Resource-Aware Throughput Optimization for High-Level Synthesis

Peng Li[1] Peng Zhang[1] Louis-Noël Pouchet[2,1] Jason Cong[1]
{pengli, pengzh, pouchet, cong}@cs.ucla.edu
[1]Computer Science Department, University of California, Los Angeles
[2]Department of Computer Science and Engineering, The Ohio State University

ABSTRACT

With the emergence of robust high-level synthesis tools to automatically transform codes written in high-level languages into RTL implementations, the programming productivity when synthesizing accelerators improves significantly. However, although the state-of-the-art high-level synthesis tools can offer high-quality designs for simple nested loop kernels, there is still a significant performance gap between the synthesized and the optimal design for real-world complex applications with multiple loops.

In this work we first demonstrate that maximizing the throughput of each individual loop is not always the most efficient approach to achieving the maximum system-level throughput. More area-efficient non-fully pipelined design variants may outperform the fully-pipelined version by enabling larger degrees of parallelism. We develop an algorithm to determine the optimal resource usage and initiation intervals for each loop in the applications to achieve maximum throughput within a given area budget. We report experimental results on eight applications, showing an average of 31% performance speedup over state-of-the-art HLS solutions.

Keywords

High-level Synthesis; Throughput Optimization; Resource Sharing; Area Constraint

1. INTRODUCTION

The automatic transformation of algorithms written in high-level languages (e.g., C, C++, SystemC) to low-level implementations by high-level synthesis (HLS) can be the key enabling technology to address the programmability challenges of FPGA-based accelerator architectures. After three decades of research and development by academia and industry, HLS has become a promising productivity boost for the semiconductor industry [1]. For example, the Xilinx Vivado HLS tool [2] based on AutoPilot [3] is now part of the standard Xilinx Vivado Suite available to every Xilinx FPGA designer.

With effective operation scheduling and resource binding/sharing algorithms, the state-of-the-art HLS tools can generate highly optimized RTL for a single module [4][1]. However, fully utilizing the impressive performance provided by FPGA devices often requires the designs to be implemented with massive parallelism. Traditional FPGA design flows follow a two-step approach. First, a single replica of the application is optimized intensively for best performance through efficient computation pipelines. Then the optimized replica is duplicated to fully utilize the overall capacity of the target FPGA device.

It is commonly believed that efficient loop pipelining can boost the performance of a single loop with acceptable area overhead. For example, [5] shows that with full loop pipelining enabled by using on-chip memory partitioning, the evaluated computation kernels can achieve an average speedup of 5.6X at the cost of only 45% area increase. While this is true for a single loop, for accelerators with multiple sequential loops, if the area increase to support loop pipelining is not shared efficiently across loops, the overall performance-area efficiency may be degraded.

In this paper, we first show a somewhat counter-intuitive example that maximizing the pipeline throughput of one replica will not always generate optimal designs with coarse-grained duplications. With less aggressive loop pipelining, logic components may be more efficiently shared among loops with better performance per area ratio. Duplicating such a design will achieve a higher global throughput. The optimize-then-duplicate approach fails to find such optimal solutions.

Then, we propose a resource-aware throughput optimization algorithm to identify the most efficient implementations by maximizing the resource sharing between different loops. The problem is different from traditional resource-constrained operator scheduling [6][7][8] or resource allocation and binding [9][10][11]. Instead of fine-grain control-data flow graphs (CDFGs), the inputs of our algorithm are loops with parameterized initiation interval (II). Different loop II will generate loop implementations with a different number of shareable components, and the objective is to improve the throughput by maximizing area efficiency through resource sharing.

The work that is most relevant to this paper is [12], [13] and [14]. [12] focused on using module replication to meet the throughput target for streaming applications. [13] and [14] proposed a combination of module selection and replication algorithm based on a synchronous data flow (SDF) representation of streaming applications. All of these focus on streaming applications where all modules are executed in parallel and no resource can be shared among different modules. The techniques proposed in this paper can be applied to streaming applications as a special case, but also to more general applications where not all loops can be executed in parallel due to inter-loop dependence. For example, iterative stencil applications [15] are such examples with loop-carried dependence.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA'15, February 22–24, 2015, Monterey, CA, USA.
Copyright 2015 ACM 978-1-4503-3315-3/15/02 ...$15.00.
http://dx.doi.org/10.1145/2684746.2689065 .

The contributions of the work include:

- We address the fact that high throughput replicas with efficient computation pipelines may not be good candidates for building massive parallel accelerators. Balanced replicas with efficient resource sharing between loops can be better building blocks.
- We formulate a resource-aware throughput optimization problem by simultaneously addressing the concept of loop pipelining, module selection, duplication and resource sharing within a unified framework. To the best of our knowledge, this the first attempt to co-optimize these problems within a single formulation.
- We make some initial attempts to solve the problem efficiently with branch-and-bound and ILP approaches.

This paper is organized as follows. Section 2 uses a motivational example to demonstrate the design trade-offs in resource-aware throughput optimization. Section 3 formulates the problem and Section 4 proposes several methods to solve the problem efficiently. Section 5 describes some implementations in more details. Section 6 presents the efficiency of the proposed throughput optimization algorithm over traditional approaches with experimental results and the conclusions are presented in Section 7.

2. A MOTIVATIONAL EXAMPLE

We now illustrate the resource-aware throughput optimization problem using a motivational example: Discrete Wavelet Transformation (DWT), an algorithm frequently used in various multimedia applications.

