explore the new longest loop for minimizing the II.

A Pareto-optimal design point here is represented as $(KC^t, II_{\mathcal{K}}^t, \mathcal{A}_{\mathcal{K}}^t)$, where KC^t is a kernel configuration with an initiation interval $II_{\mathcal{K}}^t$ and area $\mathcal{A}_{\mathcal{K}}^t$. This algorithm uses the estimated execution time and area from the prediction models mentioned in III-B.

Algorithm 1 Line 3-5 shows the initiation step of our method. We first sort the execution time E_i of all available configurations for each loop L_i in descending order and store it in a queue $EQueue$. $EQueue$ is an ordered queue consisting of a set of triples (E_i, A_i, c_i^r), where c_i^r is the configuration of loop L_i with area A_i. Initially, the execution time E_i of a loop L_i is assigned as the longest execution time E_i^{max} of loop L_i among its available configurations.

$\vec{DP}$ for this kernel is then searched in Line 6-18. We find loops lm and ls with the longest E_{lm}^{max} and second longest execution time E_{ls}^{max}. Our algorithm focuses on finding all the kernel configurations that are bottlenecked by loop lm. Until

[1] *Iteration latency* (IL) is the latency for a single iteration of the loop

Algorithm 1: Search algorithm with dataflow feature

Input: A kernel $\mathcal{K}$ containing multiple loops, FPGA platform constraint $AreaCons$
Output: The design points $\vec{DP}$ of the kernel

begin
  $exit\_flag \leftarrow 0$;
  foreach $L_i \in \mathcal{K}$ **do**
    $EQueue_i \leftarrow$ sort execution time E_i of all configurations (E_i, A_i, c_i^r) in descending mode;
    $E_i^{max} \leftarrow$ the largest element in $EQueue_i$;
  while !$exit\_flag$ **do**
    $continue\_flag \leftarrow 1$;
    $lm \leftarrow$ a loop with $\max_{i=1,...,n} E_i^{max}$ in $\mathcal{K}$;
    $ls \leftarrow$ a loop with the second longest execution time;
    while $continue\_flag$ **do**
      if $E_{lm}^{max} > E_{ls}^{max}$ **then**
        store configuration KC^t, initiation interval $II_{\mathcal{K}}^t$ and area $\mathcal{A}_{\mathcal{K}}^t$ in $\vec{DP}$ as a design point;
        remove top element from $EQueue_{lm}$;
        update E_{lm}^{max} with the new largest element in $EQueue_{lm}$;
      else
        $continue\_flag \leftarrow 0$;
      if $EQueue_{lm} == \emptyset$ **then**
        $exit\_flag \leftarrow 1$;
  return $\vec{DP}$;

execution time E_{lm}^{max} is larger than E_{ls}^{max} of loop ls, we select a kernel configuration KC^t consisting of current configuration of loop lm and minimum area configurations of the remaining loops. Based on KC^t, we calculate the initiation interval $II_{\mathcal{K}}^t$ and area $\mathcal{A}_{\mathcal{K}}^t$ of this kernel and add $(KC^t, II_{\mathcal{K}}^t, \mathcal{A}_{\mathcal{K}}^t)$ into $\vec{DP}$. After this, we remove the top most element in queue $EQueue_{lm}$ of loop lm to avoid storing redundant points and update the longest execution time E_{lm}^{max} of loop lm with the new top element in $EQueue_{lm}$; otherwise, the loop with the largest execution time will be loop ls in which case we start exploring the new most time-consuming loop.

Since initiation interval $II_{\mathcal{K}}$ of the kernel is dominated by the longest execution time of loop lm as shown in Equation 7b, we terminate the algorithm by returning $\vec{DP}$ when all levels in loop lm have been explored, i.e. ($EQueue_{lm} = \emptyset$).

