Optimal Partition with Block-Level Parallelization in C-to-RTL Synthesis for Streaming Applications

Shuangchen Li * Yongpan Liu* X.Sharon Hu† Xinyu He* Yining Zhang* Pei Zhang‡ Huazhong Yang*
Tsinghua National Laboratory for Information Science and Technology,
Dept. of Electronic Engineering, Tsinghua University, Beijing, 100084, China*
Dept. of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, U.S.†
Y Explorations Inc. San Jose, CA 95134, U.S.‡
{ypliu,yanghz}@tsinghua.edu.cn*, shu@nd.edu†, zhangpei@gmail.com‡

***Abstract*—Developing FPGA solutions for streaming applications written in C (or its variants) can benefit greatly from automatic C-to-RTL (C2RTL) synthesis. Yet, the complexity and stringent throughput/cost constraints of such applications are rather challenging for existing C2RTL synthesis tools. This paper considers automatic partition and block-level parallelization to address these challenges. An MILP-based approach is introduced for finding an optimal partition of a given program into blocks while allowing block-level parallelization. In order to handle extremely large problem instances, a heuristic algorithm is also discussed. Experimental results based on seven well known multimedia applications demonstrate the effectiveness of both solutions.**

I. Introduction

Domain-specific FPGA accelerators are gaining popularity in heterogeneous multiprocessor systems-on-chip (MPSoCs) [1], [2]. Such accelerators have the potential to provide orders of magnitude improvements in both energy consumption and execution time. Most of the domain-specific applications are developed in the C language (or its variants), instead of hardware description languages (HDL), such as VHDL and Verilog. Furthermore, decades of developing C-language based embedded applications have accumulated plenty of legacy code in various application domains, e.g., in image processing, almost all programs are developed in C or C++. It can be extremely beneficial if C-based application programs can be used directly to synthesize domain-specific accelerators for heterogeneous MPSoCs.

The increasing interests in high quality C-to-RTL (C2RTL) transformation [3] have lead to the development of a number of commercial and academic C2RTL tools (e.g., [4], [5], [6]). Most of these tools can synthesize moderate-sized C programs (e.g., with hundreds lines) quite efficiently. In fact, various domain-specific accelerators, e.g., 3G/4G wireless communication [7], digital video broadcasting [8], face detection [9], have been developed using existing C2RTL tools (e.g., Catapult-C, CoDeveloper, CWB). However, existing C2RTL tools face significant challenges in terms of synthesis result quality and running time for large-sized C programs. Furthermore, no existing tools can adequately handle user given constraints, such as minimize area subject to a throughput constraint. Depending on the application domains, different approaches may be adopted to tackle these challenges.

In this paper, we focus on C2RTL synthesis of large C (or its variants) programs for streaming applications. Such applications generally have stringent throughput and area requirements. To satisfy such requirements and address the scalability challenge, an effective approach is to partition a large C code into smaller entities to be synthesized individually using existing C2RTL tools and then connect the synthesized modules by certain interface structures (e.g., as discussed [10]). We refer to this strategy as C-code partition (or simply partition). In order to meet throughput constraints, the synthesized modules may also be duplicated multiple times, which is referred to here as block-level parallelization (or simply parallelization).

This work was supported in part by the NSFC under grant 61204032 and 61271269, National Science and Technology Major Project under contract 2010ZX03006-003-01, eXCite software donated from Y Explorations Inc., and High-Tech Research and Development (863) Program under contract 2013AA010101.

Some existing work has examined techniques for partition or parallelization (e.g., [11], [12], [13], [14]). In [11], stream graphs extracted from *StreamIt* programs are used as input and parallelization degrees are determined for the nodes in the graph. A heuristic algorithm is presented which refines a given stream graph by deciding the number of times each node in the graph to be replicated. The authors assume, however, that the partition is given. In [12], an integer linear programming (ILP) based method is proposed to determine which modules are to be included in the final design as well as the parallelization degrees of the selected modules. The method aims at minimizing area while satisfying the throughput constraint. The authors of [13] present a heuristic technique to find the parallel degrees of C-code blocks obtained. Though these methods are effective in performing parallelization, the partitioning part is assumed to be done a priori. Hara [14] et.al. introduce an ILP based method to cluster small C functions into blocks under performance and area constraints. However, their method does not consider block-level parallelization. In summery, none of the existing synthesis work can adequately explore the design space defined by both C-code partition and block-level parallelization.

A somewhat related line of work is mapping C programs to MPSoCs, where partitioning and parallelization may also be considered (e.g., [15], [16], [17], [18]). In software mapping, multiple blocks (or tasks) can be assigned to the same processor and the processor area is given. This is rather different from the hardware synthesis work considered here, thus the methods are not readily applicable.

In this paper, we present a hierarchical C2RTL synthesis framework for developing accelerators for complex streaming applications. This framework can simultaneously consider C-code partition and block-level parallelization, and synthesize high-quality circuits satisfying given area or throughput constraints. To our best knowledge, this is the first approach with such capabilities. Our specific contributions include

- introducing a novel mixed ILP (MILP) based formulation for finding a partition and parallelization solution with maximum throughput or minimum area while satisfying a given area or throughput constraint, respectively,
- proposing an efficient heuristic algorithm to deal with the scalability challenge facing the MILP formulation, and
- validating the proposed methods through developing FPGA-based accelerators for multiple streaming applications.

II. Overview

In this section, we describe our motivation and proposed hierarchical C2RTL framework supporting automatic C-code partition and block-level parallelization. We then introduce the system model used in our optimization methods.

