A C2RTL Framework Supporting Partition, Parallelization and FIFO Sizing for Streaming Applications

DAMING ZHANG, Tsinghua University
SHUANGCHEN LI, University of California, Santa Barbara
YONGPAN LIU, Tsinghua University
XIAOBO SHARON HU, University of Notre Dame
XINYU HE, Princeton University
YINING ZHANG, Tsinghua University
PEI ZHANG, Y Explorations, Inc.
HUAZHONG YANG, Tsinghua University

Developing circuits for streaming applications written in C (or its variants) can benefit greatly from C-to-RTL (C2RTL) synthesis. Yet, most existing C2RTL tools lack system-level options to trade off various design constraints, such as delay and area. This paper introduces a systematic way to accomplish C2RTL synthesis for streaming applications containing thousands of lines of C (or its variants) codes. Synthesizing circuits for such large applications presents serious challenges for existing C2RTL tools. Specifically, the proposed approach determines simultaneously the number of pipeline stages and the number of times that each functional block is duplicated in each pipeline stage. A mixed integer linear programming based solution is formulated for obtaining the optimal solution. Furthermore, a heuristic algorithm is developed for large-scale problems. To accommodate the differences of the data rates between the adjacent hardware modules, first-in-first-out (FIFO) buffers are indispensable, but their overheads are non-negligible. A parallelism-aware FIFO sizing method is also introduced to determine the optimal sizes of FIFOs. Experimental results on seven real-world applications demonstrate that the algorithms in the synthesis flow can make effective design tradeoffs and find superior solutions in a short time compared with existing approaches. Furthermore, the algorithms achieve optimal results in most cases with sub-second running time.

CCS Concepts: •**Hardware** → High-level and register-transfer level synthesis; Modeling and parameter extraction; Methodologies for EDA;•**Computer systems organization** → Embedded systems;

Additional Key Words and Phrases: System-level design and optimization, Partition, Parallelization, FIFO sizing, Streaming applications

ACM Reference Format:
Daming Zhang, Shuangchen Li, Yongpan Liu, Xiaobo Sharon Hu, Xinyu He, Yining Zhang, Pei Zhang, and Huazhong Yang, 2015. A C2RTL Framework Supporting Partition, Parallelization and FIFO Sizing for Streaming Applications. *ACM Trans. Des. Autom. Electron. Syst.* 0, 0, Article 0 (2015), 33 pages.
DOI: http://dx.doi.org/10.1145/2797135

A preliminary version of part of this work appeared in [Li et al, 2013].
This work was supported in part by High-Tech Research and Development (863) Program under contract 2013AA01320 and Huawei Shannon Lab and the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions under contract YETP0102.
Author's addresses: D. Zhang, Y. Liu (corresponding author), and H. Yang, Dept. of Electronic Engineering, Tsinghua University, Beijing, 100084, China; email: ypliu@tsinghua.edu.cn; S. Li, Dept. of Electronic and Computer Engineering, University of California, Santa Barbara, CA, 93106, USA; X. S. Hu, Dept. of Computer Science and Engineering, University of Notre Dame, Notre Dame, 46556, USA; X. He, Dept. of Electrical Engineering, Princeton University, Princeton, NJ, 08544, USA; Y. Zhang, Dept. of Microelectronics, Tsinghua University, Beijing, 100084, China; P. Zhang, Y Explorations, Inc., San Jose, CA, 95134, USA.

DOI: http://dx.doi.org/10.1145/2797135

1. INTRODUCTION

Domain-specific hardware accelerators are gaining popularity in heterogeneous multiprocessor system-on-chips (MPSoCs) [Cong et al. 2011; Iyer 2012]. Such accelerators have the potentials to provide orders of magnitude improvements in execution time. However, most of the domain-specific applications are developed in C (or its variants), instead of hardware description languages (HDL), such as VHDL and Verilog HDL. Furthermore, decades of developing C-language based embedded applications have resulted in a great deal of legacy codes in various application domains. For example, most of the image processing programs are developed in C or C++ (such as JPEG encoders/decoders and MPEG-2 encoders). Therefore, it would be of great benefit if the C programs are used directly to synthesize domain-specific accelerators for heterogeneous MPSoCs.

The increasing interest in high quality C-to-RTL (C2RTL) synthesis [Cong et al. 2011b] has led to the development of a number of commercial and academic C2RTL tools. Most of these tools can synthesize moderate-sized C programs (e.g., with hundreds of lines) into HDL files quite efficiently. In fact, various domain-specific accelerators (e.g., 3G/4G wireless communication [Guo and McCain 2006], digital video broadcasting [Rossler et al. 2009], face detection [Schafer et al. 2010], wireless sensor nodes [Pasha et al. 2012]) have been developed with existing C2RTL tools (e.g., Catapult-C, CoDeveloper, CWB).

However, existing C2RTL tools face significant challenges in terms of synthesis quality and running time. For large C programs, the difficulties can be attributed to a range of factors, such as inefficient automatic generation of finite-state machines (FSM), which is used to describe complex data dependence relationships. Detailed discussions of related work are given in Section 8.

In this paper, we introduce a comprehensive approach that can efficiently synthesize large-sized C (or its variants) programs with thousands or more lines of codes into RTL level descriptions. We leverage a hierarchical synthesis flow to achieve system-level optimization based on an existing C2RTL tool. This synthesis flow includes two major tasks: (i) simultaneous partition and parallelization, and (ii) FIFO sizing. Partition determines the number of pipeline stages in the application, while parallelization determines the number of times that each functional block is duplicated in each pipeline stage. Our specific contributions are summarized below:

— present a hierarchical synthesis flow for system-level optimization;
— introduce a mixed integer linear programming (MILP) based method to find the optimal solutions of simultaneous partition and parallelization under the given constraints;
— propose a heuristic algorithm to deal with the scalability challenge faced by the MILP-based solution;
— develop a parallelism-aware FIFO sizing method to minimize the FIFO sizes based on the result of simultaneous partition and parallelization while satisfying the given constraints.

We have applied our proposed algorithms to synthesize accelerators for seven real-world streaming applications. The algorithms achieve optimal results in most cases with sub-second running time. Though the focus of the paper is on streaming applications, the algorithms used in the synthesis flow are also suitable for the programs with branches and feedback loops. Detailed analysis is presented in Section 6.

The rest of the paper is organized as follows. Section 2 presents the motivation and describes the hierarchical synthesis flow. Section 3 introduces the system model. The MILP-based solution and the proposed algorithm for solving the problem of simultaneous partition and parallelization are discussed in Section 4. The method for parallelism-aware FIFO sizing is presented in Section 5. Section 6 discusses the extensions and limitations of the synthesis flow. Section 7 presents the experimental results. Section 8 discusses the related work and Section 9 concludes the paper.

2. PRELIMINARIES

In this section, we describe a hierarchical synthesis flow for finding a hardware implementation that has the minimum area (resp., initiation interval[1]) while satisfying the given initiation interval (resp., area) constraint for a given streaming application. We then elaborate the motivation for simultaneous partition and parallelization, as well as FIFO sizing in the flow.

2.1. Hierarchical Synthesis Flow

Flat C2RTL approaches (e.g., [Rossler et al. 2009; Zhu et al. 2010]) automatically transform entire C algorithms into large modules. Such approaches suffer from low quality (e.g., throughput or area) of the synthesized circuits (hardware) for large C programs. Furthermore, the running time of these approaches is intolerably long, or they even fail to find a solution if the C program has thousands of lines. Though the users adjust the code styles or unroll the loops, the effect of such adjustments is usually limited.

Unlike conventional flat approaches, a hierarchical synthesis flow, which is extended based on [Li et al. 2013], is capable of exploring a much larger and more complex design space. The synthesis flow exploits simultaneous partition and parallelization, as well as FIFO sizing, to reduce the initiation interval and area.

Fig. 1 depicts the hierarchical synthesis flow with an FFT example on four levels: function, block, module and system. As shown in Fig. 1(A), the input is a C program containing N functions and a constraint of initiation interval or area. The functions are connected in a straight-line pattern, i.e., the output of the ith function is the input of the $(i+1)$th function. Such data dependence pattern is common in streaming applications [Hara et al. 2008]. For example, the JPEG encoder has five functions: discrete cosine transformation (DCT), quantization, zigzag scan, DPCM encoder and RLE encoder, and the functions are connected in series, one after another.

The hierarchical synthesis flow contains two main tasks. In Task1, we optimize the partition and parallelization simultaneously. The initiation interval and area are extracted for each C function (F_i), which are synthesized by a C2RTL tool. Based on the user-specified initiation interval or area constraint, the flow determines an optimal implementation by clustering functions into blocks ($B_{i,j}$) and duplicating certain blocks (e.g., $B_{1,2}$). Note that the functions in the same block have the same parallelization degree. The blocks are then synthesized into HDL modules (M_i). In addition, to connect the duplicated modules (e.g., M_1), we use small controllers, 1-to-n demuxes and n-to-1 muxes. The shared I/O interface is a general structure for two adjacent modules with any parallelism degrees. In Task2, the sizes of the FIFOs connecting the modules are determined based on their parallelism degrees. Finally, the hardware system for the streaming application is constructed by these modules and FIFOs.

Note that the functions in the hierarchical synthesis flow usually contain less than 200 lines of C programs [McConnell 2009], for which existing C2RTL tools can synthesize high-quality circuits at this scale. For larger functions, the techniques

[1]Initiation interval is the number of clock cycles between two subsequent inputs to system and is the reciprocal of the throughput of the system.