2) **Search Algorithm without Dataflow Feature**

Algorithm 2 performs design space exploration for kernels without dataflow feature. The general idea of this algorithm is that starting from the area budget A equal to the original kernel area $\mathcal{A}_{\mathcal{K}}^o$ without any optimizations (no loop unrolling), we increase A in steps of Δ. For each area budget A, we search the combination of configurations of loops in the kernel with

Algorithm 2: Search algorithm without dataflow feature

Input: a kernel $\mathcal{K}$, FPGA platform constraint $AreaCons$
Output: The design points $\vec{DP}$ of the kernel

begin
  for $A = \mathcal{A}_{\mathcal{K}}^o$ *to* $AreaCons$ *in steps of* Δ **do**
    $\mathcal{K}_1(A) = \min_{\substack{\forall c_1^r \in C_1 \\ A(c_1^r) \le A}} \{E_1(c_1^r)\}$;
  for $A = A_{\mathcal{K}}^o$ *to* $AreaCons$ *in steps of* Δ **do**
    for $i = 2$ *to* n **do**
      $\mathcal{K}_i(A) = \min_{\substack{c_i^r \in C_i \\ A(c_i^r) \le A}} \left\{ E_i(c_i^r) + \mathcal{K}_{i-1}(A - A(c_i^r)) \right\}$;
    store the configuration, area, execution time $\mathcal{K}_n(A)$ in $\vec{DP}$ as a design point;
  return $\vec{DP}$;

the minimum execution time $\mathcal{E}_{\mathcal{K}}$ and store it into $\vec{DP}$.

ΔA_i is the area difference between $A_i(c_i^r)$ and $A_i(c_i^{r1})$. The step value Δ is calculated as the minimum area difference between two configurations of any loop in the kernel.

$$\Delta = \min_{i=1,...,n} \Delta A_i = \min_{\substack{i=1,...,n \\ c_i^r, c_i^{r1} \in C_i}} \left(A_i(c_i^r) - A_i(c_i^{r1}) \right) \quad (10)$$

Let $\mathcal{K}_i(A)$ be the minimum execution time for a kernel $\mathcal{K}$ considering loops $L_1, L_2, ..., L_i$ under an area budget A. Let $E_i(c_i^r)$ be the execution time of loop L_i with the unrolling configuration c_i^r. Then $\mathcal{K}_i(A)$ can be defined recursively.

$$\mathcal{K}_i(A) = \min_{\substack{c_i^r \in C_i \\ A(c_i^r) \le A}} \left\{ E_i(c_i^r) + \mathcal{K}_{i-1}(A - A(c_i^r)) \right\} \quad (11)$$

That is, given an area budget A, we explore all the possible configurations of loop L_i and select the one that results in the minimum execution time for the kernel $\mathcal{K}$ considering loops $L_1, L_2, ..., L_i$. The base case for loop L_1 is calculated by the following equation,

$$\mathcal{K}_1(A) = \min_{\substack{\forall c_1^r \in C_1 \\ A(c_1^r) \le A}} \{E_1(c_1^r)\} \quad (12)$$

The minimum execution time for loop $L_1, L_2, ..., L_n$ within the area budget A corresponds to $\mathcal{K}_n(A)$. Based on the Equation 7a the initiation interval $II_{\mathcal{K}}$ is equal to $\mathcal{K}_n(A)$.

IV. EXPERIMENTAL RESULTS

In this section, we first describe the experimental setup for the evaluation of our method. Next, we present the experimental results.

A. Experimental Setup

In order to evaluate the effectiveness of our method, we utilize six benchmarks. Each benchmark consists of multiple nested loops with or without dataflow feature. These benchmarks are image processing applications used in [10] and an automotive Multi-Target Tracking System (MTT) application [21]. Table II summarizes the benchmarks used in our evaluation. As the huge design space (around 2^{64} points) of the MTT benchmark surpasses the capability of exhaustive search, we manually split it into four smaller kernels, each performing a different stage in the application.