A. Motivation

Traditional C2RTL approaches (e.g., [8], [19], [20]) directly transform an entire C program into one large flat HDL module. Such approaches have three main shortcomings. (i) The quality (e.g., throughput and area) of the synthesized circuits for larger C programs is usually low. (ii) Run time of the C2RTL tools can be intolerably long or they may even fail when a C program reaches thousands of lines. (iii) Optimization options are limited and hence it is difficult to optimize a design under general throughput/area constraints.

Unlike conventional, flat approaches, a hierarchical design flow provides the capability of exploring a larger and more complex design space. It partitions a large program into a number of smaller blocks and synthesizes them separately. The synthesized blocks (i.e., modules) are then connected by appropriate interface structures, e.g., FIFOs. Such an approach provides both flexible architectures and high-quality modules. A 10× speedup [10] can be obtained using a hierarchical flow compared to a flat one.

In a hierarchical design flow, C-code partition can greatly impact the quality of the synthesized results and is an important step for trading off throughput with area [10]. From the performance point of view, since the performance of streaming applications is typically limited by the slowest module, a partition leading to comparable speed for all modules seems to be the most desirable one. However, reducing area requires more functions be integrated into one big (but slow) block so as to maximize the possibility of resource sharing.

Besides C-code partition, block-level parallelization provides another design knob in the hierarchical flow. It duplicates certain blocks to improve throughput with tolerable area overhead. Parallelization is especially effective in boosting the performance of the circuits whose throughput is limited by certain stages. Results in [13] indicate that block-level parallelization can lead to much higher performance but with nontrivial area overhead.

As different partitions may lead to different parallelization choices, it is necessary to **simultaneously** consider both in order to find the optimal solutions in the hierarchical design flow. If considered separately, it is difficult to set proper constraints for the optimization problem to be solved first from the entire system's constraints. In addition, simultaneously considering partition and parallelization allows exploration of the whole design space to avoid being stuck at a local optimal solution.

B. Proposed framework

Below we give a high-level description of our hierarchical flow as shown in Figure 1. The input is the C program containing N functions connected in a straightline fashion, i.e., the output of the i-th function is the input of the (i+1)-th function. Such input format is common in streaming applications to be implemented in hardware [21]. Note that feedback and branches can be included within functions and global variables can be handled by code modification. Throughput/area constraints are also given as input. The framework consists of four main steps. In Step 1, the timing and area data are generated for each function in the C program by using a C2RTL tool (*eXCite* [5] is used in our current implementation). Based on the user-specified performance and area constraints, Step 2 determines the optimal partition and parallelization by clustering functions into blocks and duplicating certain blocks. Then, the blocks are synthesized into HDL modules in Step 3. Step 4 assembles the duplicated modules according to the parallelization degree into PEs and connects them with FIFOs to get the final system.

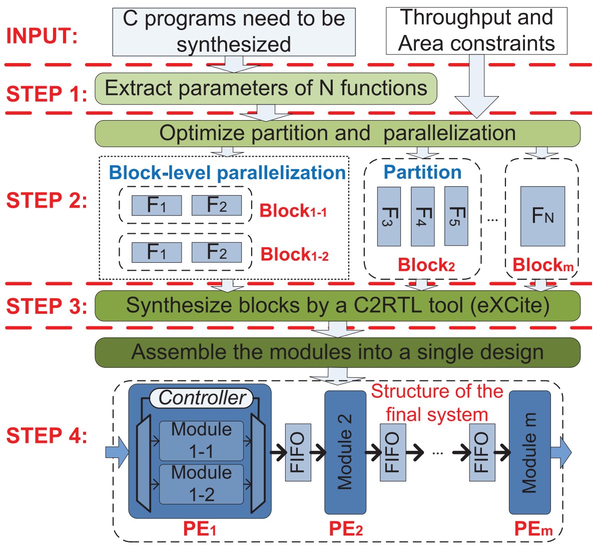
INPUT:
C programs need to be synthesized
Throughput and Area constraints
STEP 1:
Extract parameters of N functions
Optimize partition and parallelization
STEP 2:
Block-level parallelization
F1
F2
Block1-1
F1
F2
Block1-2
Partition
F3
F4
F5
...
Block2
FN
Blockm
STEP 3:
Synthesize blocks by a C2RTL tool (eXCite)
Assemble the modules into a single design
STEP 4:
Controller
Module 1-1
Module 1-2
FIFO
Module 2
FIFO
...
FIFO
Module m
Structure of the final system
PE1
PE2
PEm

Fig. 1. Proposed design flow.

In the input stage, our framework can take the finest granularity to be a function. For typical C programs, each function usually contains less than 200 lines [22]. Since existing C2RTL tools can synthesize high-quality circuits for C code at this scale, C programs at the function-level granularity can be directly used as the input to our framework. For larger functions, the techniques introduced in [16] can be employed to automatically divide the functions into smaller ones. The partition/parallelization stage (Step 2) presents one of the main technical challenges to be tackled by this paper, which will be discussed in detail in Section III and IV. The assembling stage (Step 4) implements block-level parallelization and decides the FIFO sizes. We realize the block-level parallelization by a MUX, a DEMUX and a controller as shown in the leftmost shaded area in Step 4 of Figure 1, where PE_1 contains two parallel modules synthesized from Block_1. The details of Step 3 and 4 are beyond the scope of this paper.

C. System modeling

To support the exploration of the large partition and parallelization space, we need to carefully model a given C program. Our system model focuses on characterizing the throughput and area parameters of functions, blocks and the system.

1) Function parameters: We use F_n to denote the n-th function, which contains three parts: input, processor, and output (a common practice in hardware design, otherwise, a wrapper can be added). The period of each function is represented by T_n (cycles), which includes the input latency T_n^{in} to load S_n^{in} byte of data, the processing time, and the output latency T_n^{out} to write S_n^{out} data. The area for function F_n is split into the logic part A_n^{le} and the memory one A_n^{mem}. We summarize the function-level performance and area parameters in Table I (rows marked by F_n). Given a C function, these parameters can be obtained by a C2RTL synthesis tool (such as *eXCite*).