Fig. 1 illustrates a computation kernel in DWT. Array tmp is a floating-point (FP) array and $a1, a2, a3, a4, k1, k2$ are floating-point variables. There are four loops in the code segment, namely L1 to L4. All the FP operations $(+, *)$ are emphasized in the code with $(\oplus, \otimes)$.

```
for i=0 to n-1 do
  L1:for j = 1 to m-3 step 2 do
    tmp[i][j]⊕=a1⊗(tmp[i][j-1]⊕tmp[i][j+1]);
  tmp[i][m-1]⊕=2⊗a1⊗tmp[i][m-2];
  L2:for j = 2 to m-1 step 2 do
    tmp[i][j]⊕=a2⊗(tmp[i][j-1]⊕tmp[i][j+1]);
  tmp[i][0]⊕=2⊗a2⊗tmp[i][1];
  L3:for j = 1 to m-3 step 2 do
    tmp[i][j]⊕=a3⊗(tmp[i][j-1]⊕tmp[i][j+1]);
    img[j/2 + m/2][i] = k2⊗tmp[i][j];
  tmp[i][m-1] = 2⊗a3⊗tmp[i][m-2];
  img[(m-1)/2 + m/2][i]=k2⊗tmp[i][m-1];
  L4:for j = 2 to m-1 step 2 do
    tmp[i][j]⊕=a4⊗(tmp[i][j-1]⊕tmp[i][j+1]);
    img[j/2][i] = k1⊗tmp[i][j];
  tmp[i][0] = 2⊗a4⊗tmp[i][1];
  img[0][i] = k1⊗tmp[i][0];
```

Figure 1: Discrete Wavelet Transformation (DWT)

Table 1 shows the FP operations performed in each loop and FP operators required if each loop will be fully pipelined. Due to the inter-loop data dependence, all loops must be executed sequentially. To save area, FP operators can be shared among loops. A total of two FP adders and two FP multipliers are needed for all the loops to be pipelined.

Table 1: FP Ops in DWT (Fully Pipelined)

	L1	L2	L3	L4
FP operations	$2\oplus, 1\otimes$	$2\oplus, 1\otimes$	$2\oplus, 2\otimes$	$2\oplus, 2\otimes$
Loop II	1	1	1	1
FP operators	2+ , 1*	2+ , 1*	2+ , 2*	2+ , 2*

Fig. 2 illustrates the execution graph of all the FP operators. One FP multiplier is idle during the execution of loops L1 and L2. Considering replication to exploit coarse-grained parallelism, low area utilization will likely degrade the overall performance.

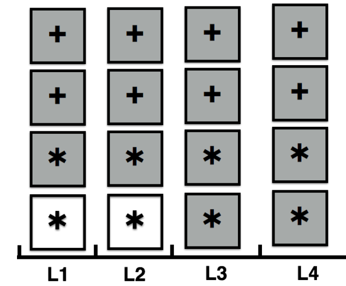

Figure 2: Fully Pipelined DWT Implementation

To improve the utilization of FP multipliers, we can have an alternative implementation that instantiate only one FP multiplier. Under such circumstances, loops L3 and L4 cannot be fully pipelined due to resource constraints. Table 2 shows the FP operations, loop II and FP operators under this configuration. A total of two FP adders and one FP multipliers are needed.

Table 2: FP Ops in DWT (Non-fully Pipelined)

	L1	L2	L3	L4
FP operations	$2\oplus, 1\otimes$	$2\oplus, 1\otimes$	$2\oplus, 2\otimes$	$2\oplus, 2\otimes$
Loop II	1	1	2	2
FP operators	2+ , 1*	2+ , 1*	1+ , 1*	1+ , 1*

Fig. 3 illustrates the execution graph of all the FP operators in the non-fully pipelined scenario. The execution time L3 and L4 will be longer compared to the case of Fig. 2 due to the larger loop II. However, with $II = 2$, each fully-pipelined FP adder and multiplier can be virtualized to support two FP operations in each loop iteration. In this case, one FP adder is always idle during the execution of L3 and L4.

We use Xilinx Virtex-7 XC7V585T FPGA device as a test platform to evaluate the throughput of the two implementations. 512*512 image size is used in the experiments. Experimental results are shown in Table 3. Area and critical path data are reported by Vivado after place and route. Cycle data are reported by Vivado HLS RTL-level simulation. All performance numbers are normalized against the performance of the single replica performance of the fully-pipelined implementation. From the table, we observe that the single-replica performance of the fully-pipelined version is

Table 3: Experimental Results for DWT

Implementation	LUT	FF	DSP	Cycles	CP(ns)	Replicas	Nomalized $\frac{Perf}{Replica}$	Nomalized System Perf
Fully	2855	1985	28	590337	8.800	45	1	45
NonFully	2716	1830	17	851457	8.821	74	0.71	50.7

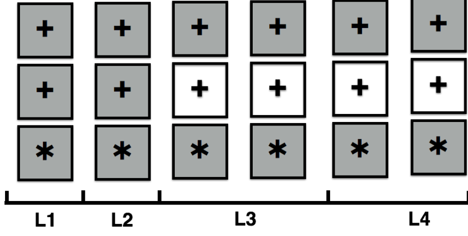

Figure 3: Non-fully Pipelined DWT Implementation

29% higher than the non-fully-pipelined version. However, with $74/45 - 1 = 64\%$ more replicas, the total performance of the non-fully-pipelined version is 13.7% higher than the fully-pipelined version after replication.

The relative efficiency comparison between the two DWT implementations will depend on many factors, including the area of both FP operators, the area of other components, trip counts of all loops and the available resource in the target device. In this paper we formulate the resource-aware throughput optimization problem with all these variables and propose some efficient solutions to solve the problem.

3. PROBLEM FORMULATION

From the motivational example, we can see that each loop in the accelerator can have different implementations with different resource usage and performance. Some resource can be shared among loop kernels. The challenge is to find the design with optimal performance under a given area constraint with resource sharing. In this section we will provide a formulation of the problem. The resource-aware throughput optimization problem can be formulated as:

$$\text{Maximize} \quad D * Performance_{PerReplica} \tag{1}$$

$$\text{Subject to} \quad D * Area_{PerReplica} \leq Area_{available} \tag{2}$$

where $Performance_{PerReplica}$ and $Area_{PerReplica}$ represent the performance and area of single replica and D are the number of replicas to be duplicated at coarse-grained level.

With resource sharing, the total area of the entire accelerator will probably be less than the sum of each loop kernels.

3.1 Shareable Component Candidate

Logic components (such as operators) can be shared among different statements/loops to save area. However, multiplexers introduced by component sharing is nontrivial on the FPGA platform. For example, the area, delay and power data of a 32-to-1 multiplexer are almost equivalent to an 18-bit multiplier in modern FPGA designs [16]. Therefore, not all components will benefit from resource sharing. The state-of-the-art HLS tools will only share the components with enough complexity compared to the I/O multiplexers, or *shareable component candidate.*

DEFINITION 1 (SHAREABLE COMPONENT CANDIDATE). *A* Shareable Component Candidate *is a component whose area is at least δ times larger than a 32-bit 8-to-1 multiplexer.*

$$\frac{Area_{Component}}{Area_{mux}} \geq \delta$$

On Xilinx Virtex-7 FPGAs, a 32-bit 8-to-1 multiplexer will take 96 LUTs and δ is set to 10 in our experiment. Table 4 shows the estimated area consumption of some shareable components on Virtex-7 FPGAs.