Benchmark	Description	Type	Explorable Operations
Rician	Image Rician Deconvolution	Dataflow/ No Dataflow	loop(4,4)[1], loop (4,4)
Seidel	Seidel stencil computation	No Dataflow	loop(6,6), loop(6,6)
MTT1	Kernel 1 of MTT	Dataflow/ No Dataflow	loop(3,3), loop(3,3,3), loop(3,3,3)
MTT2	Kernel 2 of MTT	Dataflow/ No Dataflow	loop(3,3), loop(2,3,3), loop(2,3,2), loop(2,2)
MTT3	Kernel 3 of MTT	Dataflow/ No Dataflow	loop(2,2), loop(3,3,2), loop(3,2,2), loop(2,3)
MTT4	Kernel 4 of MTT	No Dataflow	loop(2), loop(3,2), loop(3), loop(3,2,3), loop(3,3)

[1] loop(num$_1$,num$_2$,...,num$_m$) represents a loop that has m-level loops with each level containing num$_i$ configurations;

TABLE II: Benchmarks

We run our DSE algorithm on Intel Xeon CPU E5-2620 core running at 2.10GHz with 64GB RAM. The target FPGA platform is Xilinx ZC702 Evaluation Kit [22] and we utilize Xilinx Vivado HLS version 2013.3 to synthesize the C code into Verilog RTL and obtain performance/area information for all the design points of our benchmarks. Vivado HLS reports the minimum and maximum II. We use the maximum II to accommodate the worst case performance. The area metric defined by Equation 2 is used for FPGA area.

B. Experimental Results

The error of the area/performance models is calculated by the arithmetic mean of difference between real results via Vivado HLS and predictions for all the configurations. Experimental results show 4.05% and 3.92% error for the area and performance prediction model for our six benchmarks, respectively. This demonstrates the accuracy of our prediction models.

For evaluating our approach, we exhaustively run all the design point combinations for all benchmarks to obtain the Pareto-optimal curve as our reference. Figure 4 shows the Pareto-optimal curves for all benchmarks using exhaustive search and our method. It provides an intuitive visual summary of exploration results. MTT1 benchmark has dataflow feature and its Pareto-optimal curves applying the dataflow or non-dataflow pragma using exhaustive method are plotted in Figure 4c. It can be observed that the Pareto-optimal curve with dataflow (green line with stars) has higher quality performance-area trade-offs than the curve without dataflow (red line with rectangles). This confirms our observations from the motivating example in section II. The approximate Pareto-optimal curve *DSE_WITH_DF* (orange line with triangles), obtained by our method follows the trend of the Pareto-optimal curve with dataflow *Exhaustive_With_DF* (green line with stars) and is quite close to it. However, from the figure, we can observe that the approximate Pareto-optimal curve by our method does not cover all the Pareto-optimal design points on the curve *Exhaustive_With_DF*. The loss of a small set of Pareto-optimal points is a side-effect of errors introduced by performance/area prediction models. The rest of the graphs in Figure 4 illustrate similar behavior for other benchmarks.

Moreover, in order to measure the quality of an approximate Pareto-optimal curve, we borrow the metric of *average distance from reference set* (ADRS) utilized by [13][20]. In our case, we consider a two-objectives (Initiation Interval II vs. area $\mathcal{A}$) DSE problem. ADRS is used to measure the distance between an exact Pareto-optimal set $\Pi = \{\pi_1, \pi_2, ...|\pi_i = (l, a), l \in II, a \in \mathcal{A}\}$ and an approximate Pareto-optimal set $\Lambda = \{\lambda_1, \lambda_2, ...|\lambda_j = (l, a), l \in II, a \in \mathcal{A}\}$:

$$ADRS(\Pi, \Lambda) = \frac{1}{|\Pi|} \sum_{\pi \in \Pi} \min_{\lambda \in \Lambda} \delta(\pi, \lambda) \tag{13}$$