2) Block parameters: Blocks are formed by clusters of neighboring functions, F_i is the neighboring function of F_j if F_i's output is F_j's input or vise versa. We denote a block as $B_{i,j}$, which consists of adjacent functions of F_i, F_{i+1}, $\cdots$, F_j. Block-level parameters are similar to the function-level ones, and are shown in Table I (rows marked by $B_{i,j}$). The latency $T_{i,j}$ of block $B_{i,j}$ is computed by

$$T_{i,j} = T_i^{in} + T_j^{out} + \sum_{n=i}^{n=j} (T_n - T_n^{in} - T_n^{out}) \tag{1}$$

That is, $T_{i,j}$ equals the sum of the latency of each function in $B_{i,j}$. Note that the latency of data input and output of the internal functions is not included because the functions in $B_{i,j}$ will be synthesized into one module. (2) shows the input and output latency $T_{i,j}^{\text{in}}$/$T_{i,j}^{\text{out}}$ and data size $S_{i,j}^{\text{in}}$/$S_{i,j}^{\text{out}}$ of block $B_{i,j}$. They remain the same as those of the first and the last function (i.e., F_i and F_j) in block $B_{i,j}$.

$$T_{i,j}^{in} = T_i^{in} \qquad T_{i,j}^{out} = T_j^{out} \qquad S_{i,j}^{in} = S_i^{in} \qquad S_{i,j}^{out} = S_j^{out} \tag{2}$$

The logic element area, $A_{i,j}^{\text{le}}$, is calculated as (3), where α^{le} is the area saving factor to reflect resource sharing when clustering more than one functions into one block.[1] Since functions in one block work serially, they can share one memory block without conflicts. That is, the function with the largest memory area determines $A_{i,j}^{\text{mem}}$.

$$A_{i,j}^{le} = \begin{cases} \sum_{n=i}^{n=j} A_n^{le} \cdot (1 - \alpha^{le}) & i < j \\ A_i^{le} & i = j \end{cases} \tag{3}$$

$$A_{i,j}^{mem} = \max A_k^{mem} \qquad \forall k \in [i,j] \tag{4}$$

Last but not least, we introduce an upper bound on the parallelization degree of block $B_{i,j}$, and denote it by $P_{i,j}$. The upper bound exists because multiple modules of a block sharing one I/O interface in each PE and have to read or write in a serial manner (see PE$_1$ in STEP 4 in Figure 1). Note that the upper bound is also affected by area resource, but we put this consideration into the entire area constraint in (13). $P_{i,j}$ here depends on the relationship between the input/output latency, $T_{i,j}^{\text{in}}$/$T_{i,j}^{\text{out}}$, and the total latency, $T_{i,j}$, of block $B_{i,j}$, and is computed by (5).

$$P_{i,j} = \lceil T_{i,j} / \max\{T_{i,j}^{in}, T_{i,j}^{out}\} \rceil \tag{5}$$

3) System parameters: As indicated earlier, designing accelerators for streaming applications often require the hardware to satisfy certain throughput and area constraints. Here, we use R_{req} and $A_{\text{req}}^{\text{le}}$/$A_{\text{req}}^{\text{mem}}$ to represent the system's design constraints for throughput and area. Parameters O^{le}/O^{mem} and $A_{\text{fifo}}^{\text{le}}$/$A_{\text{fifo}}^{\text{mem}}$ are the area overhead for realizing block-level parallelization and inserting FIFOs, respectively. These parameters are summarized in Table I (rows marked by system).

[1]For C programs from the same application domain, α^{le} usually share some similarity, which can be estimated by data fitting based on a selective set of synthesis results.

TABLE I
PARAMETERS FOR F_n, $B_{i,j}$, AND SYSTEM

	Parameters	Description
F_n	N	Total number of functions
	T_n	Latency of function F_n (*cycle*)
	T_n^{in}/T_n^{out}	Input/output latency of F_n (*cycle*)
	S_n^{in}/S_n^{out}	Input/output data size for F_n (*byte*)
	A_n^{le}/A_n^{mem}	Area of logic elements (*numbers*)/memory (*bits*) for F_n
$B_{i,j}$	$T_{i,j}$	Latency of block $B_{i,j}$ (*cycle*)
	$T_{i,j}^{\text{in}}$/$T_{i,j}^{\text{out}}$	Input/output latency of $B_{i,j}$ (*cycle*)
	$S_{i,j}^{\text{in}}$/$S_{i,j}^{\text{out}}$	Input/output data size for $B_{i,j}$ (*cycle*)
	$A_{i,j}^{\text{le}}$/$A_{i,j}^{\text{mem}}$	Area of logic element/memory for $B_{i,j}$
	α^{le}	Area saving factor when functions clustered into block
	$P_{i,j}$	Parallelization upper boundary of $B_{i,j}$
Sys-tem	R_{req}	Throughput constraint (*byte/cycle*)
	$A_{\text{req}}^{\text{le}}$/$A_{\text{req}}^{\text{mem}}$	Area constraint of logic element/memory
	A_{req}	Normalized area constraint ($A_{\text{req}}^{\text{le}}$+$\beta \cdot A_{\text{req}}^{\text{mem}}$)
	O^{le}/O^{mem}	Area overhead when duplicated
	$A_{\text{fifo}}^{\text{le}}$/$A_{\text{fifo}}^{\text{mem}}$	Area of logic elements/memory by every byte in FIFO

III. MILP-BASED SOLUTION

In this section, we describe our MILP-based method for C-code partition and block-level parallelization. The method **simultaneously** determines which functions are to be clustered in which block as well as the parallelization degree of each block. The goal is to either maximize the throughput or minimize the area while satisfying the area and/or throughput constraints.