Table 4: Area of Example Shareable Component Candidates

	LUT	FF	DSP
DP $\pm$	781	445	3
DP *	203	299	11
DP /	3242	2230	0
$\sqrt{x}$	1919	1303	0
$\frac{1}{x}$	246	440	14
$>$	113	130	0
sin	9662	2597	17

Integer add/substract operations are not shareable component candidates.

DEFINITION 2 (LOOP SHAREABLE LOAD VECTOR). *Suppose that there are M types of shareable component candidates ($R_1, R_2, ..., R_M$) in an application, the* shareable load vector *of a loop L_k is a vector $\langle s_{1,k}, s_{2,k}, ..., s_{j,k}, ..., s_{M,k}\rangle$ where $s_{j,k}$ is the number of shareable component candidate R_j in loop L_k.*

EXAMPLE 1. *The* shareable load vector*s of loops L1 and L2 in Figure 1 are $\langle 2, 1\rangle$. The* shareable load vector*s of loops L3 and L4 in Figure 1 are $\langle 2, 2\rangle$.*

DEFINITION 3 (LOOP RESOURCE ALLOCATION VECTOR). *Suppose that there are M types of shareable component candidates ($R_1, R_2, ..., R_M$) in an application, the* resource allocation vector *of a loop L_k is a vector $\langle a_{1,k}, a_{2,k}, ..., a_{j,k}, ..., a_{M,k}\rangle$ where $a_{j,k}$ is the number of shareable component candidates R_j instantiated for loop L_k.*

EXAMPLE 2. *The* resource allocation vector*s of loops L1 and L2 in Figure 1 of the non-pipelined version are $\langle 1, 1\rangle$. The* resource allocation vector*s of loops L3 and L4 in Figure 1 of the non-pipelined version are $\langle 2, 1\rangle$.*

DEFINITION 4 (TOTAL RESOURCE ALLOCATION VECTOR). *Suppose that there are N Loops ($L_1, L_2, ..., L_N$) in an application, the* resource allocation vector *of the entire accelerator is a vector $\langle a_1, a_2, ..., a_j, ..., a_M\rangle$ where a_j is the number of shareable component candidates R_j instantiated for the entire accelerator.*

EXAMPLE 3. *The* resource allocation vector *of the fully-pipelined DWT accelerator is* $\langle 2, 2\rangle$. *The* resource allocation vector *of the non-fully-pipelined DWT accelerator is* $\langle 2, 1\rangle$.

PROPERTY 1. $\forall 1 \le j \le M, \max_{k=1}^{N} a_{j,k} \le a_j \le \sum_{k=1}^{N} a_{j,k}$.

The total number of shareable component candidates instantiated in the accelerator should be no more than the total number of components instantiated in each loop.

3.2 Performance and Area Estimation

In this section we describe the performance and area estimation methodology adopted in the paper.

Loop pipelining [17] is a key optimization technique in high-level synthesis. Using the technique, parallelism across loop iterations can be exploited by initiating the next iteration of the loop before the completion of the current iteration. Operations from several iterations are overlapped by loop pipelining to decrease the total execution cycles of the entire loop. The throughput achieved is limited both by resource constraints and data dependencies in the application.

$$Cycle_k = II_k * (TC_k - 1) + Dth_k \quad (3)$$

where $Cycle_k, II_k, TC_k, Dth_k$ are the total execution cycle, pipeline initiation interval, trip count and pipeline depth of loop k respectively.

The total execution cycle of a pipelined loop can be estimated using Eq. (3). With given input data, the trip count of each loop (TC_k in Eq. 3) is fixed. The minimum initiation interval II_k is a function of the resource allocation vector $\langle a_{1,k}, a_{2,k}, ..., a_{M,k}\rangle$ defined by Eq. (4). Minimal II_k is also constrained by loop-carried dependence or memory port conflict, as shown in Eq.(5). Loop depth Dth_k is also affected by the resource allocation vector, but in this paper we made a simplification by treating it as a constant. When with non-trivial trip counts, the estimation could be high accurate, as shown in Sec. 6.

$$\forall 1 \le j \le M, II_k * a_{j,k} \ge s_{j,k} \quad (4)$$
$$\forall 1 \le k \le N, II_k \ge II_k^{Min} \quad (5)$$

The area consumption of a loop can be estimated by accumulating the areas of shareable and non-shareable component candidates. Multiplexers for resource sharing are non-shareable component candidates. However, since the area of shareable component candidates are significantly larger than the multiplexers by definition, in this paper, we made a simplification by assuming that the area of non-shareable component candidates is irrelevant to the resource allocation vectors. In Sec. 6 we will validate these assumptions by comparing the estimated results and synthesized results with real examples.

Suppose the resource allocation vector in the entire accelerator is $\langle a_1, a_2, ..., a_M\rangle$, the total area of the application can be estimated by:

$$Area = \sum_{k=1}^{N} Area_k^{NS} + \sum_{j=1}^{M}(Area_j^S * a_j) \quad (6)$$

where $Area_k^{NS}$ is the area of non-shareable component candidates in loop k and $Area_j^S$ is the area of shareable component candidate j.

For FPGA devices, area consumption can be represented by a vector $\langle DSP, LUT, FF\rangle$, and Equation 6 can be formulated as:

$$DSP = \sum_{k=1}^{N} DSP_k^{NS} + \sum_{j=1}^{M}(DSP_j^S * a_j) \quad (7)$$
$$LUT = \sum_{k=1}^{N} LUT_k^{NS} + \sum_{j=1}^{M}(LUT_j^S * a_j) \quad (8)$$
$$FF = \sum_{k=1}^{N} FF_k^{NS} + \sum_{j=1}^{M}(FF_j^S * a_j) \quad (9)$$

3.3 Accelerators with Total Inter-Loop Dependency

If all loops in an accelerator have to be executed in sequence due to dependence, the total execution cycles can be estimated by accumulating the cycles of individual loops, as shown in Eq. (10). In such a case, all the shareable component candidates can be reused among loops, and the total shareable component candidates can be calculated by Eq. (11).

$$Cycle = \sum_{k=1}^{N} Cycle_k \quad (10)$$
$$a_j = max_{k=1}^{N} a_{j,k} \quad (11)$$