where δ is defined by,

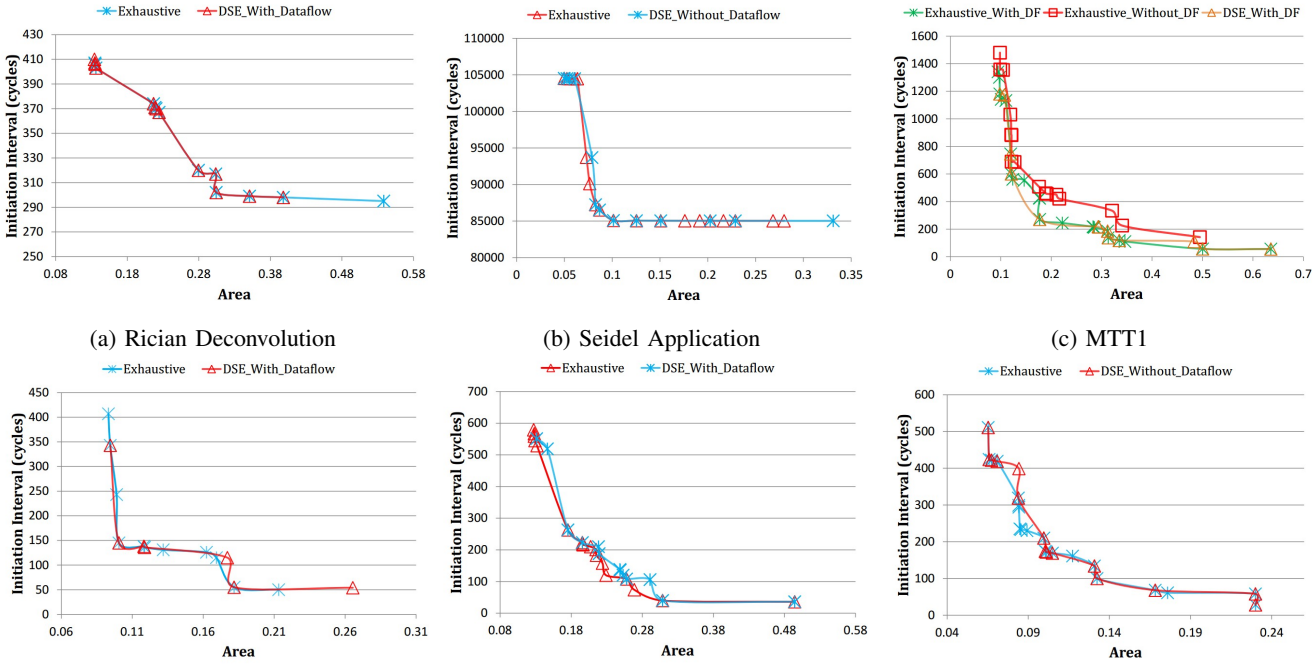
$$\delta(\pi = (l_\pi, a_\pi), \lambda = (l_\lambda, a_\lambda)) = \max\{0, \frac{l_\lambda - l_\pi}{l_\pi}, \frac{a_\lambda - a_\pi}{a_\pi}\}.$$

ADRS is usually represented by percentage. The lower the ADRS, the better is the quality of the approximate set Λ with respect to Π. Table III summarizes the ADRS for our method. The maximum difference among the benchmarks is less than 4%, which means the approximate Pareto-optimal curves obtained by our method are of high quality.

Benchmarks	ADRS(%)
Rician	0.08
Seidel	0.62
MTT1	2.51
MTT2	2.67
MTT3	3.67
MTT4	3.66

TABLE III: Average ADRS of All Benchmarks

To demonstrate the efficiency of our approach, we also compare exploration time for obtaining the approximate Pareto curve using our approach with that of exhaustive method. For the exhaustive method, the total exploration time consists of the time to run Vivado HLS for all the design points. For our approach, the total exploration time comprises of selective invocation of Vivado HLS and the time spent on our algorithm. It is important to note that time spent on the algorithm is negligible when compared to that of Vivado HLS for one configuration. In addition, we also compare with the method proposed in Schafer's work [2]. However, their work [2] ignores dataflow feature. For a fair comparison, we extend their technique with dataflow feature. The algorithm they have proposed has two steps: (1) Exhaustively perform DSE using



(a) Rician Deconvolution (b) Seidel Application (c) MTT1

(d) MTT2 (e) MTT3 (f) MTT4

Fig. 4: Comparison of Pareto-optimal Curves with Exhaustive Method versus our Approximate DSE