A. Variable definitions

Variables play a critical role in making a mathematical programming formulation to be linear. Below, we discuss in detail the variables used in our MILP formulation. (Note that we use lower case letters for variables and upper case letters for parameters/constants whenever appropriate.) Variable $\{x_n\}$ (the set of all x_n's) represents the partition and parallelization and is the final output of our solution. x_n equals to zero if function F_n is clustered with one or more functions following it to form a block. Otherwise, the non-zero value of x_n indicates the parallelization degree of the block ending with F_n. For the example partition and parallelization in Figure 1, we have

$$\{x_n\} = \{\underbrace{0, 2}_{Block_1}, \underbrace{0, 0, 1}_{Block_2}, \cdots, \underbrace{1}_{Block_m}\} \tag{6}$$

Here, F_1 and F_2 are clustered into one block. Thus x_1=0 and x_2=2 indicate that F_2 is the last function of the first block, and that the block has a parallelization degree of 2. Furthermore, F_3, F_4, F_5 are clustered in the second block with no duplication, which makes x_5=1.

To help derive our MILP formulation, we introduce a zero-one variable, $y_{i,j}$. If $y_{i,j}$ equals to one, $F_i, F_{i+1}, \cdots, F_j$ are merged to form block $B_{i,j}$. Otherwise, no block is made up by functions F_i to F_j. For the example in Figure 1, we have:

$$\{y_{i,j}\} = \left\{ \begin{array}{ccccccc} 0, & 1, & 0, & 0, & 0, & \cdots, & 0 \\ & 0, & 0, & 0, & 0, & \cdots, & 0 \\ & & 0, & 0, & 1, & \cdots, & 0 \\ & & & 0, & 0, & \cdots, & 0 \\ & & & & 0, & \cdots, & 0 \\ & & & & & & \cdots \\ & & & & & & 1 \end{array} \right\} \tag{7}$$

Here, the first block is composed of F_1 and F_2, thus $y_{1,2}$=1. F_3 to F_5 forms the second block so $y_{3,5}$=1. The last block consists of only F_N, hence $y_{N,N}$=1. All other $y_{i,j}$'s are set to zero. We further introduce another auxiliary variable $z_{i,j}$ with $z_{i,j}$=$x_j \cdot y_{i,j}$, which indicates the parallelization degree of block $B_{i,j}$.

We use $r_{i,j}$ to represent the throughput of block $B_{i,j}$ after parallelization and r_{all} to represent the final system's throughput after partition and parallelization. Variable $a_{\text{fifo}}^{\text{le}}$/$a_{\text{fifo}}^{\text{mem}}$ represents the entire FIFO area overhead for connecting the blocks into a whole system, and $a_{\text{all}}^{\text{le}}$/$a_{\text{all}}^{\text{mem}}$ represents the final system's area. All the variables are listed in Table II.

TABLE II
VARIABLES USED IN THE MILP FORMULATION

Variables	Description
x_n	Presenting partition and parallelization information
$y_{i,j}$	If $y_{i,j}$=1, functions F_i to F_j are clustered into one block
$z_{i,j}$	Temporary variable representing ($x_j \cdot y_{i,j}$)
$r_{i,j}$	Throughput of $B_{i,j}$ after parallelization (*byte/cycle*)
r_{all}	Throughput of the system (*byte/cycle*)
$a_{\text{fifo}}^{\text{le}}$/$a_{\text{fifo}}^{\text{mem}}$	Area of logic elements/memory used by all the FIFOs
$a_{\text{all}}^{\text{le}}$/$a_{\text{all}}^{\text{mem}}$	Area of logic elements/memory used by system with FIFO

B. MILP formulation

The MILP formulation is to either maximize the overall throughput or minimize the area subject to area or throughput constraints. Let the throughput and area constraints be represented by R_{req} and A_{req}, respectively. If we assume that the areas of the memory and logic elements are related by scaling factor β, we have

$$a_{all} = a_{all}^{le} + \beta \cdot a_{all}^{mem} \qquad A_{req} = A_{req}^{le} + \beta \cdot A_{req}^{mem} \tag{8}$$

Then, the MILP formulation can be written as

$$\begin{aligned} &obj: \quad && \max r_{all} \quad \text{or} \quad \min a_{all} \\ &s.t.: && a_{all} \le A_{req} \quad \text{and} \quad r_{all} \ge R_{req} \end{aligned} \tag{9}$$

Besides the throughput and area constraints, the variables must also satisfy connectivity constraints. Below we describe each type of constraints in detail.

*1) **Throughput constraints**:* The throughput is decided by the slowest block. Note that block $B_{i,j}$ exists only when $y_{i,j}$=1. Hence, we have $r_{\text{all}}=\min_{y_{\text{i,j}}=1}\{r_{\text{i,j}}\}$. We can rewrite the expression in a linear form as follows,

$$r_{all} \le r_{i,j} + M \cdot (1 - y_{i,j}) \qquad \forall 1 \le i \le j \le N \tag{10}$$

where M is a very large constant (larger than any possible values in the MILP formulation). If $B_{i,j}$ exists in the system (i.e., $y_{i,j}$=1), $r_{i,j}$ is a linear function of its parallelization degree x_j, unless the parallelization upper bound is reached, in which case $r_{i,j}$ is limited by $T_{i,j}^{\text{in}}/T_{i,j}^{\text{out}}$. Hence, we have (11), which can be readily re-written in the linear form shown as (12).