With Eq. (3) to Eq. (11), resource-constrained performance optimization with resource sharing among loops for an accelerator with sequential loops can be formulated as:

Maximize

$$Replica/Cycle \quad (12)$$

Subject to

$$DSP = \textstyle\sum_{k=1}^{N} DSP_k^{NS} + \sum_{j=1}^{M}(DSP_j^S * a_j) \quad (13)$$
$$DSP * Replica \le DSP_{Avail.} \quad (14)$$
$$LUT = \textstyle\sum_{k=1}^{N} LUT_k^{NS} + \sum_{j=1}^{M}(LUT_j^S * a_j) \quad (15)$$
$$LUT * Replica \le LUT_{Avail.} \quad (16)$$
$$FF = \textstyle\sum_{k=1}^{N} FF_k^{NS} + \sum_{j=1}^{M}(FF_j^S * a_j) \quad (17)$$
$$FF * Replica \le FF_{Avail.} \quad (18)$$
$$\forall 1 \le j \le M, \forall 1 \le k \le N, II_k * a_{j,k} \ge s_{j,k} \quad (19)$$
$$\forall 1 \le k \le N, Cycle_k = II_k * (TC_k - 1) + Dth_k \quad (20)$$
$$\forall \text{Chain}\{L_{t_1}, ..., L_{t_L}\} \subseteq \mathcal{P}, Cycle \ge \textstyle\sum_{t=1}^{L} Cycle_{t_k} \quad (21)$$
$$\forall \text{Antichain}\ \{L_{t_1}, L_{t_2}, ..., L_{t_L}\} \subseteq \mathcal{P}, a_j \ge \textstyle\sum_{t=1}^{L} a_{j,t_k} \quad (22)$$

where D, $Cycle$, $Cycle_k$, a_j, $a_{j,k}$, II_k are integer variables, $Area$ is a rational variable, and $Area_k^{NS}$, $Area_j^S$, $Area_{Avail.}$, $s_{j,k}$, II_k^{Min}, TC_k, Dth_k are constant values.

3.4 Accelerators with Partial Inter-Loop Dependency

Multiple loops without data dependence can be parallelized by HLS tools to improve performance. In this section, we will discuss the resource-constrained performance optimization for more general cases with arbitrary dependence between loops.

Since dependence relations between loops are reflexive, antisymmetric and transitive, they can be defined as a *partial order*. The *partially ordered set*, or *poset* for short, defined by loop dependence information is referred to the *loop dependence poset* $\mathcal{P}$.

DEFINITION 5 (COMPARABLE). *For any two elements a and b of a partially ordered set P, if $a \leq b$ or $b \leq a$, then a and b are* comparable. *Otherwise they are* incomparable.

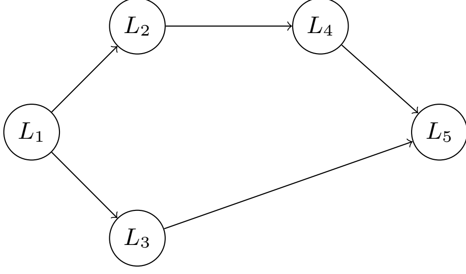

Figure 4: Example Loop Dependence Graph

In Fig. 4, L_1 are L_2 and comparable with $L_2 < L_1$; L_2 and L_3 are incomparable.

DEFINITION 6 (CHAIN). *A* chain *in a poset P is a subset $C \subseteq P$ such that any two elements in C are comparable.*

$\{L_1, L_2, L_4, L_5\}$ is a chain in Fig. 4.

DEFINITION 7 (ANTICHAIN). *An* antichain *in a poset P is a subset $A \subseteq P$ such that any two elements in A are incomparable.*

$\{L_2, L_3\}$ is an antichain in Fig. 4.

If loops in an accelerator can be parallel, the total execution cycles of the accelerator can be estimated by the maximum weighted chain of the dependence poset $\mathcal{P}$, as shown in Eq. (23).

$$\forall \text{Chain } \{L_{t_1}, L_{t_2}, ..., L_{t_P}\} \subseteq \mathcal{P}, Cycle \geq \sum_{t=1}^{P} Cycle_{t_k} \quad (23)$$

Under such circumstances, loops in an antichain may execute in parallel. Therefore, loops in an antichain will not share components with conservative scheduling. The total number of shareable component candidates can be calculated by the maximum weighted antichain of the dependence poset $\mathcal{P}$, as shown in Eq.(24).

$$\forall \text{Antichain } \{L_{t_1}, L_{t_2}, ..., L_{t_P}\} \subseteq \mathcal{P}, a_j \geq \sum_{t=1}^{P} a_{j,t_k} \quad (24)$$

Area-constrained performance optimization with resource sharing for an accelerator with arbitrary loop dependence can be formulated using Eq. (12) to (20), (23) and (24).

4. EFFICIENT SOLUTIONS

In Sec. 3, we formulated the problem of resource-constraint performance optimization with resource sharing. In the formulation, the objective and some constraints (12, 18 and 19) are not linear or posynomials (17, 20-24). Therefore, the problem cannot be directly solved by integer linear programming (ILP) or mixed integer geometric programming. In this section we will describe how to solve the problem with different approaches.

4.1 Enumeration-based Approach

The simplest approach to solving the problem is through enumeration. The initiation intervals of each loop II_k can be bounded using Eq. (25). Given loop initiation interval II_k, the shareable component candidates allocated $a_{j,k}$ can be calculated using Eq.(26). The total execution cycle and area can be estimated use Eq. (3), (6), (23) and (24).

$$\forall 1 \leq k \leq N,$$
$$II_k^{LB} \leq II_k \leq II_k^{UB} = max(II_k^{LB}, max_{j=1}^{M} s_{j,k}) \quad (25)$$
$$\forall 1 \leq j \leq M, \forall 1 \leq k \leq N, a_{j,k} = \lceil \frac{s_{j,k}}{II_k} \rceil \quad (26)$$

Note that not all IIs in the range given by Eq. (25) will possibly generate solutions with Pareto-optimal performance under resource-constraint. Some values can be easily filtered out with local loop information.

If $II_k^{(1)} < II_k^{(2)}$ and $\forall 1 \leq j \leq M, \lceil \frac{s_{j,k}}{II_k^{(1)}} \rceil = \lceil \frac{s_{j,k}}{II_k^{(2)}} \rceil$ then $II_k^{(2)}$ is not a candidate II for Pareto-optimal solution.