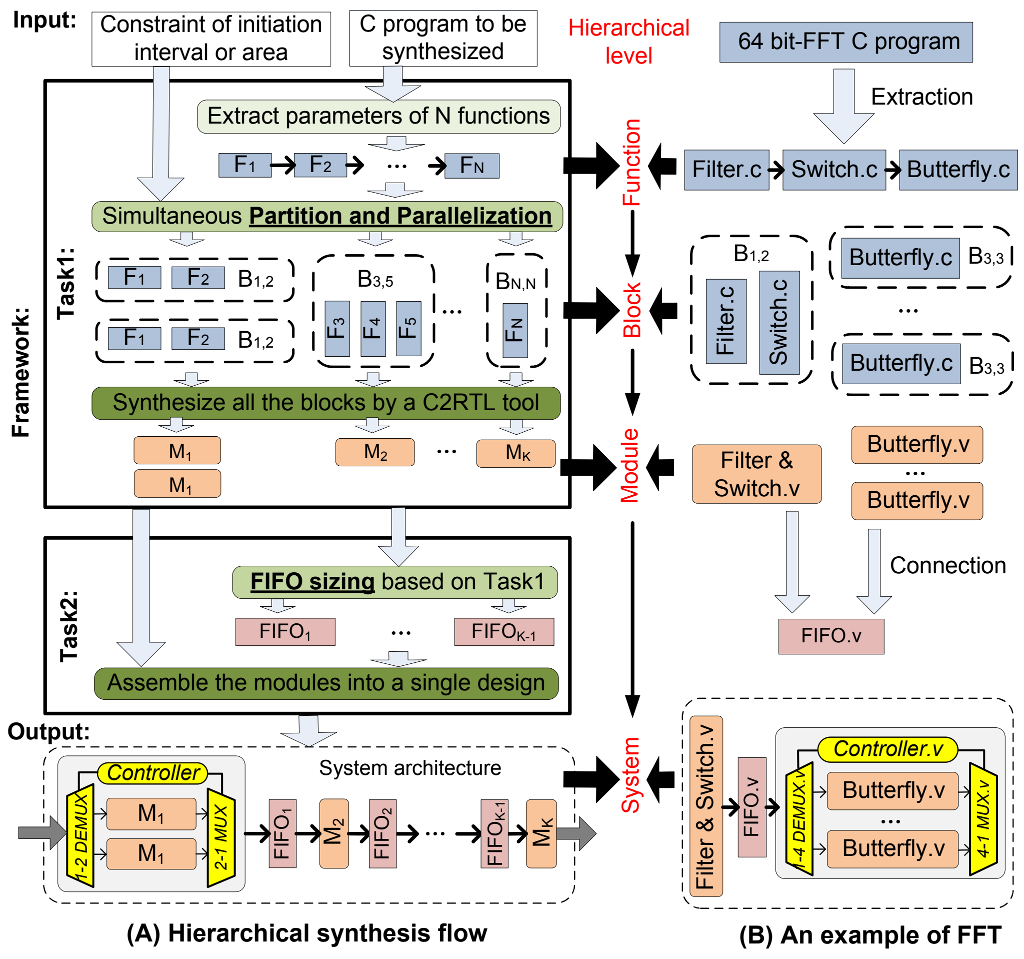


Fig. 1. Hierarchical synthesis flow with an FFT example.

introduced in [Ceng et al. 2008] can be employed to automatically divide the functions into smaller ones.

Fig. 1(B) shows an example of applying the hierarchical synthesis flow to implement a 64-bit FFT. The FFT program contains three functions: filters, switch functions, and butterfly units. Through the procedure of simultaneous partition and parallelization, the functions are merged and duplicated as two types of blocks. Block $B_{1,2}$ contains filters and switch functions while block $B_{3,3}$ includes four duplicated butterfly computing units. After Task1, the blocks are synthesized as modules by a C2RTL tool. Task2 is then performed to determine the suitable size of the FIFO between the two types of modules. Finally, the FFT system is formed with these modules, the FIFO, a 1-to-4 demux, a 4-to-1 mux and a small controller.

2.2. Motivation

We present the motivations for simultaneous partition and parallelization, and FIFO sizing in the hierarchical synthesis flow.

2.2.1. Motivation for Simultaneous Partition and Parallelization. A streaming application usually contains several functions (F_1-F_N). To decrease the initiation interval or the area of the application, there are two key design knobs: partition and parallelization. Shown in Fig. 2, they optimize the streaming application in two dimensions. Partition determines the number of pipeline stages in the application, while parallelization determines the number of times that each functional block is duplicated in each pipeline stage.

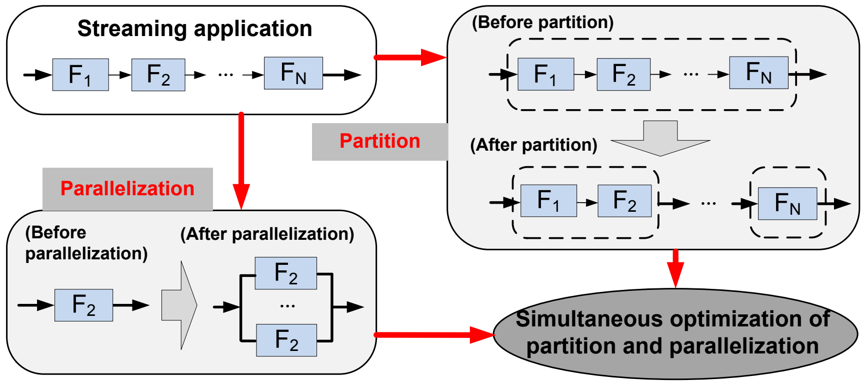


Fig. 2. Applying partition and parallelization on functions within a streaming application.

Function-level partition greatly impacts the initiation interval and area of a streaming application. Fig. 3 summarizes the initiation intervals and areas of several different partition strategies for the GSM program [Li et al. 2013]. Here, the Y-axis shows the initiation interval ($cycles$) and the area (the number of $LUTs$ in an FPGA). The X-axis presents different partition strategies (different combinations of the given six functions) without parallelization. The initiation intervals and areas are obtained from the synthesized modules. As we can see, the first partition strategy (P1) clusters the first five functions into one big synthesized block (module), and results in the longest initiation interval. The last partition strategy (P4) uses smaller functions, and achieves a smaller initiation interval, but consumes more area.

As the initiation interval is typically limited by the slowest block, a partition strategy leading to a consistent speed for all modules is preferred. However, reducing area requires more functions integrated into one big (but slow) synthesized block to maximize resource sharing. Therefore, the partition strategy has to trade off between initiation interval and area.

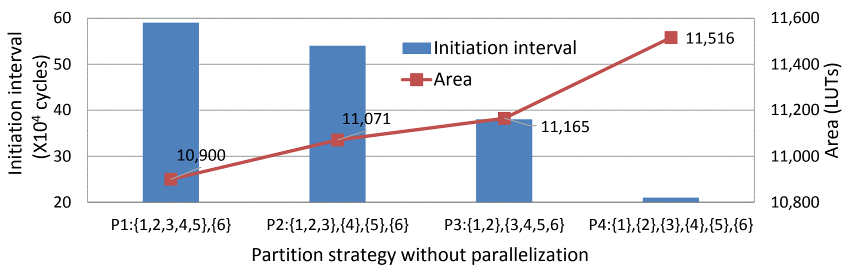


Fig. 3. Synthesized results for GSM using different partition strategies.

Besides partition, block-level parallelization provides another design knob. It duplicates certain blocks to reduce the initiation interval while incurring some area overhead. The shared I/O interface is used as a general structure for duplicated modules. Fig. 4 presents the initiation interval ($cycles$) and the area (the number of $LUTs$ in an FPGA) of a module (synthesized block with multiple DCT functions in JPEG decoder) in the Y-axis, with different parallelism degrees in the X-axis. As the parallelism degree increases with the area, the initiation interval is reduced. However, at parallelism degrees of 9 and 10, the initiation interval no longer decreases, because the shared I/O interface for duplicated modules has limited bandwidth for data transmission. Thus, the initiation interval has reached the minimum value, based on the latencies of data input and output. This implies that there exists a maximum parallelism degree for a block, and it is wasteful to use too many duplicated modules.

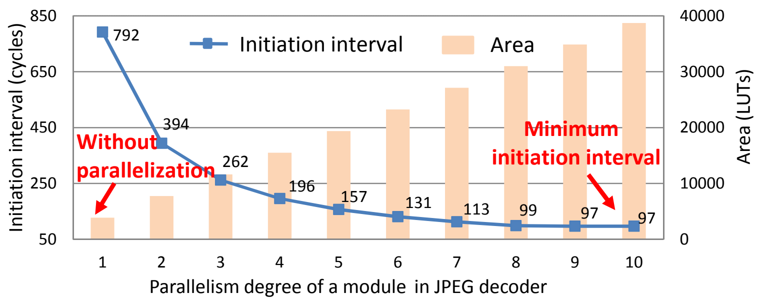


Fig. 4. Synthesized results for DCT using different parallelism degrees.

In general, both partition and parallelization make tradeoffs between initiation interval and area. Furthermore, different partition strategies lead to different parallelization choices, and vice versa. Therefore, it is necessary to consider partition and parallelization simultaneously in order to find the optimal result. However, the algorithm for simultaneous optimization developed in [Li et al. 2013] is coarse-grained and is usually not able to find the optimal result when the solution is already near the optimal one. In this paper, we propose an improved heuristic algorithm. It has an extra fine-grained partition-based tuning step, which performs well when the solution is close to the optimal one. The proposed algorithm is introduced in detail in Section 4.3.

2.2.2. Motivation for FIFO Sizing. In the hierarchical synthesis flow, it is important to determine the FIFO sizes between modules after simultaneous partition and parallelization because the inserted FIFOs help to match the different data rates between the modules. We show the total latency per execution of a JPEG encoder for different FIFO sizes between module 1 and module 2. As seen from Fig. 5, different FIFO sizes in the X-axis lead to a 50% performance difference. Moreover, the total latency per execution in the Y-axis no longer decreases after a certain point, as the FIFO with a large enough size matches the data rates of the module 1 and module 2. Therefore, it is wasteful to use too large FIFOs.