Benchmarks	Design Points	Number of HLS invocation			Exploration Time (s)			Speedup		
		Exhaustive	[2]	Our	Exhaustive	[2]	Our	Exhaustive	[2]	Our
Rician	512	512	181	19	5349.90	1891.27	198.53	1	2.83	26.95
Seidel	1296	1296	361	31	22121.44	6161.91	529.14	1	3.59	41.81
MTT1	13122	13122	246	43	132006.40	2474.74	432.58	1	53.34	305.16
MTT2	15552	15552	374	30	148978.40	3582.69	287.38	1	41.58	518.4
MTT3	10368	10368	364	31	74273.09	2607.58	222.07	1	28.48	334.45
MTT4	5832	5832	230	32	34397.76	1356.57	188.74	1	25.36	182.25

TABLE IV: Design Space Exploration Time Comparison

HLS for each loop in a kernel and extract the Pareto-optimal sets for individual loops and (2) Combine the configurations of design points in the Pareto-optimal sets for individual loops and invoke HLS again for extracting the Pareto-optimal set for the kernel. The exploration time comparison is shown in Table IV. Our method is, on an average, 235x faster than exhaustive method. The speedup increases as design space enlarges. For MTT2 kernel, the speedup is up to 520x compared with exhaustive method. Moreover, we obtain 9x speedup in exploration on an average compared to [2].

In summary, our technique can perform design space exploration efficiency and return design points with high quality.

V. Related Work

Design space exploration (DSE) for FPGAs is a multi-objective optimization problem. The problem is to resolve conflicting objectives by finding the points on the Pareto-optimal curve. Typical objectives for this exploration are performance (latency/throughput/initiation interval) and area. Existing approaches in DSE for FPGAs can be classified into the following two categories:

Compiler techniques: [5][12][17] estimate performance and area at control data flow graph (CDFG) level. They perform DSE starting from direct loop transformations and apply diverse compiler optimization techniques to generate different architectures with fast estimated execution time and area. Bilavarn et al. [5] and So et al. [17] consider performance/area trade-offs regarding loop transformations such as loop unrolling. So et al. utilize a balance metric to prune configuration space in their DSE algorithm, while Bilavarn et al. perform an exhaustive search for all possible configurations to find the Pareto-optimal curve. Holzer et al.[12] introduce an evolutionary multi-objective optimization approach to the find Pareto-optimal curve. However, all these works focus on one loop.

HLS tools as a blackbox: [2][3][14][18][20][27] explore the design space using commercial high level synthesis tool as a blackbox. Schafer et al. propose a divide and conquer algorithm [2] for solving HLS design space exploration prob-

lems. They first parse kernels into a set of clusters which consist of loops, functions and arrays. Then they exhaustively search each cluster by invoking HLS tools with all possible configurations to find the local Pareto-optimal points. Finally, they combine the local Pareto-optimal configurations and invoke HLS tools again to find the global Pareto-optimal points. As they need actual simulation/synthesis to acquire design points at every step, their method suffers from long simulation/synthesis runtime. Instead of invoking HLS tools frequently, Schafer et al.[3] and Liu et al.[20] propose machine learning algorithms for this problem. Learning-based algorithms guide the design space exploration by predicting design points. This helps to reduce the total runtime to perform design space exploration. Compared with local-search algorithms, learning-based methods require shorter simulation/synthesis runtime. However, the learning-based approaches search all possible configurations without any pruning. Apart from the time-consuming training step to obtain a learning model, the learning model is only trained with one application in [20]. Thus the accuracy of this learning model for a new application with different features is not clear.

Prior design space exploration techniques using HLS [2][3][5][12][14][17][20] focus primarily on nested loops ignoring dataflow (producer-consumer) dependence among them. Design space exploration for loops with dataflow dependence is still not well studied. This work proposes efficient design space exploration techniques for applications that consist of multiple nested loops with or without dataflow dependence. In addition, instead of searching all possible configurations [2][3][14][20], we prune the design space by eliminating the dominated configurations. Accurate performance and area models are also developed to assist the design space exploration to reduce number of invocation of the HLS tool. To show the accuracy and efficiency of our design space exploration technique, various scientific kernels and real world applications are tested in our work.