EXAMPLE 4. *Given a loop with shareable load vector $s = \langle 16, 5 \rangle$ and $II_{LB} = 3$, the candidate IIs for Pareto-optimal solution and allocated component vector are listed as in the following table:*

II	3	4	5	6	8	16
Resource	$\langle 6, 2 \rangle$	$\langle 4, 2 \rangle$	$\langle 4, 1 \rangle$	$\langle 3, 1 \rangle$	$\langle 2, 1 \rangle$	$\langle 1, 1 \rangle$

Only 6 different IIs out of 14 possible values can serve as candidates for Pareto-optimal solution, while other II variants can be filtered out with only local loop information.

With the increase of loops and shareable component candidates, the simple brute-force enumeration approach may not scale well. Instead of enumerating all legal transformations in a brute-force approach, the features of resource-constrained performance optimization can be used to prune suboptimal partial results to greatly reduce the computation complexity of finding optimal solutions for large designs. In this section we propose an efficient branch-and-bound algorithm for the problem considering the weights of different loops and the interaction between partial loop II selection results and maximum performance.

We observe that the execution cycle of a loop is proportional to its trip count. Therefore, design choices of loops with larger trip counts will have a larger impact on the overall resource-constrained performance. Based on this observation, we first sort the loops in the entire accelerator according to their trip counts and travel the enumeration tree branch by branch from the loop with the largest trip count to the loop with smallest trip count.

For each unenumerated loop L_k, we can estimate the lower bounds and upper bounds of area and execution cycle with Eq. (27) - (30), where II_k^{UB} is defined in Eq. (25). The lower bound of the performance at branch B is calculated by II and a_j of the enumerated loops and $Cycle^{LB}$ and a_j^{LB} of the unenumerated loops. The upper bound of the performance at branch B is calculated by II and a_j of the enumerated loops and $Cycle^{UB}$ and a_j^{UB} of the unenumerated loops.

$$a_{j,k}^{LB} = (s_{j,k} \geq 1) \quad (27)$$
$$Cycle_k^{LB} = II_k^{LB} * (TC_k - 1) + Dth_k \quad (28)$$
$$a_{j,k}^{UB} = s_{j,k} \quad (29)$$
$$Cycle_k^{UB} = II_k^{UB} * (TC_k - 1) + Dth_k \quad (30)$$

We can prove that the performance range of a child branch is always covered by that of its parent branch: $LB_{Parent} \leq LB_{Child} \leq$

$UB_{Child} \leq UB_{Parent}$. If two branches have non-overlapped performance ranges, we can prune the branch with smaller performance and all its sub-branches without losing optimality.

4.2 Integer Linear Programming

Given an accelerator with N loops and the number of candidate IIs of each loop represented as $C_1, C_2, ..., C_N$, the total enumeration number is $\prod_{k=1}^{N} C_k$. The maximum enumeration number in all our experiments will not exceed 10^4 and enumeration-based approach will finish within a second. Therefore, we just use the enumeration-based approach in our experiments. In this section, we show that the problem can be converted into an ILP optimization by parameterizing some variables and introducing extra binary variables, which could help on large scale problems.

The duplication factor D can be bounded by Eq. (31).

$$D \leq \frac{Area_{Avail.}}{(\sum_{k=1}^{N} Area_k^{NS} + \sum_{j=1}^{M} Area_j^S)} \quad (31)$$

For practical problem on the current platform, D is typically less than 100. Under such an assumption, we can parameterize the duplication factor D as a constant value and use an outer loop to enumerate D for the global optimal solution.

Then we introduce a set of binary variables $\delta_{j,k,l,m}$, where j is the type of the shareable component candidates ($1 \leq j \leq M$), k is the loop index ($1 \leq k \leq N$), l is the instance index of the shareable component candidate j in loop k ($1 \leq l \leq s_{j,k}$) and the component is scheduled to cycle m. With modulo scheduling used by loop pipelining, $1 \leq m \leq II_k^{UB}$, where II_k^{UB} is bounded by Eq. (25). $\delta_{j,k,l,m} = 1$ indicates that the l-th type-j shareable component candidate in loop k is scheduled to cycle m under modulo scheduling.

With these binary variables, we can replace the non-linear constraint Eq. (19) with a set of linear constraints, shown in Eq. (39) - (43). Eq. (40) describes a shareable component candidate is scheduled to one and only one cycle with modulo scheduling. Together with Eq. (40), Eq. (39) declares $\delta_{j,k,l,m}$ as binary variables. Eq. (41) converts resource constraint to an equivalent condition where at most $a_{j,k}$ type-j shareable component candidates can be scheduled to the same cycle for loop k. $\gamma_{j,k,m}$ in Eq. (42) equals to 1 means that at least one shareable component candidate j in loop k is scheduled to cycle m. Therefor, II_k will be at least the total number of cycles with $\gamma_{j,k,m} = 1$, as shown in Eq. (43).

Minimize

$$Cycle/D \quad (32)$$

Subject to

$$Area = \sum_{k=1}^{N} Area_k^{NS} + \sum_{j=1}^{M} (Area_j^S * a_j) \quad (33)$$

$$Area * D \leq Area_{Avail.} \quad (34)$$

$$\forall 1 \leq k \leq N, II_k \geq II_k^{Min} \quad (35)$$

$$\forall 1 \leq k \leq N, Cycle_k = II_k * (TC_k - 1) + Dth_k \quad (36)$$

$$\forall \text{Chain } \{L_{t_1}, ..., L_{t_P}\} \subseteq \mathcal{P}, Cycle \geq \sum_{t=1}^{P} Cycle_{t_k} \quad (37)$$

$$\forall \text{Antichain } \{L_{t_1}, ..., L_{t_P}\} \subseteq \mathcal{P}, a_j \geq \sum_{t=1}^{P} a_{j,t_k} \quad (38)$$

$$\forall j,k,l,m,\ \delta_{j,k,l,m} \geq 0 \quad (39)$$

$$\forall j,k,l,\ \sum_{m=1}^{II_k^{UB}} \delta_{j,k,l,m} = 1 \quad (40)$$

$$\forall j,k,m,\ \sum_{l=1}^{s_{j,k}} \delta_{j,k,l,m} \leq a_{j,k} \quad (41)$$

$$\forall j,k,l,m,\ \gamma_{j,k,m} \geq \delta_{j,k,l,m} \quad (42)$$

$$\forall j,k,\ \sum_{m=1}^{II_k^{UB}} \gamma_{j,k,m} \leq II_k \quad (43)$$

where $Cycle$, $Cycle_k$, a_j, $a_{j,k}$, II_k, $\delta_{j,k,l,m}$, $\gamma_{j,k,m}$ are integer variables, $Area$ is a rational variable, and $Area_k^{NS}$, $Area_j^S$, $Area_{Avail.}$, $s_{j,k}$, II_k^{Min}, TC_k, Dth_k are constant values. D is treated as a constant value and an outer loop is used to enumerate possible duplication factors for global optimal solution.