Finding the right FIFO size is non-trivial. Traditional methods for FIFO sizing (e.g., [Wang et al. 2012]) are based on exhaustive RTL simulation, and are quite time-consuming. Authors in [Li et al. 2012a] proposed an analytical method with a linear complexity for FIFO sizing, but the method does not consider the parallelism degrees of the modules. Thus, the method fails to work if the parallelism degrees of two modules are different. We introduce a parallelism-aware FIFO sizing method in Section 5.

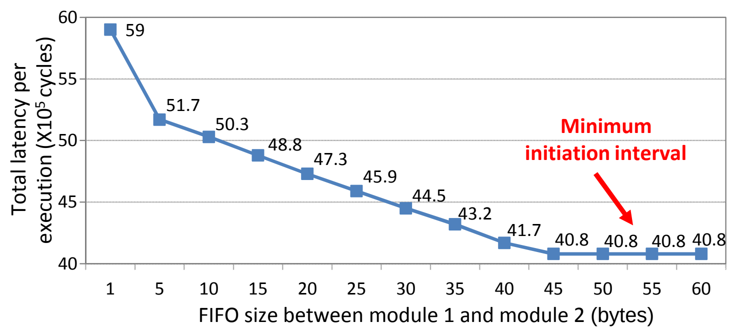


Fig. 5. Synthesized results for two modules using different FIFO sizes.

3. SYSTEM MODELING

To explore the large system-level design space for simultaneous partition and parallelization, as well as FIFO sizing, we need efficient models for a given C program. In this section, we introduce the parameters of the models on four levels: function, block, module and system. The parameters are presented in Table I, and the relationships among the four levels are shown in Fig. 6. More discussions are given below.

Table I. Parameters for system modeling

Level	Parameters	Description
Function	T_n	Latency of function F_n ($cycles$)
	T_n^{in}, T_n^{out}	Input/Output latency of F_n ($cycles$)
	S_n^{in}, S_n^{out}	Input/Output data size for F_n ($bytes$)
	A_n	Area of F_n after C2RTL synthesis ($LUTs$)
Block	$T_{i,j}$	Latency of block $B_{i,j}$ ($cycles$)
	$T_{i,j}^{\text{in}}$, $T_{i,j}^{\text{out}}$	Input/Output latency of $B_{i,j}$ ($cycles$)
	$S_{i,j}^{\text{in}}$, $S_{i,j}^{\text{out}}$	Input/Output data size for $B_{i,j}$ ($cycles$)
	$A_{i,j}$	Area of $B_{i,j}$ after C2RTL synthesis ($LUTs$)
	α_1, α_2	Area-saving factors when clustering
	$P_{i,j}$	Parallelization upper bound of $B_{i,j}$
Module	TM_k	Latency of all duplicated M_k modules ($cycles$)
	TM_k^{in}, TM_k^{out}	Input/Output latency of all duplicated M_k modules ($cycles$)
	SM_k^{in}, SM_k^{out}	Input/Output data size for all duplicated M_k modules ($bytes$)
	XM_k	Parallelism degree of all duplicated M_k modules
	A_{fifo}	Area overhead of FIFOs per byte of data ($LUTs$)
System	R_{req}	Initiation interval constraint ($cycles$)
	A_{req}	Area constraint ($LUTs$)

3.1. Parameters of Functions

Functions are the basic elements for optimization. We use F_n to denote the nth function, and assume that there are N functions. For each function, there is no overlapping between reading input, computing and writing output. The parameters of F_n obtained by C2RTL synthesis are assumed to be constant.[2] These constant

[2]In our work, the parameters of a function are extracted by running a C2RTL tool under the configuration for maximizing performance. Therefore, these parameters are constants. It is possible to extend our approach to more general cases by allowing a set of values for each parameter.

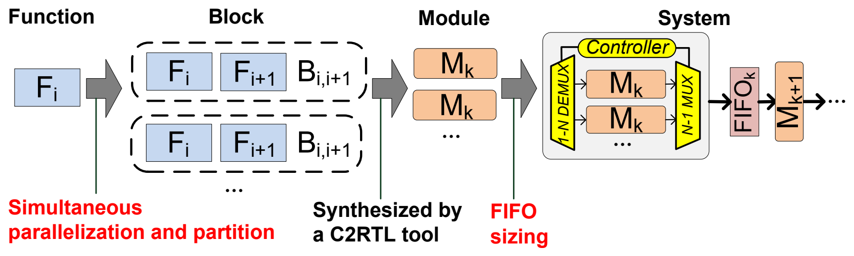


Fig. 6. Relationships among function, block, module and system.

parameters are presented below. The latency of F_n is represented as T_n, which contains the input latency, the processing time and the output latency. (Note that the focus of this paper is block-level pipeline design, and therefore the intra-function pipeline is not considered here. To extend this work to fully pipelined applications, we need to modify the latency model.) T_n^{in} and T_n^{out} denote the input and output latency for F_n to load and write S_n^{in} and S_n^{out} bytes of the input and output data, respectively. Since the output data from a prior function are consumed by the next function immediately following it in a streaming application, we have

$$S_n^{\text{in}} = S_{n\text{-}1}^{\text{out}} \qquad \forall n \in [2, N]. \tag{1}$$

The area of function F_n is A_n.

3.2. Parameters of Blocks

Blocks are formed by clustering neighboring functions. We denote a block including the functions F_i, F_{i+1}, $\cdots$, F_j as $B_{i,j}$. Block-level parameters are shown in Table I. They are similar to function-level parameters, and are derived directly from function-level parameters as follows.

The total latency $T_{i,j}$ of block $B_{i,j}$ is calculated as follows,

$$T_{i,j} = T_i^{\text{in}} + T_j^{\text{out}} + \sum_{n=i}^{n=j} (T_n - T_n^{\text{in}} - T_n^{\text{out}}). \tag{2}$$

That is, $T_{i,j}$ is obtained by summing the latencies of the functions in $B_{i,j}$. The block latency does not include the input and output latencies (i.e., T_n^{in} and T_n^{out}) of the internal functions. Since the internal functions in $B_{i,j}$ are synthesized into one module, only the input latency of the first function (T_i^{in}) and the output latency of the last function (T_j^{out}) contribute to the latency of the block. Equation (3) shows the input/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 $B_{i,j}$. They are equal to those of the first/last function (i.e., F_i/F_j) in $B_{i,j}$.

$$T_{i,j}^{\text{in}} = T_i^{\text{in}}, \quad T_{i,j}^{\text{out}} = T_j^{\text{out}}; \qquad S_{i,j}^{\text{in}} = S_i^{\text{in}}, \quad S_{i,j}^{\text{out}} = S_j^{\text{out}}. \tag{3}$$

$A_{i,j}$ is the area of $B_{i,j}$ after C2RTL synthesis. It is possible to use the actual synthesized results for $A_{i,j}$. However, the overhead could be rather significant if the number of the functions in $B_{i,j}$ is large. We adopt a linear function to calculate $A_{i,j}$ as follows,

$$A_{i,j} = \begin{cases} \sum\limits_{n=i}^{n=j} [A_n - (A_n \cdot \alpha_1 + \alpha_2)] & i < j \\ A_i & i = j. \end{cases} \tag{4}$$

α_1 and α_2 are referred to as area-saving factors, which are used to capture resource sharing when clustering multiple functions into one block. The values of α_1 and α_2 depend on the actual functions to be clustered. For C programs from the same application domain, the functions usually have similar area-saving factors. The values of α_1 and α_2 are obtained by data fitting for a linear function based on a set of the synthesized results. α_1 is the slope while α_2 is the intercept. Considering the different data quantities and access patterns (including aging variables, i.e., data that remain in memory during iterations) in each function, we choose the minimum values (the worst case) among all the saved synthesized area after clustering to calculate α_1 and α_2. In this way, the calculated $A_{i,j}$ is not smaller than the synthesized area. Thus, the unified values of α_1 and α_2 are suitable for different applications. Fig. 7 presents an example of data fitting to obtain the area-saving factors among all the synthesized results (from CHstone [Hara et al. 2008] benchmarks). The X-axis is the synthesized area before clustering and the Y-axis is the minimum saved synthesized area after clustering. It can be seen that the average error is lower than 10%. More results are validated in the experiments (Section 7.2).

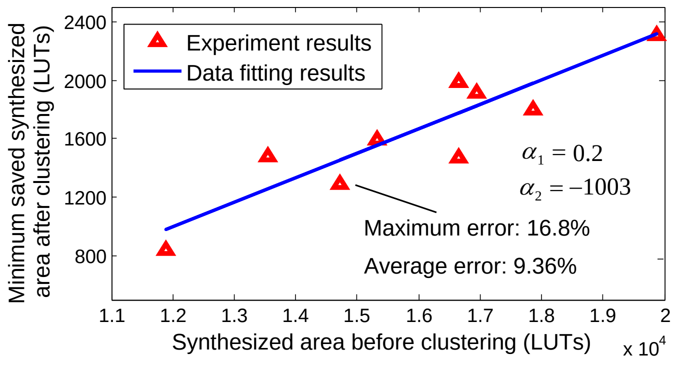


Fig. 7. Data fitting for area-saving factors.

In addition, we introduce an upper bound on the parallelism degree of block $B_{i,j}$, and denote it as $P_{i,j}$. The upper bound exists, because the synthesis flow uses a general interface, where the multiple functions in a block share the same I/O interface. $P_{i,j}$ depends on the relationship between the input, the output, and the total latencies of block $B_{i,j}$ as follows,

$$P_{i,j} = \left\lceil \frac{T_{i,j}}{\max\{T_{i,j}^{\text{in}}, T_{i,j}^{\text{out}}\}} \right\rceil. \tag{5}$$

Fig. 8 illustrates why the upper bound exists. In the figure, the X-axis is the time, and the duplicated blocks are presented in the Y-axis. Block $B_{1,2}$ has a parallelism degree of 3, i.e., $B_{1,2(1)}$ to $B_{1,2(3)}$. $B_{1,2(1)}$ cannot start a new execution cycle when it finishes the previous execution at T_1, as the input port is occupied by $B_{1,2(3)}$ ($T_1 < 3T_1^{\text{in}}$). Thus, $B_{1,2(1)}$ has to wait until $B_{1,2(3)}$ finishes data loading at $3T_1^{\text{in}}$. In this case, adding more copies of $B_{1,2}$ cannot decrease the initiation interval any further. Hence the upper bound of the parallelism degree is equal to 3.