VI. CONCLUSION

We have presented an efficient and accurate design space exploration technique using HLS for applications consisting of multiple nested loops with or without data dependencies. Experimental results demonstrate that our method can perform DSE for applications with huge design space (more than 10,000 design points) and provide an approximate Pareto-optimal curve within at most nine minutes. The proposed method runs 235x faster than exhaustive search and 9x faster than Schafer's method [2] on an average. The quality of the obtained Pareto-optimal curve is very close to the optimal. The efficiency and accuracy of our method open up opportunities for design space exploration of more complex application kernels on FPGAs using HLS.

ACKNOWLEDGMENT

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2012-T2-1-115 and French-Singaporean Merlion 2012 PhD Project.

REFERENCES

[1] Altium Inc. Altium Designer. http://www.altium.com.
[2] Schafer B. and Wakabayashi K. Divide and Conquer High-level Synthesis Design Space Exploration. *ACM Trans. Des. Autom. Electron. Syst.*, 2012.
[3] Schafer B. and Wakabayashi K. Machine Learning Predictive Modelling High-Level Synthesis Design Space Exploration. *Computers Digital Techniques, IET*, 2012.
[4] Cadence Inc. C-to-Silicon Compiler, 2012. http://www.cadence.com.
[5] Bilavarn S. et al. Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2006.
[6] Bordoloi U.D. et al. Evaluating Design Trade-offs in Customizable Processors. In *Design Automation Conference (DAC), 2009*.
[7] Canis A. et al. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, 2011.
[8] Chen D. et al. xPilot: A Platform-Based Behavioral Synthesis System. In *Proceedings of SRC Techcon Conf*, 2005.
[9] Cong J. et al. Platform-Based Behavior-Level and System-Level Synthesis. In *SOC Conference, 2006 IEEE International*, 2006.
[10] Cong J. et al. Customizable Domain-Specific Computing. *Design Test of Computers, IEEE*, 2011.
[11] Cong J. et al. Combining Module Selection and Replication for Throughput-driven Streaming Programs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, 2012.
[12] Holzer M. et al. Design Space Exploration with Evolutionary Multi-Objective Optimisation. In *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, 2007.
[13] Palermo G. et al. ReSPIR: A Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2009.
[14] Papakonstantinou A. et al. Multilevel Granularity Parallelism Synthesis on FPGAs. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '11, 2011.
[15] Papakonstantinou A. et al. Throughput-oriented Kernel Porting onto FPGAs. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, 2013.
[16] Sbalzarini I. F. et al. Multiobjective Optimization Using Evolutionary Algorithms. In *Proceedings of the Summer Program*, 2000.
[17] So B. et al. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, 2002.
[18] Xydis S. et al. A Meta-model Assisted Coprocessor Synthesis Framework for Compiler/Architecture Parameters Customization. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 2013.
[19] Zuo W. et al. Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, 2013.
[20] Liu H. and Carloni L.P. On Learning-based Methods for Design-space Exploration with High-Level Synthesis. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, 2013.
[21] Liu H. and Niar S. Radar Signature in Multiple Target Tracking System for Driver Assistant Application. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 887–892, March 2013.
[22] Xilinx Inc. ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide, 2013.
[23] NEC Inc. CyberWorkBench. http://www.nec.com.
[24] Gupta S. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, 2003.
[25] Synopsys Inc. Synphony High-Level Synthesis Solution, 2012. http://www.synopsys.com.
[26] Xilinx Inc. Vivado High-Level Synthesis. http://www.xilinx.com.
[27] Sotirios Xydis, Kiamal Pekmestzi, Dimitrios Soudris, and George Economakos. A High Level Synthesis Exploration Framework with Iterative Design Space Partitioning. In *VLSI 2010 Annual Symposium*, 2011.