With such techniques, the problem formulation can be converted to a set of ILP optimizations. The number of binary variables introduced is less than the total number of shared components in the entire accelerator multiplied by the maximum loop II defined by Eq. (25).

5. IMPLEMENTATION ISSUES

In this section, we will discuss some implementation related issues about performance optimization with resource constraint in HLS.

5.1 Computing Load Information Extraction

The first implementation issue is how to extract the shareable component information from the source code. It is relatively easy to detect whether there is a certain type of components in a certain loop, but to get the exact number is nontrivial. Direct syntactic extraction may be inaccurate due to the optimizations by HLS tools such as common subexpression elimination. Implementing a customized lightweight HLS tool [18] is one approach, but it is time-consuming and difficult to match perfectly to the result of a commercial HLS tool.

In this section, we will demonstrate how to extract the precise shareable component information by profiling the transformed/instrumented input code using HLS tools.

We first preprocess the input code by removing loop-carried dependence with HLS directives. Then we eliminate array port conflicts by replacing each distinct array reference with a temporal volatile scalar. The transformed code will have a different function with the original code, but the shareable component information remains unchanged.

Then, for each type of shareable component, we add HLS resource constraint directives to instantiate one instance of the component and perform HLS on the preprocessed code to obtain loop II from the HLS report. In such cases, the number of shareable components in each loop is equal to the II of the loop.

5.2 Trip Count Calculation

The second implementation issue is how to obtain the trip count of each loop. For simple loops with constant bounds, trip count of the loop can be calculated and reported by HLS tools. For loops with non-constant but affine loop bounds,[1] trip count of the loop can be calculated by integer point counting [19] of iterator polytopes. For a general HLS-synthesizable program, trip counts can be collected by executing the input code that has been instrumented with trip count calculation statements. Program slicing [20] can be used to reduce the profiled program to containing only statements that impact the control-flow.

5.3 Parallelization and Resource Sharing between Loops

In the Vivado HLS [2], coarse-grained parallelization is explored at function-level instead of loop-level. Loop-level parallelization is not directly supported but can be achieved by capsuling the parallel loops into functions. However, for scalability consideration,

[1] The set of iterations of these loop bodies can be exactly captured using inequalities on the values of the loop iterators and program constants.

resource sharing in Vivado HLS is performed within a function. Resource sharing between functions is disabled. These strategies are incompatible with our assumptions that paralleled loops can still share resources with their common preceding or succeeding loops.

Our current implementation is to first capsule the parallel loops into functions automatically and then modify the RTL generated by HLS for resource sharing between modules.

6. EXPERIMENTAL RESULTS

In this section we present our experimental results using a set of computation kernels and applications. We first use Segmentation as case study to illustrate details of the proposed algorithm. Then, we show the performance improvements of our proposed optimization technique with extensive benchmarks.

Although our algorithm is applicable to both ASIC and FPGA designs, we chose FPGA as the target device in this work because of the availability of downstream behavioral synthesis and implementation tools. The Xilinx Virtex-7 XC7V585T FPGA device is selected as the target hardware platform. The Xilinx Vivado HLS Suite 2013.4 is used to perform the HLS and physical implementation flow. Techniques proposed in the paper are also be applicable to other high-level synthesis tools e.g. LegUp [21].

6.1 Segmentation: A Case Study

Table 5: Program Information of Segmentation

	$\pm$	*	/	$\sqrt{x}$	$\frac{1}{x}$	$>$	II_{LB}	TripCount
L1	3	1	1	0	0	0	1	32768
L2	6	3	0	0	0	0	2	29791
L3	5	7	1	1	1	1	1	32768
L4	3	6	0	0	0	0	2	29791
L5	16	2	1	0	1	2	5	32768

In this section, we use a more complicated benchmark Segmentation algorithm as a case study to demonstrate details in the proposed algorithm. Table 5 shows the basic information of the program. There are five loops and six types of shareable component candidates in the design. Some loops (L2, L4, L5) can not be fully pipelined because of loop-carried dependence or BRAM port constraint. The computation load vectors and trip count of each loop are specified in the table.

In Sec. 3, we assume that loop depths and the area of non-shareable component candidates will not change among different implementations. Table 6 is used to demonstrate the accuracy of the performance and area model against HLS results. From the table, we can observe that among the FPGA area tuple, DSP can always be precisely estimated. The error rate of lookup table (LUT) estimation is under 5%, while the error rate of flip-flop (FF) estimation can range up to 13.79%. The most possible reason is that pipeline registers will change significantly with loop IIs. However, in all the implementations of Segmentation and all other benchmarks used in the paper, FF will never be the scarcest resource. Therefore, as shown in the table, we can always model the number of replicas D with the proposed area model. The performance model is also very accurate, with a maximum error rate of 0.001% compared to RTL-level simulation by HLS tools.

Table 7 illustrates all the Pareto-optimal implementations of Segmentation with different IIs and resource sharing vectors. Performance numbers are normalized against the performance of the first implementation. In the first implementation, single-replica performance is maximized by minimizing the IIs of all loops. However, with the largest single-replica, the smallest number of replicas will be generated. With the increase of loop IIs, less shareable component candidates will be allocated. Area consumption of a single replica is reduced at the cost of worse single-replica performance. Among all 14 implementations, implementation 4 achieves the optimal tradeoff between performance and area consumption. It has the highest performance/area ratio and the largest throughput under the area constraint of the FPGA device after module duplication.