3.3. Parameters of Modules

A module is a synthesized block based on the partition. As shown in Fig. 1, we define M_k as the kth type of modules and there are K types of modules in the system.

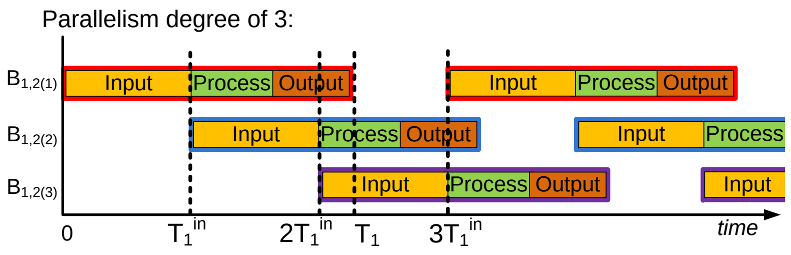


Fig. 8. Execution patterns of three duplicated blocks.

The parameters of M_k are obtained by C2RTL synthesis. TM_k is the latency of all the duplicated modules of M_k. $TM_k^{\mathrm{in}}/TM_k^{\mathrm{out}}$ denotes the input/output latency for all the modules of M_k to load/write $SM_k^{\mathrm{in}}/SM_k^{\mathrm{out}}$ bytes of data. According to the defined input/output data size of F_n in Section 3.1, we have

$$SM_k^{\mathrm{in}} = S_i^{\mathrm{in}}, \quad SM_k^{\mathrm{out}} = S_j^{\mathrm{out}}, \tag{6}$$

where $S_i^{\mathrm{in}}/S_j^{\mathrm{out}}$ is the input/output data size ($bytes$) of the first/last function in all the modules of M_k. Furthermore, the output data of a prior module are consumed by the next modules (M_{k+1}) in streaming applications and we have

$$SM_k^{\mathrm{out}} = SM_{k+1}^{\mathrm{in}}. \tag{7}$$

XM_k is the parallelism degree of all duplicated M_k modules, and A_{fifo} represents the area overhead of FIFOs per byte of data.

3.4. Constraints of the System

As indicated earlier, designing the system for a streaming application often requires the hardware to satisfy a certain constraint of initiation interval or area. Here, we use R_{req} and A_{req} to represent the constraints of initiation interval and area for the system design.

Based on the parameters introduced here, Section 4 discusses the algorithms of simultaneous partition and parallelization, while Section 5 presents the methods of FIFO sizing.

4. SIMULTANEOUS PARTITION AND PARALLELIZATION

In this section, we describe the variables and the formulation of simultaneous partition and parallelization. We then present an MILP-based solution, followed by a heuristic algorithm.

4.1. Variables and Problem Formulation

4.1.1. Variables. We summarize the variables in Table II. The variable set $\{x_n\}$ represents the result of simultaneous partition and parallelization. x_n equals zero if function F_n is not the last function in any block. Otherwise, the non-zero value of x_n indicates the parallelism degree of the block ending with F_n. For example, Fig. 9 presents a result, where F_1 and F_2 are clustered into the first block ($B_{1,2}$) while F_3, F_4 and F_5 are clustered in the second block ($B_{3,5}$). The corresponding variable set $\{x_n\}$ is expressed as follows,

$$\{x_n\} = \{\underbrace{0,2}_{B_{1,2}},\underbrace{0,0,1}_{B_{3,5}},\cdots,\underbrace{1}_{B_{N,N}}\}. \tag{8a}$$

Table II. Variables for simultaneous partition and parallelization

Variables	Description
$\{x_n\}$	Result of simultaneous partition and parallelization
$y_{i,j}$	Intermediate variable based on $\{x_n\}$
$z_{i,j}$	Temporary variable representing $x_j \cdot y_{i,j}$
$r_{i,j}$	Initiation interval of $B_{i,j}$ (*cycles*)
r_{all}	Initiation interval of the system (*cycles*)
a_{all}	Area of the system (FIFO included) (*LUTs*)
$a_{\text{block}}, a_{\text{fifo}}$	Area of all the blocks / FIFOs (*LUTs*)

$x_1 = 0$, because F_1 is not the last function in block $B_{1,2}$; $x_2 = 2$, which indicates F_2 is the last function in block $B_{1,2}$ and the parallelism degree of $B_{1,2}$ is 2.

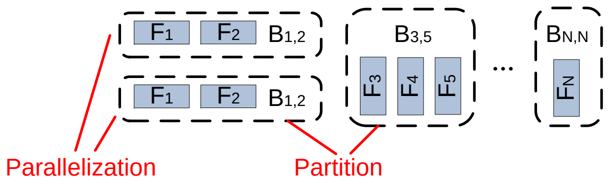


Fig. 9. Explanation for the variable $\{x_n\}$.

We introduce a 0-1 variable $y_{i,j}$ as an intermediate variable. $y_{i,j} = 1$, if $F_i, F_{i+1}, \cdots, F_j$ are merged to form the block $B_{i,j}$. Otherwise, $y_{i,j} = 0$. For example, the case in Fig. 9 is denoted as follows,

$$\{y_{i,j}\} = \left\{ \begin{array}{ccccccc} 0, & 1, & 0, & 0, & 0, & \cdots, & 0 \\ & 0, & 0, & 0, & 0, & \cdots, & 0 \\ & & 0, & 0, & 1, & \cdots, & 0 \\ & & & 0, & 0, & \cdots, & 0 \\ & & & & 0, & \cdots, & 0 \\ & & & & & & \cdots \\ & & & & & & 1 \end{array} \right\}, \tag{8b}$$

where the first block ($B_{1,2}$) is composed of F_1 and F_2 ($y_{1,2} = 1$) while F_3, F_4 and F_5 ($y_{3,5} = 1$) form the second block ($B_{3,5}$). The last block ($B_{N,N}$) consists of F_N ($y_{N,N} = 1$).

We introduce an auxiliary variable $z_{i,j} = x_j \cdot y_{i,j}$ and use r_{all} to represent the initiation interval of the system and denote a_{all} as the area of the system. Note that the FIFOs are included. Moreover, a_{block} and a_{fifo} are the area of all the blocks and FIFOs, respectively.

4.1.2. Problem Formulation. The problem is to minimize either the initiation interval (r_{all}) or the area (a_{all}) of the system under a constraint of area (A_{req}) or initiation interval (R_{req}). The formulation is written as follows,

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

We describe the variables (r_{all} and a_{all}) in detail below. First, we explain how to calculate the initiation interval of the system (r_{all}), which is determined by the slowest block. Let $r_{i,j}$ be the initiation interval of block $B_{i,j}$. Then we have

$$r_{\text{all}} = \max_{y_{i,j}=1}\{r_{i,j}\}. \tag{10}$$

Based on the parameters of block $B_{i,j}$, $r_{i,j}$ is computed by Equation (11), if block $B_{i,j}$ exists in the system ($y_{i,j} = 1$). Two situations exist: (1) When the parallelization upper bound $P_{i,j}$ (in Equation (5)) is not reached, $r_{i,j}$ is a function of block $B_{i,j}$'s parallelism degree x_j and equals $T_{i,j}/(x_j \cdot y_{i,j})$. (2) When the parallelism degree of $B_{i,j}$ is equal to $P_{i,j}$ (shown in Fig. 8), $r_{i,j}$ is limited by $T^{\text{in}}_{i,j}$ and $T^{\text{out}}_{i,j}$, and the value is $\max\{T^{\text{in}}_{i,j}, T^{\text{out}}_{i,j}\}$. Thus,

$$r_{i,j} = \min\{\frac{T_{i,j}}{(x_j \cdot y_{i,j})}, \max\{T^{\text{in}}_{i,j}, T^{\text{out}}_{i,j}\}\}, \quad \text{if } \; y_{i,j} = 1. \tag{11}$$

We calculate the total area (a_{all}) by considering both the blocks and the FIFOs as follows,

$$a_{\text{all}} = a_{\text{block}} + a_{\text{fifo}}, \tag{12}$$

where a_{block} and a_{fifo} are the area of all the blocks and FIFOs ($LUTs$), respectively. Considering parallelization, duplicating blocks leads to an area overhead (denoted as AO), which contains 1-to-n demuxes, n-to-1 muxes and small controllers as shown in Fig. 1. Thus, a_{block} is calculated as follows,

$$a_{\text{block}} = \sum_{i=1}^{i=N}\sum_{j=i}^{j=N}[(z_{i,j} - y_{i,j}) \cdot AO + z_{i,j} \cdot A_{i,j}]. \tag{13}$$

As the FIFO sizing has not been performed in the procedure of simultaneous partition and parallelization, the area overhead of a FIFO is estimated on the basis of output data size ($z_{i,j} \cdot S^{\text{out}}_j$). For a streaming application, the FIFO needs to store all the output data in the worst case. Thus, the FIFO size is set to be the largest value to guarantee the data streaming between two adjacent modules. a_{fifo} is calculated as follows,

$$a_{\text{fifo}} = A_{\text{fifo}} \cdot (\sum_{i=1}^{i=N-1}\sum_{j=i}^{j=N} z_{i,j} \cdot S^{\text{out}}_j), \tag{14}$$

where A_{fifo} represents the area overhead of FIFOs per byte of data.

The equations above for initiation interval and area calculation based on the system model are validated in Section 7.2.