$$\textit{Original}: \qquad r_{i,j} = \begin{cases} (x_j \cdot y_{i,j})/T_{i,j} & (x_j \cdot y_{i,j}) < P_{i,j} \\ 1/\max\{T_{i,j}^{in}, T_{i,j}^{out}\} & \text{otherwise} \end{cases} \tag{11}$$

$$\textit{Linear form}: \qquad r_{i,j} \le \begin{cases} z_{i,j}/T_{i,j} \\ 1/\max\{T_{i,j}^{in}, T_{i,j}^{out}\} \end{cases} \tag{12}$$

*2) **Area constraints**:* To compute the total area, we should consider together the areas of the FIFOs and the blocks after parallelization. When duplicating blocks, an area overhead due to the inclusion of a MUX and etc. (see STEP 4 in Figure 1) must be taken into account. Therefore, we have (13), which can be made linear by using auxiliary variable $z_{i,j}$=$x_j \cdot y_{i,j}$.

$$a_{\text{all}}^{\text{le/mem}} = a_{\text{fifo}}^{\text{le/mem}} + \sum_{i=1}^{i=N} \sum_{j=i}^{j=N} ((x_j - 1) \cdot O^{\text{le/mem}} + x_j \cdot A_{i,j}^{\text{le/mem}}) \cdot y_{i,j} \tag{13}$$

As to the FIFO area overhead, it is a function of the FIFO capacity which can be estimated by the output data size ($z_{i,j} \cdot S_j^{\text{out}}$). For a more accurate FIFO size, the method in [10] can be used. Then we have the FIFO area overhead as

$$a_{\text{fifo}}^{\text{le/mem}} = A_{\text{fifo}}^{\text{le/mem}} \cdot \sum_{i=1}^{i=N-1} \sum_{j=i}^{j=N} z_{i,j} \cdot S_j^{\text{out}} \tag{14}$$

*3) **Connectivity constraints**:* Variable x_n is greater than or equal to one only if F_n is the last function in a block. Otherwise, x_n is forced to zero. (15) captures the above constraint. According to the relationship between adjacent blocks, F_{j-1} must be the last function of a block if F_j is the first function of a block. Therefore, $y_{i,j}$ should also satisfy (16). Since one function can only be present in one block, we have (17). Lastly we have a constraint for the total number of blocks in the system in (18).

$$\sum_{i=1}^{i=n} y_{i,n} \le x_n \le M \cdot \sum_{i=1}^{i=n} y_{i,n} \qquad x_n \in \mathbf{N} \tag{15}$$

$$\sum_{i=1}^{i=j-1} y_{i,j-1} = \sum_{i=j}^{i=N} y_{j,i} \qquad \forall j \in [2, N] \tag{16}$$

$$\sum_{i=1}^{i=j} y_{i,j} + \sum_{i=j}^{i=N} y_{j,i} - y_{j,j} \le 1 \qquad \forall j \in [1, N] \tag{17}$$

$$1 \le \sum_{j=1}^{j=N} \sum_{i=1}^{i=j} y_{i,j} \le N \tag{18}$$

As seen earlier, we have introduced an auxiliary variable $z_{i,j}$=$x_j \cdot y_{i,j}$ to help the derivation of the MILP formulation. The multiplicative expression for $z_{i,j}$ can be re-written to linear form as follows.

$$\begin{aligned} &-M \cdot y_{i,j} \le z_{i,j} \le M \cdot y_{i,j} \\ &x_j - M \cdot (1 - y_{i,j}) \le z_{i,j} \le x_j + M \cdot (1 - y_{i,j}) \end{aligned} \tag{19}$$

The entire MILP formulation consists of expressions (9), (10), (12) to (19). Additionally, we require x_n's, $z_{i,j}$'s be integers and $y_{i,j}$'s be zero-one variables. We briefly discuss the complexity of the MILP problem. For a C program with N functions, there are $\frac{3}{2}N^2 + \frac{5}{2}N + 2$ variables and $5N^2 + 7N + 2$ constraints. Therefore, the constraint matrix of the MILP instance has a dimension of $O(N^4)$.

IV. Heuristic solution

It is well known that MILP based methods are not scalable. For our MILP algorithm, experimental results show that it can be time consuming if the problem size is very large. Furthermore, it can also fail to find solutions when the constraints are *bad* (corresponding to extremely *un-smooth* feasible regions) even for relatively small problem sizes. In this section, we propose a heuristic algorithm to solve the partition and parallelization problem.

We make use of the following observation in developing our heuristic algorithm. When considered separately, C-code partition, which clusters functions to form blocks thus allows more resource sharing, always reduces area but cannot increase throughput, while block-level parallelization increases throughput at the cost of area. Hence, there is a monotonicity property if one considers partition and parallelization separately. Our heuristic exploits this property by performing partition and parallelization in an iterative manner.

The algorithm for the case when $\max r_{\text{all}}$ is the objective is shown in Algorithm 1. After the necessary initialization (Lines 1-2), the algorithm improves the throughput by incrementally increasing the parallelization degrees of the bottleneck function using procedure *Incx* until the throughput constraint is met (Lines 3-6). Then we do partition in an iterative manner (Lines 7-17) and evaluate the system area. Based on the parallelization degree obtained so far, *Clst* finds the optimal partition not violating r_{all} (Line 8). In Line 10, if the area constraint is violated, the parallelization degree should be decreased. This is achieved by simply using the last $\{x'_n\}$ from the stack. (Note that a stack is used to maintain $\{x'_n\}$ and r_{all}.) A stack empty error indicates no solution under the give constraints. If the area constraint is met, we store current $\{x'_n\}$ and r_{all} on the stack and call *Incx* to continue increasing the throughput (Lines 14-15). The loop is terminated when partitions with the same $\{x'_n\}$ have been explored before (Line 13). If $\min a_{\text{all}}$ is the objective, we can still use Algorithm 1 while skipping Lines 10-17.