Table 6: Error Analysis of Area and Performance Model

#	LUT (%)	FF (%)	DSP (%)	Cycle (%)	Dup (%)
1	0	0	0	0	0
2	1.54	5.97	0	0	0
3	0.68	7.52	0	0	0
4	1.14	9.09	0	0.00042	0
5	1.14	7.78	0	0	0
6	2.27	6	0	0.00097	0
7	1.38	7.6	0	0.00089	0
8	1.87	9.26	0	0.00118	0
9	0.47	8.61	0	0.00055	0
10	4.02	8.53	0	0	0
11	2.69	10.3	0	0.00016	0
12	2.24	9.6	0	0.00048	0
13	0.41	13.79	0	0.00019	0
14	0.97	13.61	0	0	0

6.2 Benchmark Description

In addition to DWT and Segmentation, we use six other real-world computation kernels and applications to evaluate the proposed throughput optimization algorithm. FDTD, MM and Syrk are selected from PolyBench/C 3.2[22]. Deno, IMin and EMin are extracted from X-Ray CT image pipeline algorithms. A description of each benchmark can be found in Table 8. The number of loops (N), types of shareable component candidates (M) and total number of shareable component candidates in the entire code ($\sum s_{j,k}$) of each benchmark are enumerated in the table. Some loops in IMin and FDTD can be executed in parallel. Loops in other benchmarks must be executed in sequence due to inter-loop dependence. Double precision floating point is used as the default data type in computations, as in the original code. Due to the relative small-size of the benchmarks, branch-and-bound solutions are used in the experiments.

6.3 Experimental Results

Table 9 compiles the experimental results for all benchmarks. In the Baseline implementation, the smallest loop IIs are specified to achieve best performance for a single replica, then the replica is duplicated within the FPGA device capacity. We then enumerate all possible implementations with different IIs and allocated shareable component candidates to find the optimal solution with the best overall performance under the area constraint of the FPGA device. HLS directives **"#pragma HLS allocation instances = ComponentName limit=xx operation/ function"** and **"#pragma HLS pipeline II=yy"** are inserted to the program to guide Vivado HLS tool to achieve optimal resource allocation and pipeline II generated by our proposed algorithm. Then Xilinx Vivado Design Suite is used to synthesize the transformed code to bitstream and perform RTL-level simulation to generate all the performance

Table 7: Design Space Exploration for Image Segmentation

No.	Operators						Initiation Intervals					Area			Cycles	#(Replica)	Performance
	$\pm$	$*$	$/$	$\sqrt{x}$	$\frac{1}{x}$	$>$	L1	L2	L3	L4	L5	LUTs	FFs	DSPs			
1	5	7	1	1	1	1	1	2	1	2	5	16143	12801	106	348783	11	1
2	4	4	1	1	1	1	1	2	2	2	5	14983	11012	70	381551	18	1.496
3	4	3	1	1	1	1	1	2	3	2	5	14649	10640	59	414319	21	1.607
4	4	2	1	1	1	1	1	2	4	3	5	14512	10277	48	476876	25	1.662
5	4	1	1	1	1	1	1	3	7	6	5	14307	10189	37	694346	25	1.142
6	3	4	1	1	1	1	1	2	2	2	6	14296	10589	67	414315	18	1.378
7	3	3	1	1	1	1	1	2	3	2	6	13962	10218	56	447083	22	1.56
8	3	2	1	1	1	1	1	2	4	3	6	13825	9854	45	509640	26	1.618
9	3	1	1	1	1	1	1	3	7	6	6	13300	9702	34	727110	27	1.177
10	2	3	1	1	1	1	2	3	3	2	8	13532	9721	53	575182	23	1.268
11	2	2	1	1	1	1	2	3	4	3	8	13139	9357	42	637742	27	1.342
12	2	1	1	1	1	1	2	6	7	3	8	12870	9208	31	825422	28	1.076
13	1	2	1	1	1	1	3	6	5	3	16	11955	8679	39	1054796	30	0.902
14	1	1	1	1	1	1	3	6	7	6	16	11688	8491	28	1209703	31	0.813

Table 8: Benchmark Description

Name	Description	N	M	$\sum s_{j,k}$	Dep
Segmentation	Image Segmentation	5	6	61	Sequential
Denoise	Rician Noise Removal	2	6	56	Sequential
DWT	Discrete Wavelet Transform	4	2	10	Sequential
Image Minimization	X-Ray Image Minimization	10	2	47	Parallel
Edge Minimization	X-Ray Edge Minimization	3	4	37	Sequential
FDTD-2D	Finite Diff. Time Domain	4	2	11	Parallel
Matrix Multiplication	Matrix Multiplication	3	2	6	Sequential
Syrk	Symmetric Rank-k Operations	2	2	6	Sequential

numbers. The entire experiment was implemented by a push-button flow shown in Figure 5.

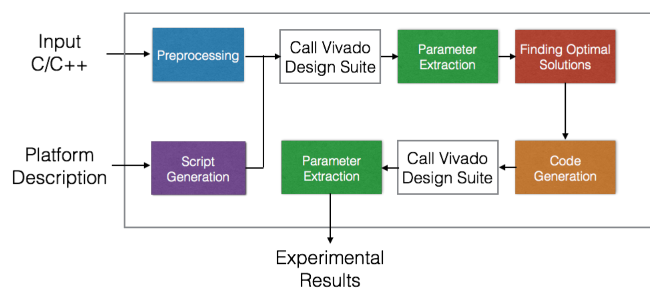

Figure 5: Experimental Flow

Compared to the base implementation, the speedup of optimal solution found by the proposed design space exploration algorithm ranges from 1X (Syrk) to 1.85X (IMin), with an average of 1.31X and the geometric mean as 1.28X, shown in Fig. 6. In Syrk, the traditional optimize-and-duplicate approach already achieves the optimal solution in terms of performance-area density.

The loop initiation intervals of the baseline and optimal implementations are shown in Table 10. From the table, we can see that except for Syrk, in all other benchmark designs, higher IIs in some loops will better overall throughput after exploiting coarse-grained performance through duplication. Considering the moderate scale of the benchmark applications, branch-and-bound solution is selected in the experiment. The runtime of the entire optimization algorithm is from 12.9s to 103s, as shown in Table 10.

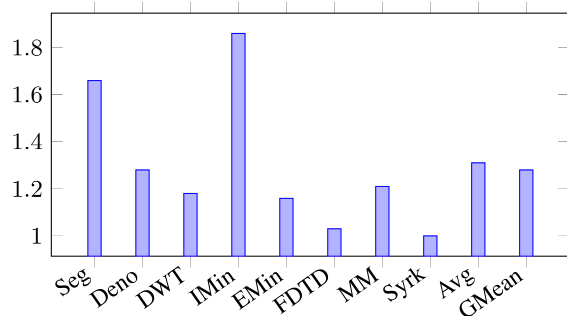

Figure 6: Performance Speedup

In all benchmarks used in this paper, the outer-most loop is dependence-free. Another approach is to implement the designs with pipelined outer-most loop, which could be an interesting future work.