4.2. MILP-based Solution

In order to find the optimal result of simultaneous partition and parallelization, we present a solution based on mixed integer linear programming (MILP). The constraints associated with x_n and $y_{i,j}$ are presented as follows,

$$\sum_{i=1}^{i=n} y_{i,n} \leq x_n \leq Q \cdot \sum_{i=1}^{i=n} y_{i,n} \qquad x_n \in \mathbf{N}, n \in [1, N]; \tag{15a}$$

$$\sum_{i=1}^{i=j-1} y_{i,j-1} = \sum_{k=j}^{k=N} y_{j,k} \qquad \forall j \in [2, N]; \tag{15b}$$

$$\sum_{i=1}^{i=j-1} y_{i,j} + \sum_{k=j+1}^{k=N} y_{j,k} + y_{j,j} \leq 1 \qquad \forall j \in [1, N]; \tag{15c}$$

$$1 \leq \sum_{i=1}^{i=N} \sum_{j=i}^{j=N} y_{i,j} \leq N. \tag{15d}$$

Equation (15a) means that the variable x_n is greater than or equal to one only if F_n is the last function in a block. Otherwise, $x_n = 0$. Q in the equation is a very large constant (larger than any possible value in the MILP formulation). Equation (15b) means that $F_{j\text{-}1}$ is the last function in the previous block, if F_j is the first function in the current block. Equation (15c) means that each function only appears in one block. Equation (15d) limits the total number of the blocks in the system. Furthermore, we remove the auxiliary variable $z_{i,j} = x_j \cdot y_{i,j}$, which leads to nonlinear constraints. The multiplicative expression for $z_{i,j}$ is re-written as follows,

$$-Q \cdot y_{i,j} \leq z_{i,j} \leq Q \cdot y_{i,j}. \tag{15e}$$

The entire MILP formulation consists of Equation (9) to (15e). The values of x_n and $z_{i,j}$ are integers. $y_{i,j}$ is a 0-1 variable. We analyze the complexity of the formulation. Given a C program with N functions, the number of variables is the sum of $\{x_n, y_{i,j}, z_{i,j}\}$, i.e., $N^2 + N$. The number of constraints is $5N^2 + 7N + 2$. As an NP-hard problem, the MILP-based solution has an exponential complexity. Though it is extremely time-consuming to apply the MILP-based solution to large-scale problems, it is useful to gauge the quality of heuristic algorithms.

4.3. Heuristic Algorithm

Though the MILP-based solution in Section 4.2 achieves the optimal result, it faces a scalability challenge. Furthermore, it fails sometimes even for small-scale problems when the solutions reside in non-smooth feasible regions. Therefore, a more efficient algorithm is required for simultaneous partition and parallelization.

To design the algorithm, we make the following observations. The partition process brings area reduction with more resource sharing under a given constraint of initiation interval. The parallelization process reduces the initiation interval with some area overhead. The effects of the partition and parallelization processes inspire us to perform them separately in an alternate manner, which we refer to as alternate parallelization and partition (AP). To get closer to the optimal result, we also develop a partition-based tuning (PT) step after AP. Fig. 10 depicts our proposed AP+PT algorithm at a high level. In addition to AP in [Li et al. 2013], the proposed algorithm has an extra PT step based on the result of AP to provide a better result.

Below, we introduce the workflow of the proposed AP+PT algorithm and then present the partition process in the algorithm. Finally, the complexity of the algorithm is analyzed.

4.3.1. Workflow of the Proposed Algorithm. Fig. 11 presents the workflow of the proposed algorithm for minimizing the initiation interval (r_{all}) under an area constraint (A_{req}).

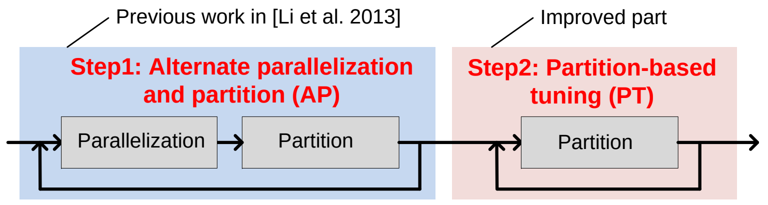

Previous work in [Li et al. 2013]
Improved part
Step1: Alternate parallelization and partition (AP)
Parallelization
Partition
Step2: Partition-based tuning (PT)
Partition

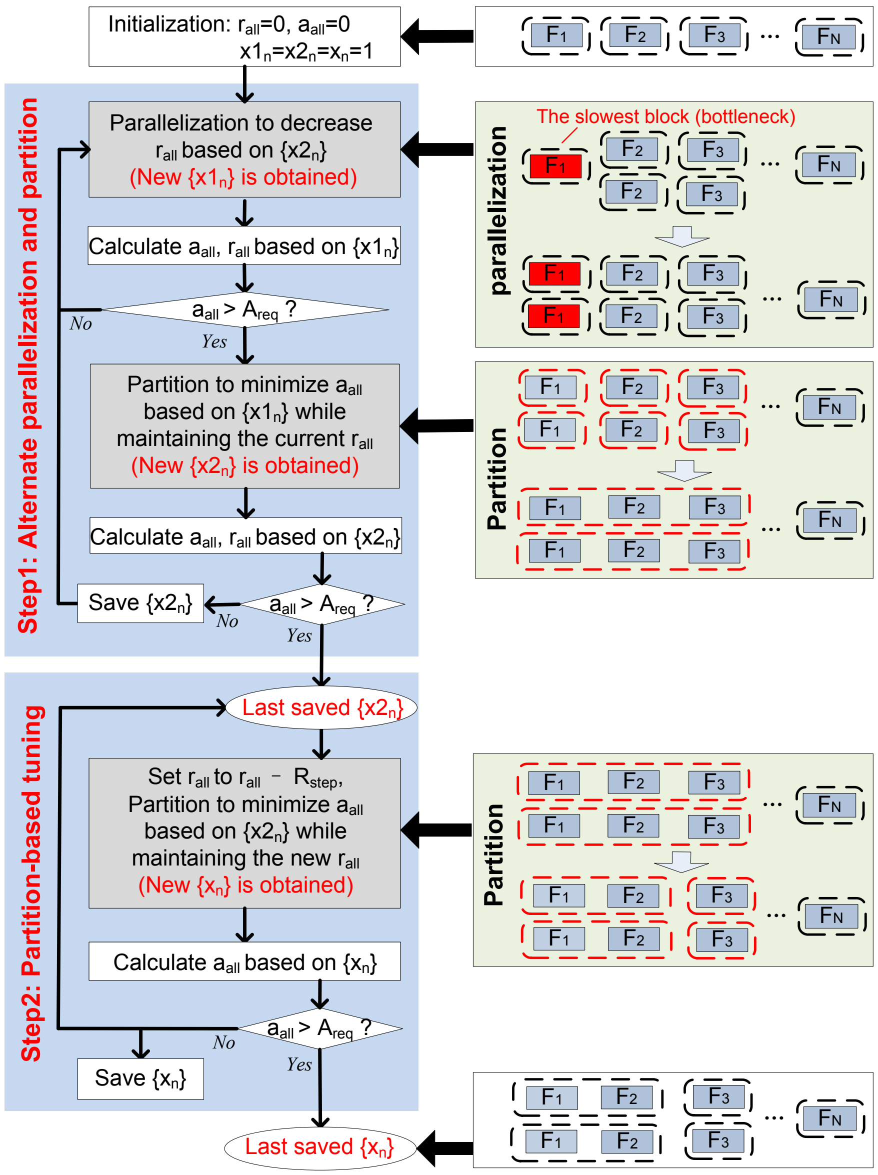
Fig. 10. Framework of the AP+PT algorithm.

The initiation interval (r_{all}) and the area (a_{all}) of the system are initialized to zeros. We use $\{x1_n\}$, $\{x2_n\}$ and $\{x_n\}$ to represent the sets of the current optimized results in different steps. $\{x1_n\}$ is the result of parallelization in Step1. $\{x2_n\}$ and $\{x_n\}$ are the results of partition in Step1 and Step2, respectively. Each variable in $\{x1_n\}$, $\{x2_n\}$ and $\{x_n\}$ is set to one at the beginning, assuming no clustering and parallelization for each function. That means each function is a separated block.

Step1 is AP, which contains two processes: parallelization and partition. During each round of the AP execution, to decrease the initiation interval, the parallelization process first increases the parallelism degree of the current slowest block by one based on the current result $\{x2_n\}$. The parallelization process is repeated until the area constraint (A_{req}) is violated. After that, the result is stored in $\{x1_n\}$. The partition process then performs to minimize the area (a_{all}) based on the current result $\{x1_n\}$, where the initiation interval (r_{all}) obtained by the parallelization process is set as the performance constraint. After the partition process is complete, the result of this round, $\{x2_n\}$, is obtained. A new round then executes. When the partition process cannot keep the area within the area constraint (A_{req}), Step1 is terminated.

The AP step, however, is usually not able to find the optimal result when the solution is already near the optimal one, as the initiation interval cannot be reduced via the area-consuming parallelization under a tight area constraint. Differing from the parallelization process, the partition process supports fine-grained tuning, by providing a gradually reduced initiation interval as the constraint while minimizing the corresponding area overhead. Therefore, we add partition-based tuning in Step2 to further decrease the initiation interval based on the result $\{x2_n\}$ obtained in Step1. During each round of the PT execution, we first reduce the current initiation interval, r_{all}, by R_{step} ($r_{all} = r_{all} - R_{\text{step}}$). R_{step} is a user defined-parameter, which is set to some percentage of r_{all} obtained in Step1. Then, the partition process minimizes the area (a_{all}) based on the result $\{x2_n\}$, while maintaining the reduced initiation interval (r_{all}). After the partition process is complete, the result of this round, $\{x_n\}$, is obtained. A new round then executes. As the initiation interval is smaller, the area of the system becomes larger. When the partition process cannot reduce the initiation interval under the area constraint (A_{req}) any more, this step is terminated and we get the last saved $\{x_n\}$, which is the final result. In addition, different R_{step} mean different converging speeds in PT. We discuss the impact of R_{step} in Section 7.4.2. The proposed algorithm for minimizing area under a constraint of initiation interval is able to adopt the same workflow.