The heuristic algorithm relies on two main procedures, i.e., the parallelization procedure, *Incx* and the partition procedure, *Clst*. *Incx* simply searches for the slowest function, increments its parallelization degree by one, and calculates r_{all} by (10). In the *Clst* procedure, we construct a partition solution ($\{x_n\}$) that minimizes the area while not reducing the throughput given (i.e. $\{r_{\text{all}}\}$). This is achieved by performing a sequential scan of all the possible partitions based on previous parallelization ($\{x'_n\}$). Due to the page limit, we omit the details of *Incx* and *Clst*, which can be found in [23]. We will investigate the effectiveness of the heuristic from the experimental results in Section V.

V. Experimental Evaluation

In this section, we present our experimental results based on seven realistic streaming programs.

Algorithm 1: Partition and parallelization to maximize r_{all}

```
input : $N, \{T_n\}, \{A_n\}, R_{\text{req}}, A_{\text{req}}, R_{step}, A_{save}$
output: $\{x_n\}, r_{\text{all}}, a_{\text{all}}$
1  Initialize all $\{x_n\}$ to 1, set $r_{all} = 0$, $a_{all} = 0$;
2  Initialize $i = 0$, and all $\{x'_n\}$ to 1. /* use $\{x'_n\}$ to represent the
   parallelization degree of $F_n$, where $x'_n > 0$ */;
3  while $r_{all} < R_{req}$ do
4      $r_{all}$=Incx ($\{x'_n\}, \{T_n\}$); /* select the bottleneck block and
       increase its parallelization degree by increasing $\{x'_n\}$ */
5  end
6  compute $a_{all}$ using $\{x'_n\}$ by (13);
7  repeat
8      $\{x_n\}$=Clst ($\{x'_n\}, r_{all}, \{T_n\}, \{A_n\}$); /* to find the optimal
       partition for the given parameters */
9      compute $a_{all}$ using $\{x_n\}$ by (13);
10     if $a_{all} > A_{req}$ then
11         pop $\{x'_n\}$ and $r_{all}$ from the stack; /* the current solution is not
           feasible. Go to the previous one */
12     else
13         if This $\{x'_n\}$ has been explored before then break;
14         push $\{x'_n\}$ and $r_{all}$ onto the stack;
15         $r_{all}$=Incx ($\{x'_n\}, \{T_n\}$);
16     end
17 until;
18 Return ($\{x_n\}$, $r_{all}$, $a_{all}$);
```

A. Experimental setup

In our experiments, we use *eXCite* with default settings to accomplish all the synthesis tasks needed [5]. We use *ModelSim* [24] and *Quartus* II [25] (*Cyclone* II [26] as the target hardware) to obtain the timing and area information of the HDL files. We use *lp_solve* [27] to solve the MILP problem.

We have adopted seven large streaming applications from the high-level synthesis benchmark suite *CHstone* [21]. They come from real applications and consist of programs from the areas of image processing, security, telecommunication, and digital signal processing. They are ADPCM, AES encryption and decryption, JPEG encoder and decoder, GSM (LPC analysis part), and Filter groups. The C-code size of these benchmarks are around 1000 lines.

B. Validation of proposed approach

Both our MILP and heuristic solutions rely on proper modeling of functions, blocks, modules as well as the overall system parameters. To validate the models, we compare the throughput and area data obtained by using our model with corresponding data obtained directly using *eXCite*. The relevant results are given in Table III. Here, the second column indicated by $\{x_n\}$, lists designs with different random partitions and parallelizations. The last four columns summarize each design's throughput (represented by its inverse) and area derived from our model and from *eXCite*. We have set α^{le} in (3) to 0.235 from data fitting[2] and β in (8) to 0.025 [26].

TABLE III
System model validation

Benchmark	$\{x_n\}$	r_{all}^{-1} (cycles/Byte)		a_{all}[1]	
		Model	*eXCite*	Model	*eXCite*
GSM	{0,2,1,0,2,0,1,0,0,1}	1667	1661	15392	14781
	{0,2,1,1,2,2,2,0,1,1}	1545	1551	17995	17096
	{1,0,1,1,1,1,0,0,0,1}	1545	1545	12522	12132
	{0,2,1,0,2,1,1,0,0,1}	1204	1208	16087	16008
JPEG decoder	{0,0,0,0,1,0,0,2}	1766	1861	10301	8401
JPEG encoder	{2,1,0,1,0,1,0,1,1,2}	419	453	17997	16309

[1] a_{all} presents total system area normalized by $a_{\text{all}}^{\text{le}}+\beta \cdot a_{\text{all}}^{\text{mem}}$

[2]The α^{le} value was obtained as follows: (i) used *eXCite* to estimate the area for every possible partition of the first three functions in the GSM program, and (ii) determined α^{le} by fitting the area data to Equations (3).

We can conclude from Table III that the throughput and area obtained from our system model (by Equations (1) to (14)) and actual synthesis results (by *Quartus*) match well, with an average difference of 2% and 7% for throughput and area, respectively. Observed from the effectiveness of our solution later, such differences are tolerable considering typical variations in high-level synthesis results.

C. Evaluation of MILP and heuristic solutions

We first compare the effectiveness of our MILP and heuristic solutions. We show the solutions for seven benchmarks in Table IV. The objective function is minimizing a_{all} subject to a given throughput constraint (used for many ASIC implements for streaming applications), and the throughput constraint is shown in the third column (as the inverse of R_{req}) in the table. The last three columns are the partition and parallelization results, the throughput (from simulation by *Modelsim*), and area results (from synthesis by *Quartus*) for the MILP and heuristic solution, respectively.