7. CONCLUSIONS

In this paper, We formulate an area-aware throughput optimization problem by simultaneously addressing the concept of loop pipelining, module selection, duplication and resource sharing within a unified framework. We propose some methods to solve the problem efficiently. Experimental results show that an average of 31% speedup can be achieved compared to previous results.

Table 9: Experimental Results

Benchmark	Implemtation	LUT	FF	DSP	CP(ns)	Cycles	#(Replicas)	Performance
Seg	Baseline	16143	12801	106	8.8	348783	11	1
	Optimized	14512	10277	48	8.821	476876	25	1.66
	Comparison(%)	-10.1	-19.7	-54.7	0.2	36.7	127.3	66.2
Deno	Baseline	15330	11590	85	8.171	28849	14	1
	Optimized	12516	9514	54	8.194	37047	23	1.28
	Comparison(%)	-18.4	-17.9	-36.5	0.3	28.4	64.3	27.9
DWT	Baseline	2855	1985	28	8.253	590337	45	1
	Optimized	2716	1830	17	8.091	851457	74	1.14
	Comparison(%)	-4.9	-7.8	-39.3	-2	44.2	64.4	14
IMin	Baseline	8969	6691	56	8.701	38196620	22	1
	Optimized	5437	4387	14	8.562	61789701	66	1.85
	Comparison(%)	-39.4	-34.4	-75	-1.6	61.8	200	85.5
EMin	Baseline	10496	8112	42	8.27	64882073	30	1
	Optimized	9452	7366	28	8.183	70780403	38	1.16
	Comparison(%)	-9.9	-9.2	-33.3	-1.1	9.1	26.7	16.1
FDTD	Baseline	4451	2737	23	8.125	7485620	54	1
	Optimized	2907	1852	17	8.214	9975630	74	1.03
	Comparison(%)	-34.7	-32.3	-26.1	1.1	33.3	37	2.8
MM	Baseline	2276	1744	25	8.078	67620	50	1
	Optimized	2097	1471	14	7.982	100387	90	1.21
	Comparison(%)	-7.9	-15.7	-44	-1.2	48.5	80	21.2
Syrk	Baseline	1874	1553	25	8.171	2113560	50	1
	Optimized	1874	1553	25	8.171	2113560	50	1
	Comparison(%)	0	0	0	0	0	0	0

Table 10: Baseline and Optimal Loop IIs

Bmk .	Baseline	Optimal	Runtime
Seg	$\langle 1,2,1,2,5\rangle$	$\langle 1,2,4,3,5\rangle$	103s
Deno	$\langle 4,3\rangle$	$\langle 4,5\rangle$	15.5s
DWT	$\langle 1,1,1,1\rangle$	$\langle 1,1,2,2\rangle$	12.9s
IMin	$\langle 3,1,5,1,1,$ $5,5,5,1,1,\rangle$	$\langle 12,1,5,1,1$ $5,5,5,1,1,\rangle$	23.7s
EMin	$\langle 5,5,1\rangle$	$\langle 6,5,1\rangle$	14.0s
FDTD	$\langle 1,1,1,1\rangle$	$\langle 1,1,1,2\rangle$	16.4s
MM	$\langle 1,1,1,1\rangle$	$\langle 1,2,1,1\rangle$	16.7s
Syrk	$\langle 1,1\rangle$	$\langle 1,1\rangle$	14.5s

Acknowledgment. This work was supported in part by the National Science Foundation through awards 0926127 and 1321147; and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

8. REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[2] "Vivado HLS," http://www.xilinx.com/products/design-tools/vivado/index.htm.

[3] Z. Zhang, Y. Fan, W. Jiang, G. han, and J. Cong, "AutoPilot: a platform-based esl synthesis system," in *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[4] B. D. T. Inc., "An independent evaluation of: High-level synthesis tools for xilinx FPGAs," http://www.bdti.com/MyBDTI/pubs/Xilinx_hlstcp.pdf, 2010.

[5] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 2, pp. 15:1–15:25, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1929943.1929947

[6] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 8, no. 6, pp. 661–679, Nov. 2006.

[7] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, no. 3, pp. 355–383, Jul. 2003.

[8] I. Ahmad, M. K. Dhodhi, and C.-Y. Chen, "Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis," in *Computers and Digital Techniques, IEE Proceedings-*, vol. 142, no. 1. IET, 1995, pp. 65–71.

[9] J. Cong and J. Xu, "Simultaneous FU and register binding based on network flow method," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 1057–1062.

[10] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 111–114. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145712

[11] D. Chen, J. Cong, Y. Fan, and J. Xu, "Optimality study of resource binding with multi-vdds," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 580–585.

[12] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah, "A computing origami: Folding streams in FPGAs," in

Proceedings of the 46th Annual Design Automation Conference, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 282–287. [Online]. Available: http://doi.acm.org/10.1145/1629911.1629987

[13] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou, "Combining module selection and replication for throughput-driven streaming programs," in *DATE*, 2012, pp. 1018–1023.

[14] J. Cong, M. Huang, and P. Zhang, "Combining computation and communication optimizations in system synthesis for streaming applications," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, 2014, pp. 213–222.

[15] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 6, pp. 975–1028, Nov. 2004.

[16] D. Chen, J. Cong, and Y. Fan, "Low-power high-level synthesis for FPGA architectures," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, ser. ISLPED '03. New York, NY, USA: ACM, 2003, pp. 134–139.

[17] A. Aiken and A. Nicolau, "Perfect pipelining: A new loop parallelization technique." in *ESOP*, ser. Lecture Notes in Computer Science, H. Ganzinger, Ed., vol. 300. Springer, 1988, pp. 221–235. [Online]. Available: http://dblp.uni-trier.de/db/conf/esop/esop88.html#AikenN88

[18] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[19] A. Barvinok, *Integer Points in Polyhedra*. European Mathematical Society, 2008.

[20] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[21] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[22] Polybench3.2, http://polybench.sourceforge.net. [Online]. Available: http://polybench.sourceforge.net