4.3.2. Partition in the Proposed Algorithm. We now present the partition process in detail, which is used in both Step1 and Step2 of the proposed algorithm. The partition process minimizes the area under a given initiation interval. The inputs are the current parallelism degrees, the constraint of initiation interval, and parameters of each function.



Initialization: r_all=0, a_all=0 x1_n=x2_n=x_n=1
Step1: Alternate parallelization and partition
Parallelization to decrease r_all based on {x2_n} (New {x1_n} is obtained)
Calculate a_all, r_all based on {x1_n}
a_all > A_req ?
No
Yes
Partition to minimize a_all based on {x1_n} while maintaining the current r_all (New {x2_n} is obtained)
Calculate a_all, r_all based on {x2_n}
Save {x2_n}
Step2: Partition-based tuning
Last saved {x2_n}
Set r_all to r_all – R_step, Partition to minimize a_all based on {x2_n} while maintaining the new r_all (New {x_n} is obtained)
Calculate a_all based on {x_n}
Save {x_n}
Last saved {x_n}
The slowest block (bottleneck)
parallelization
Partition
F1
F2
F3
FN

Fig. 11. Workflow of the proposed algorithm (obj: $\min r_{\text{all}}$).

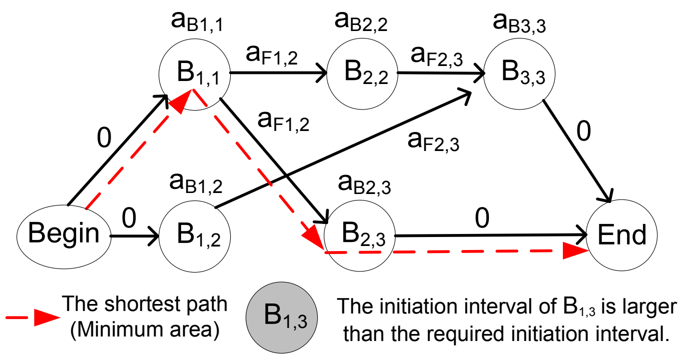
We convert the partition process into a graph-based problem. Thus, the objective is to find the shortest path in a directed acyclic graph (DAG). In Fig. 12, we first calculate the initiation intervals of all the possible blocks clustered by different functions. We then remove those blocks (e.g., $B_{1,3}$), whose initiation interval is larger than the required initiation interval. After that, we construct the DAG as follows. Each node in the DAG represents a possible block. The value on each node corresponds to the total area of the block (e.g., $a_{\text{B}1,2}$), which contains the area overhead of a demux, a mux and a small controller. The area $a_{\text{B1,2}}$ is calculated based on Equation (13) as follows,

$$a_{\text{B1,2}} = (z_{i,j} - y_{i,2}) \cdot AO + z_{1,2} \cdot A_{1,2}. \tag{16}$$

An edge exists when two blocks are adjacent to each other. Each edge is associated with a weight, which is equal to the area of the FIFO between the two blocks (e.g., $a_{\text{F1,2}}$ is the area of the FIFO between block $B_{1,1}$ and $B_{2,2}/B_{2,3}$). The area $a_{\text{F1,2}}$ is calculated based on Equation (14) as follows,

$$a_{\text{F1,2}} = A_{\text{fifo}} \cdot z_{1,2} \cdot S_2^{\text{out}}. \tag{17}$$

Therefore, the shortest path (denoted as the red dashed arrows in the DAG) is equivalent to the minimum area of the system.



a_B1,1
a_B2,2
a_B3,3
a_F1,2
a_F2,3
B1,1
B2,2
B3,3
0
a_F1,2
a_F2,3
0
a_B1,2
a_B2,3
Begin
0
B1,2
B2,3
0
End
The shortest path (Minimum area)
B1,3
The initiation interval of B1,3 is larger than the required initiation interval.

Fig. 12. Directed acyclic graph model for the partition process.

4.3.3. Complexity of the Proposed Algorithm. We now analyze the complexity of the proposed algorithm. For the parallelization process, the complexity is $O(N)$, where N is the number of functions in the system. For the partition process, the complexity in the worst case is $O(N^2)$, which is equal to the complexity of a shortest-path algorithm (e.g., Dijkstra algorithm). However, as some nodes violate the initiation interval constraint (e.g., $B_{1,3}$ in Fig. 12), the complexity is much lower in real-world applications. The total complexity of the algorithm also depends on the number of iterations in the two steps, which is related to the constraint (R_{req} or A_{req}) and the step length for partition-based tuning in Step2 (R_{step}). For minimizing the initiation interval (r_{all}) under the area constraint (A_{req}), the upper bound of the complexity is presented as follows,

$$O((N^2 + N) \cdot \frac{A_{\text{req}}}{\min A_n} + N^2 \cdot \frac{\max T_n}{R_{step}}). \tag{18}$$

$A_{\text{req}}/(\min A_n)$ is the largest number of iterations in Step1, while $(\max T_n)/R_{step}$ is the largest number of iterations in Step2. Compared with the MILP-based solution, the proposed algorithm has a much lower complexity. In addition, the algorithm for minimizing the area under a constraint of initiation interval has the similar complexity.

5. FIFO SIZING

This section focuses on the FIFO sizing problem. We first present the variables and the problem formulation. We then present a parallelism-aware FIFO sizing method to decide the sizes of the FIFOs. Since all the blocks have been synthesized as modules by a C2RTL tool (see the synthesis flow in Section 2.1), we discuss the problem of FIFO sizing at the module level.

5.1. Variables and Problem Formulation

The variables related to FIFO sizing are summarized in Table III. We use w_k to denote the FIFO size between modules M_k and M_{k+1}. Recall that we use r_{all} and a_{all} for the optimized initiation interval and the area in the procedure of simultaneous partition and parallelization, during which all FIFO sizes are estimated by the corresponding bounds as shown in Equation (14). We now use rm_{all} and am_{all} to represent the initiation interval and area after FIFO sizing.

Table III. Variables for FIFO sizing

Variables	Description
w_k	Size of FIFOs between M_k and M_{k+1} $(bytes)$
rm_{all}	Initiation interval of the system under given $\{w_k\}$ $(cycles)$
am_{all}	Area of the system under given $\{w_k\}$ $(LUTs)$

Based on the variables defined in Table III, the FIFO sizing problem for minimizing the area (am_{all}) under the constraint of initiation interval (r_{all}) is written as follows,

$$\begin{aligned} obj: &\quad \min am_{\text{all}} \\ s.t.: &\quad rm_{\text{all}} = r_{\text{all}}. \end{aligned} \tag{19}$$

As the procedure of simultaneous partition and parallelization has been performed, r_{all} and a_{all} are fixed. Therefore, the area of the system (am_{all}) with the given $\{w_k\}$ is calculated as follows,

$$am_{\text{all}} = a_{\text{block}} + A_{\text{fifo}} \cdot (\sum_{k=1}^{k=K-1} w_k), \tag{20}$$

where a_{block} is the area of all the blocks obtained from simultaneous partition and parallelization. A_{fifo} represents the area overhead of FIFOs per byte of data. Considering a FIFO with the largest size (see Equation (14)), we have

$$w_k \leq z_{i,j} \cdot S_j^{out}, \quad i \leq j, \quad i,j \in [1,N]. \tag{21}$$

$z_{i,j} \cdot S_j^{out}$ is the output data size of module M_k (SM_k^{out}), which is synthesized from block $B_{i,j}$. As a_{block} is fixed for FIFO sizing, minimizing the area is equivalent to minimizing the total FIFO size. Therefore, we rewrite the FIFO sizing problem as follows,

$$\begin{aligned} obj: &\quad \min \textstyle\sum_{k=1}^{k=K-1} w_k \\ s.t.: &\quad rm_{\text{all}} = r_{\text{all}}. \end{aligned} \tag{22}$$

5.2. Parallelism-aware FIFO Sizing

Based on the motivation for FIFO sizing in Section 2.2.2, we propose a parallelism-aware FIFO sizing method. Fig. 13 presents the framework of the method. Since the modules synthesized after simultaneous partition and parallelization may have different parallelism degrees, we solve the FIFO sizing problem in two situations:

(1) When the parallelism degrees of two modules are the same, we use an analytical method to calculate the FIFO size. (2) When the parallelism degrees of two modules are different, we develop a simulation-based binary search algorithm to optimize the FIFO size. We explain our method for parallelism-aware FIFO sizing in two steps. First, we describe the method that finds the optimal FIFO size between two modules. Then, we extend it to the multiple-module case.

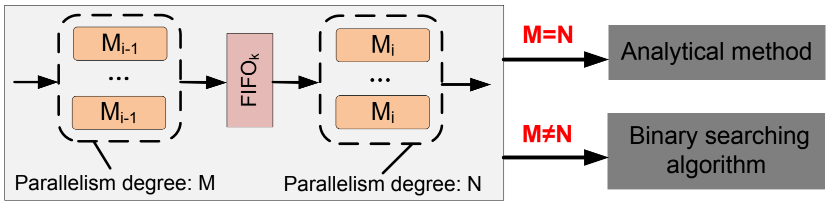


Fig. 13. Framework of the parallelism-aware FIFO sizing method.

5.2.1. Two-module FIFO Sizing. Without loss of generality, we consider the case when the latency of module M_1 is larger than that of module M_2, i.e., $TM_1 > TM_2$ (see Table I in Section 3.3 for the parameters used below). Based on the parallelism degrees of the modules, the algorithm is divided into two parts.

(1) FIFO Sizing with the Same Parallelism Degree. When the parallelism degree of module M_1 is equal to that of module M_2, (i.e., $XM_1 = XM_2$), we classify the FIFO sizing problem into two cases as illustrated in Fig. 14.