TABLE IV
Partition and parallelization results for seven benchmarks
(**OBJ:** $\min a_{\text{ALL}}$)

Benchmark		Constraints[1]	MILP vs. Heuristic results		
		R_{req}^{-1}	$\{x_n\}$	r_{all}^{-1}	a_{all}
ADPCM	MILP	200	{0,2,0,0,1,1}[2]	197	16624
	Heu.		{2,0,0,1,1,1}	164	17513
AES encryption	MILP	5000	{1,1,1,0,1,0,1}	4678	16463
	Heu.		{1,1,1,0,1,0,1}	4678	16463
AES decryption	MILP	4500	{1,0,0,2,0,0,2}	4325	24768
	Heu.		{1,2,0,2,2,0,2}	3867	31036
JPEG encoder	MILP	375	{2,1,1,1,1,1,1,1,0,2}	355	23944
	Heu.		{2,1,1,1,1,1,1,1,0,2}	355	23944
JPEG decoder	MILP	2500	{0,0,0,0,1,0,0,2}	1766	8065
	Heu.		{0,0,0,0,1,2,0,1}	1670	8362
GSM	MILP	1000	{0,2,1,0,2,0,2,0,0,1}	987	15937
	Heu.		{2,2,0,2,2,0,2,0,0,1}	833	18775
Filter Groups	MILP	18000	{0,0,0,1,0,1,0,2,2,2,0,1,1,1}	17102	28776
	Heu.		{1,0,0,1,0,1,0,2,2,2,0,1,1,1}	10372	28994

[1] With the objective as $\min a_{\text{all}}$, we do not consider the area constraint (A_{req}) here.
[2] The $\{x_n\}$ indicates that the first two functions are merged into one block which is duplicated once, the next 3 functions are merged into one block, and the last function is its own block.

From the experiment data, we can conclude that results from both the MILP and heuristic methods meet the design constraints well. As to the optimality, for minimizing area, MILP solutions always have an equal or smaller area than the heuristic ones, which reflects the fact that the MILP based approach is an exact method. The heuristic solutions are either the same or very close to the optimal MILP ones, with a difference of 7.5% on average, which shows its effectiveness.

We further investigate the quality of our MILP and heuristic solutions. Figure 2 illustrates the various designs of the GSM benchmark with 52 different partition and parallelization solutions found either by the MILP (green triangles) or the heuristic (blue circles) method or randomly (black dots). It can be readily seen that MILP results are all on the pareto-optimal front with different throughput and area combinations, which demonstrates the optimality. As to heuristic results, they either overlap with the pareto-optimal front or are close to, with an average 2.3% difference with the optimal MILP results, which shows its effectiveness.

Table V summarizes GSM accelerator designs satisfying different throughput and area constraints and objectives. In this table, the first four columns show the objectives and the constraints, and the last three columns show the partition and parallelization results obtained by the MILP and heuristic methods. The first four rows' objective is to minimize area as Table IV. Given area constraints, the last six rows maximizes throughput, which may represent the case where the best performance on a specific FPGA chip with limited resource is

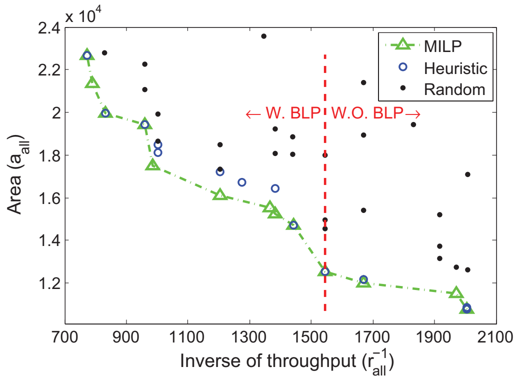

Fig. 2. Parato-optimal front in the GSM application

needed. The last two rows illustrate the case with the constraints on logic element and memory area, respectively. In all the cases, both MILP and heuristic solutions meet the constraints, while the quality of the MILP solutions are better, which demonstrates the effectiveness of the methods for in various constraints and objectives.

TABLE V
PARTITION AND PARALLELIZATION RESULTS FOR THE GSM
(UNDER DIFFERENT CONSTRAINTS AND OBJECTIVE FUNCTIONS)

	Objective	Constraints		MILP vs. Heuristic result		
		R_{req}^{-1}	A_{req}	$\{x_n\}$	r_{all}^{-1}	a_{all} or $a_{all}^{le}/a_{all}^{mem}$
MILP	$\min a_{all}$	1000	–	{0,2,1,0,2,0,2,0,0,1}	987	15937
Heuristic				{2,2,0,2,2,0,2,0,0,1}	833	18775
MILP	$\min a_{all}$	1600	–	{1,0,1,1,1,1,0,0,0,1}	1545	12132
Heuristic				{1,0,1,1,1,1,0,0,0,1}	1545	12132
MILP	$\max r_{all}$	–	15000	{1,0,1,1,1,1,0,0,0,1}	1545	12132
Heuristic				{1,0,1,1,1,1,0,0,0,1}	1545	12132
MILP	$\max r_{all}$	–	19000	{0,2,1,0,2,2,1,0,0,1}	987	17415
Heuristic				{0,2,2,0,2,1,1,0,0,1}	1204	18341
MILP	$\max r_{all}$	–	[1]19000	{2,2,0,2,2,0,2,0,0,1}	987	17094/67284
Heuristic			/70000	{0,0,2,0,2,1,1,0,0,1}	1204	19927/52208

[1] With two separated constraints for A_{req}^{le} and A_{req}^{mem}, respectively.

The heuristic can find reasonable solutions in much shorter time as shown in Table VI. The first four columns show the objectives and constraints. The next two columns summarize the run time of the two methods, while the last two columns show the corresponding throughput and area results. The table shows that the heuristic takes 350-15000× shorter time than the MILP, while the quality of the solutions degrade by 7.2% on average. Furthermore, it can find a valid solution when the MILP method fails as shown in the sixth and ninth rows. In these cases, the applications either contain lots of functions or have tight constraints. It is known that the shape of the feasible region defined by the constraints for an MILP instance can influence its complexity significantly. Hence, our heuristic method is a good alternative when the problem instances prevent the MILP method to find a solution within a reasonable amount of time.