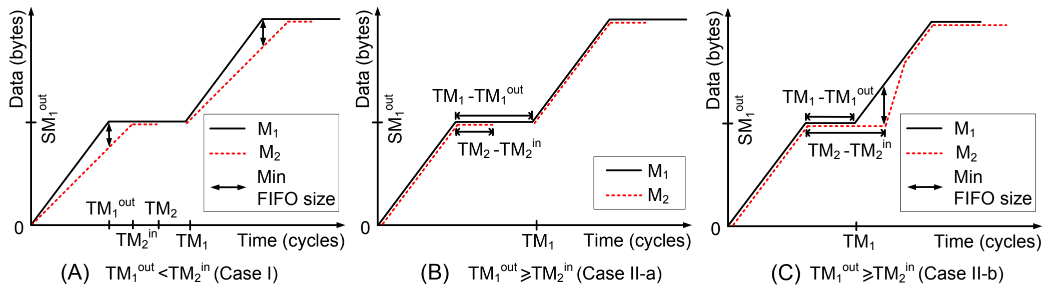


Fig. 14. FIFO sizing between M_1 and M_2 ($TM_1 > TM_2$).

Case I: The latency that M_1 takes to produce data is smaller than the latency that M_2 takes to consume data, i.e., $TM_1^{\text{out}} < TM_2^{\text{in}}$. Fig. 14(A) depicts this scenario. The solid line represents the output data of M_1 with respect to the latency of M_1. The dashed line depicts the data consumed by M_2 with respect to the latency of M_2. The maximum difference between the two data sizes determines the minimum FIFO size. Equation (23) calculates the FIFO size in this case.

$$w_1 = \left\lceil SM_1^{out} \cdot \left(1 - \frac{TM_1^{out}}{TM_2^{in}}\right) \right\rceil. \tag{23}$$

Note that SM_1^{out} is the output data size of M_1. TM_1^{out} and TM_2^{in} denote the output latency of M_1 and the input latency of M_2, respectively.

Case II: The latency that M_1 takes to produce data is no smaller than the latency that M_2 takes to consume the data, i.e., $TM_1^{\text{out}} \geq TM_2^{\text{in}}$. Fig. 14(B) and 14(C) show two different scenarios for this case. Fig. 14(B) shows that M_2 starts at 0 or TM_1 when $TM_2 - TM_2^{\text{in}} \leq TM_1 - TM_1^{\text{out}}$. In this scenario, no FIFO is needed. Fig. 14(C) shows the other case: M_2 cannot start at TM_1 when $TM_2 - TM_2^{\text{in}} > TM_1 - TM_1^{\text{out}}$, because M_2 cannot finish the previous execution at that time. We summarize the FIFO sizes under the two scenarios as follows,

$$w_1 = \begin{cases} 1, \text{if } TM_2 - TM_2^{\text{in}} \leq TM_1 - TM_1^{\text{out}} \\ \left\lceil [(TM_2 - TM_2^{\text{in}}) - (TM_1 - TM_1^{\text{out}})] \cdot \frac{SM_1^{\text{out}}}{TM_1^{\text{out}}} \right\rceil, \text{otherwise.} \end{cases} \tag{24}$$

Therefore, the two-module FIFO sizing problem is solved analytically with a linear complexity.

(2) FIFO Sizing with Different Parallelism Degrees. When $XM_1 \neq XM_2$, the equations presented above no longer hold. We develop a simulation-based binary search algorithm. Different from traditional simulation-based approaches, we only simulate the read and write operations at the interfaces of the modules. Therefore, the logic of the modules is simplified to several counters, and the simplified model greatly shortens the simulation time. Using the parameters in Table I and Table III, the algorithm is presented in Algorithm 1.

ALGORITHM 1: FIFO sizing algorithm for two modules with different parallelism degrees

```
input  : $TM_1$,$TM_2$,$TM_1^{out}$,$TM_2^{in}$,$SM_1^{out}$,$SM_2^{in}$,$XM_1$,$XM_2$, $r_{all}$
output: $w_1$
1  Initialization: $w_1 = SM_1^{out}$, $rm_{all} = r_{all}$, $r_{new} = rm_{all}$, $Upper = Mid = SM_1^{out}$, $Lower = 1$;
2  while $Upper \neq Lower$ do
3      if $r_{new} = rm_{all}$ then
4          $w_1 = ceil((Mid + Lower)/2)$;
5          $Upper = Mid, Mid = w_1$;
6      else
7          $w_1 = ceil((Upper + Mid)/2)$;
8          $Lower = Mid, Mid = w_1$;
9      end
10     $r_{new}$ = Get_R($M_1, M_2, w_1$);
11 end
12 return $w_1$.
```

The inputs are the parameters for modules M_1 and M_2, and the initiation interval obtained after simultaneous partition and parallelization (r_{all}). The parameters of M_1 and M_2 include the total latencies (TM_1 and TM_2), the output and input latencies (TM_1^{out} and TM_2^{in}), the output and input data sizes (SM_1^{out} and SM_2^{in}) and the parallelism degrees (XM_1 and XM_2). The output is the optimal FIFO size (w_1).

Line 1 is the initialization process. The initial FIFO size is set to SM_1^{out} (the maximum value), which is the output data size of M_1. It is big enough to satisfy the initiation interval constraint (r_{all}). rm_{all} is initially set to r_{all}. r_{new} is the current initiation interval. $Upper$, Mid and $Lower$ decide the binary search range, which are set to SM_1^{out}, SM_1^{out} and 1, respectively.

Lines 2-11 are the main loop of the algorithm. In the loop, we first update the value of w_1 and the search range (lines 3-9). Then, we obtain r_{new} (line 10) using function Get_R(). Get_R() determines the initiation interval by simulating the resulting hardware based on the parameters of the two modules and the current FIFO size (w_1). When $Upper = Lower$, the optimal FIFO size is obtained.

As we can see, the most time-consuming part of the algorithm is the Get_R() function. It calls for a simulation of the entire hardware system. To shorten the optimization time, we built a system-level simulator instead of using an RTL-level one. The system-level simulator employs the module parameters and the FIFO size (w_1) to get the current initiation interval (r_{new}).

5.2.2. Multiple-module FIFO Sizing. We transform the K-module FIFO sizing problem into $K-1$ two-module problems (see Fig. 15). Specifically, for the kth FIFO, we treat all modules from M_1 to M_k as one module M_- and all modules behind (from M_{k+1} to M_K) as the other module M_+ (see Fig. 15). Since the slowest modules in M_- and M_+ determine the initiation intervals of M_- and M_+ (TM_- and TM_+), all the modules have to be taken into account.

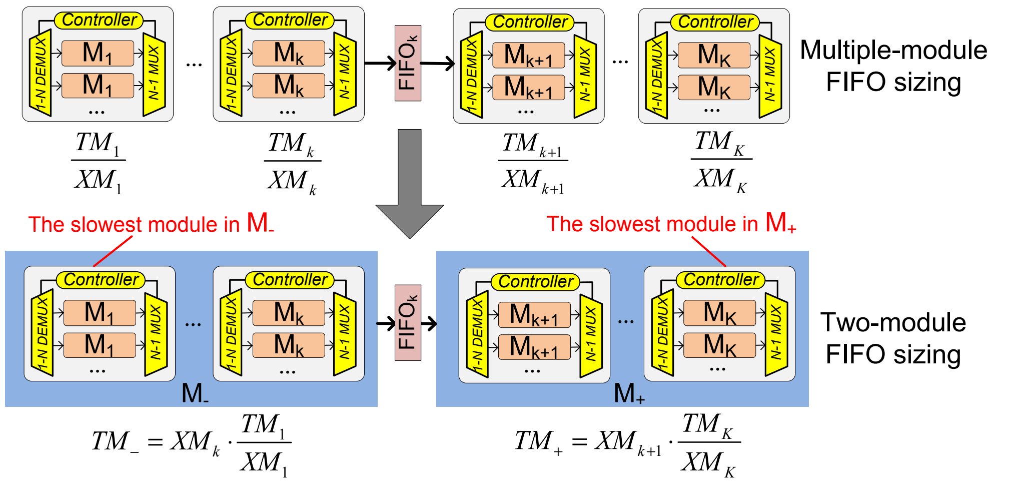


Fig. 15. Transforming the multiple-module FIFO sizing problem into the two-module FIFO sizing problem.

We summarize the approach in Algorithm 2. The algorithm initializes the variables (lines 1-2), where TM_-, TM_+, TM_-^{out}, TM_+^{in}, SM_-^{out} and SM_+^{in} are the parameters of M_- and M_+, respectively. Then, we calculate TM_- and TM_+ for M_- and M_+ (lines 4-5) based on the bottleneck modules, i.e., the modules with the maximum initiation interval among all the modules. TM_-^{out}, TM_+^{in}, SM_-^{out} and SM_+^{in} are set to the parameters of M_k and M_{k+1} (line 6). We calculate the FIFO size by using the two-module method, based on their parallelism degrees (lines 7-11), and get the sizes of all the FIFOs after $K-1$ iterations.

The parallelism-aware FIFO sizing method for FIFO size calculation based on the system model is validated in Section 7.2.

6. EXTENSION AND LIMITATION OF THE SYNTHESIS FLOW

In this section, we introduce some extensions to the synthesis flow, and then present its limitations.

With some additional processing, the hierarchical synthesis flow is able to be exploited for applications with branches and feedback loops. A simple way to handle branches is merging. In Fig. 16, functions F_2 and F_3 in a branch pattern are merged into one function F_{23}, where the input and output parameters of F_{23} are equal to the

ALGORITHM 2: FIFO sizing algorithm for multiple modules (K > 2)

```
input  : $K, \{TM_k\}, \{TM_k^{\text{in}}\}, \{TM_k^{\text{out}}\}, \{SM_k^{\text{in}}\}, \{SM_k^{\text{out}}\}, \{XM_k\}$
output: $\{w_k\}$
Set all $\{w_k\}$ to 1;
Initialization: set $TM_{-/+}$, $TM_-^{out}$, $TM_+^{in}$,$SM_-^{out}$,$SM_+^{in}$ to 0;
for $k = 1$ to $(K-1)$ do
    $TM_- = XM_k \cdot \max\{\frac{TM_1}{XM_1}, \dots \frac{TM_k}{XM_k}\}$;
    $TM_+ = XM_{k+1} \cdot \max\{\frac{TM_{k+1}}{XM_{k+1}}, \dots \frac{TM_K}{XM_K}\}$;
    $TM_-^{out} = TM_k^{out}$, $TM_+^{in} = TM_{k+1}^{in}$, $SM_-^{out} = SM_+^{in} = SM_k^{out}$;
    if $XM_k = XM_{k+1}$ then
        use Equation (23), (24) to calculate $w_k$;
    else
        use simplified simulation-based binary search to get $w_k$ (Algorithm 1);
    end
end
return$\{w_k\}$.
```

maximum values of those in F_2 and F_3. Then, functions F_1, F_{23} and F_4 follow in a streaming pattern.