VI. CONCLUSIONS

In this paper, we focus on the development accelerators for streaming applications, and introduce two methods aiming at improving the flexibility and the result quality of C2RTL design methodology. Our work adopts a hierarchical framework with automatic C-code partition and block-level parallelization so as to produce the most desirable synthesis results. We propose an MILP based method to find the optimal partition and parallelization for a given C program. We also devise a heuristic algorithm in order to handle cases where the MILP method is too costly. Experimental results obtained from seven real applications show that our approaches are effective in exploring the partition and parallelization degree. Specifically, the solutions obtained by the MILP method are always optimal while the solutions obtained by the heuristic method are not as good but the run time can be several orders of magnitude shorter.

TABLE VI
COMPARISON OF RUNNING TIME

Bench -mark	Objective	Constraints		Time (sec)		Result (r_{all}^{-1}, a_{all})	
		R_{req}^{-1}	A_{req}	MILP	Heu.	MILP	Heu.
GSM (N=10)	$\min a_{all}$	1000	–	9.089	0.025	987, 15937	833, 18775
	$\max r_{all}$	3000	17000	37.648	0.192	1208, 16008	1442, 15171
	$\max r_{all}$	–	[1]19000/400000	41.135	0.098	833, 18775	1024, 18031
	$\max r_{all}$	–	18900/300000	*Failed*	0.095	*Failed*	1024, 18031
Filter groups (N=14)	$\min a_{all}$	19000	–	355.80	0.026	18548, 20829	18548, 20829
	$\max r_{all}$	–	50000	395.47	0.025	17102, 26571	10372, 28994
	$\max r_{all}$	–	30000/25000	*Failed*	0.121	*Failed*	10222, 23907

[1] With two separated constraints for A_{req}^{le} and A_{req}^{mem}, respectively.

REFERENCES

[1] J. Cong, K. Guruaj, and et al., "Domain-specific processor with 3d integration for medical image processing," in *ASAP*, 2011, pp. 247–250.
[2] R. Iyer, "Accelerator-rich architectures: Implications, opportunities and challenges," in *ASP-DAC*, 2012, pp. 106–107.
[3] J. Cong, B. Liu, and et al., "High-level synthesis for fpgas: From prototyping to deployment," *TCAD*, vol. 30, no. 4, pp. 473–491, 2011.
[4] "*Catapult-C*," http://www.mentor.com/esl/catapult.
[5] "*eXCite*," http://www.yxi.com.
[6] S. Gupta and et al., "Spark: A high-level synthesis framework for applying parallelizing compiler transformations," in *ISVLSI*, 2003.
[7] Y. Guo and D. McCain, "Rapid prototyping and vlsi exploration for 3g/4g mimo wireless systems using integrated catapult-c methodology," in *WCNC*, 2006, pp. 958–963.
[8] M. Rossler and et al., "Rapid prototyping of a dvb-sh turbo decoder using high-level-synthesis," in *FDL*, 2009, pp. 1–6.
[9] B. Schafer, A. Trambadia, and K. Wakabayashi, "Design of complex image processing systems in esl," in *ASP-DAC*, 2010, pp. 809–814.
[10] S. Li, Y. Liu, and et al., "A hierarchical c2rtl framework for fifo connected stream applications," in *ASP-DAC*, 2012, pp. 133–138.
[11] A. Hagiescu and et al., "A computing origami: Folding streams in fpgas," in *DAC*, 2009, pp. 282 –287.
[12] J. Cong and et al., "Combining module selection and replication throughput-driven streaming programs," in *DATE*, 2012, pp. 1018–1023.
[13] Y. Liu, S. Li, H. Z. Yang, and P. Zhang, "A hierarchical c2rtl framework for hardware configurable embedded systems," in *Embedded Systems - Theory and Design Methodology*. Intech, 2012, pp. 367–386.
[14] Y. Hara and et al., "Partitioning of behavioral descriptions with exploiting function-level parallelism," vol. E93-A, 2010, pp. 488–499.
[15] A. Hormati and et al., "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," in *PACT*, 2009.
[16] J. Ceng and et al., "Maps: an integrated framework for mpsoc application parallelization," in *DAC*, 2008, pp. 754–759.
[17] E. Raman and et al., "Parallel-stage decoupled software pipelining," in *CGO*, 2008, pp. 114–123.
[18] D. Cordes and et al., "Automatic extraction of pipeline parallelism for embedded software using linear programming," in *ICPADS*, 2011.
[19] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, "Optimized generation of data-path from c codes for fpgas," in *DATE*, 2005, pp. 112–117.
[20] Y. Zhu and et al., "Acceleration of pedestrian detection algorithm on novel c2rtl hw/sw co-design platform," in *ICGCS*, 2010, pp. 615–620.
[21] Y. Hara and et al., "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *ISCAS*, 2008, pp. 1192–1195.
[22] S. McConnell, *Code complete*. O'Reilly Media, Inc., 2009.
[23] S. Li and et al., "Optimal partition and parallelization in a hierarchial c2rtl framework with fifo sizing," in *Technical Report of Tsinghua University*, http://nics.ee.tsinghua.edu.cn/people/liuyp/doc/partition.pdf.
[24] "*ModelSim*," http://www.mentor.com/products/fpga/simulation/modelsim.
[25] "*Quatrus*," http://www.altera.com.
[26] "*Cyclone* handbook," http://www.altera.com/literature.
[27] "*lp_slove* reference guide," http://lpsolve.sourceforge.net.