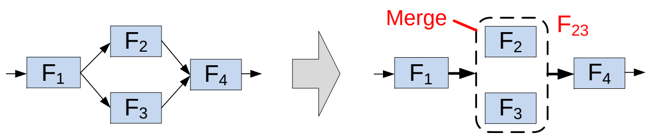
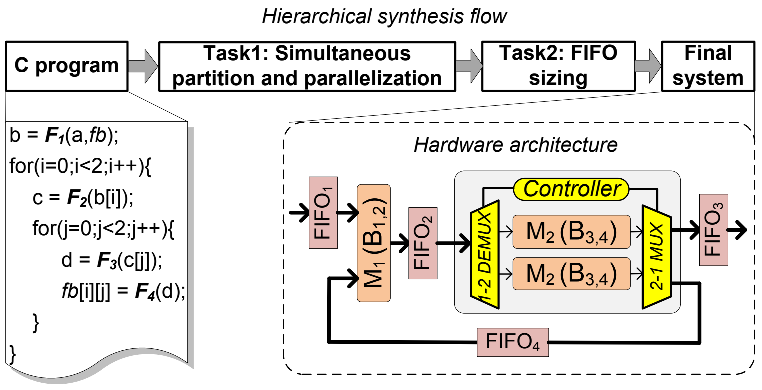


Fig. 16. An example of dealing with the branch pattern.

Fig. 17 presents an example of dealing with the feedback loop pattern based on the hierarchical synthesis flow. The input is a C program, which contains four functions (F_1 to F_4). The optimization procedures execute based on these functions and their data dependencies. $B_{1,2}$ consists of F_1 and F_2 while $B_{3,4}$ consists of F_3 and F_4. The modules M_1 (the synthesized block $B_{1,2}$) and M_2 (the synthesized block $B_{3,4}$) are connected by a 1-to-2 demux, 2-to-1 mux, a small controller and some FIFOs to form the final system.



```
b = F1(a,fb);
for(i=0;i<2;i++){
    c = F2(b[i]);
    for(j=0;j<2;j++){
        d = F3(c[j]);
        fb[i][j] = F4(d);
    }
}
```

Fig. 17. An example of dealing with the feedback loop pattern based on the hierarchical synthesis flow.

To handle the feedback loops, we leverage the modeling effort of a dataflow graph (DFG) and an execution sequence graph (ESG). The DFG describes the data transmission relationships among the functions, while the ESG presents the execution sequence of them. In the DFG, the nodes denote the functions. The edges represent the data dependencies among them. The DFG is then extended to the ESG based on the execution sequence of all the functions in the C program [Edward and David 1987]. In the ESG, a node denotes a single execution for a function. The edges represent the execution sequence between nodes. Fig. 18(A) shows the DFG of the C program in Fig. 17 and Fig. 18(B) shows the corresponding ESG. In the ESG, node F_i^k represents the kth execution for function F_i. The edge from F_i^k to $F_{i'}^{k'}$ exists if F_i^k is executed before $F_{i'}^{k'}$ in the C program. As one function may be executed several times in the application, parallelization means that the initiation interval for each execution is reduced. As the ESG has a streaming pattern, we can apply our proposed methods to get the synthesis result.

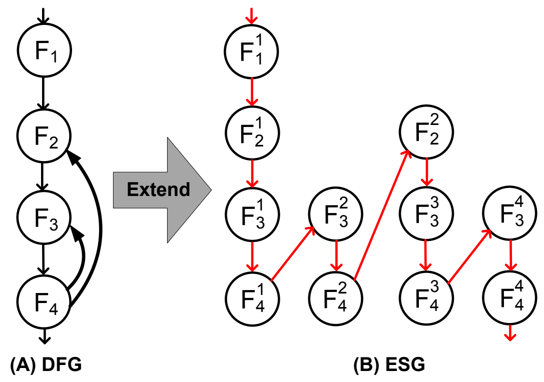


Fig. 18. DFG and ESG for the feedback loop pattern.

In addition, the proposed synthesis flow based on existing C2RTL tools (e.g., eXCite [http://www.yxi.com/ 2013]) has its limitations. For example, dynamic memory allocation, variables with custom bit length and pointers are not supported. Furthermore, recursive algorithms cannot be handled by the synthesis flow. More efforts are needed to remove these limitations.

7. EXPERIMENTAL EVALUATION

In this section, we first explain the experimental setup. After that, we validate the system model. Furthermore, we evaluate the entire framework with several algorithms. Finally, we compare and analyze the AP+PT algorithm, as well as the parallelism-aware FIFO sizing method.

7.1. Experimental Setup

We use a C2RTL tool, eXCite[3] [http://www.yxi.com/ 2013] from Y Explorations Inc. to synthesize all the tasks with automatic optimization options. The generated HDL files are simulated by ModelSim to verify the functionality. The initiation interval is also obtained from the simulation. The area is obtained from Quartus II by Altera Inc.,

[3]Other commercial C2RTL tools (e.g., Vivado HLS [http://www.xilinx.com/ 2015]) are also suitable for synthesis.

where a Cyclone II FPGA is selected as the target hardware. We use *lp_solve* to solve the MILP formulation.

We adopted six large streaming applications from the high-level synthesis benchmark suite CHstone [Hara et al. 2008]. In addition, an application of Filter Group is also added as the seventh benchmark. These benchmarks come from real-world applications and consist of programs from the fields of image processing, security, telecommunication, and digital signal processing. The benchmarks are:

ADPCM. Adaptive differential pulse code modulation is an algorithm for voice compression. (7 functions, 990 lines)

AES encryption/decryption. AES (Advanced Encryption Standard) is a symmetric key cryptosystem. This includes two separate programs (i.e., encryption and decryption). (7 functions, 557 lines/7 functions, 541 lines)

JPEG encoder/decoder. JPEG transforms images between the JPEG and BMP formats. This includes two separate programs (i.e., encoder and decoder). (10 functions, 597 lines/8 functions, 331 lines)

GSM. LPC (Linear Predictive Coding) analysis of GSM (Global System for Mobile Communications) (10 functions, 523 lines)

Filter Group. The group includes two FIR filters, an FFT and an IFFT block. (14 functions, 987 lines)

We set the area-saving factors α_1 and α_2 to 0.2 and -1,000. These values are obtained by data fitting (see Fig. 7) with the method in Section 3.2. The parameter R_{step} is set to 25 and it is used in Step2 of the AP+PT algorithm. Moreover, the unit of initiation interval (r_{all} and R_{req}) is *cycle*, while the units of area (a_{all} and A_{req}) is number of *LUTs* in the FPGA. The unit of FIFO sizes is *byte*.

7.2. System Model Validation

Before comparing algorithms, we validate the calculated results of initiation interval, area and FIFO size based on the system model.

Table IV presents the validation results of initiation interval and area calculation. The second column lists the random values $\{x_n\}$ for different combinations of partition and parallelization. In the following six columns, we compare the model-based initiation interval and area (Model) with the direct synthesized results (Synthesis). According to Equation (2) to (5) and (10) to (14), we calculate the model-based initiation interval and area based on the parameters extracted from eXCite. The direct synthesized results of initiation interval and area are obtained from ModelSim and Quartus II, respectively. Specifically, the errors of initiation interval and area between the model-based results and the synthesized ones are presented in the fifth and eighth columns. They show that the model-based results and the synthesized ones match well. The average error of initiation interval is 4.57%, while that of area is 8.91%. Such differences are reasonable for optimization in high-level synthesis.

Table V presents the validation results of FIFO size calculation. The third column includes the FIFO sizes ($\{w_k\}$) calculated by the proposed parallelism-aware FIFO sizing method (This work) based on the system model and a traditional simulation-based method (Simulation, [Wang et al. 2012]). Since our algorithm uses an approximate method to simplify the optimization procedure, it cannot always get optimal results. The total size of the FIFOs and the difference between the two methods in each benchmark are given in the fourth and fifth columns. They show that the results obtained from the methods match well, where the average difference is only 0.857%. It shows that our proposed method is suitable for FIFO sizing.

Table IV. Validation of initiation interval and area calculation

Benchmark	Simultaneous partition & parallelization results ($\{x_n\}$)	Initiation interval (r_{all})			Area (a_{all})		
		Model	Synthesis	Error	Model	Synthesis	Error
ADPCM	{0,2,0,0,1,1}	197	166	15.7%	17,933	16,624	7.30%
AES decryption	{1,2,0,2,2,0,2}	3,867	3,835	0.828%	35,236	31,036	11.9%
AES encryption	{1,1,1,0,1,0,1}	4,963	4,961	0.0403%	18,014	16,463	8.61%
JPEG decoder	{0,0,0,0,1,0,0,2}	1,766	1,861	5.38%	10,301	8,401	18.4%
JPEG encoder	{2,1,0,1,0,1,0,1,1,2}	419	453	8.11%	17,997	16,309	9.38%
GSM	{0,2,1,2,2,2,2,0,0,1}	961	965	0.416%	21,048	19,708	6.37%
Filter groups	{1,0,0,1,0,0,1,2,2,2,0,1,0,2}	15,291	15,520	1.50%	28,633	28,741	0.377%

Table V. Validation of FIFO size calculation

Benchmark		FIFO size ($\{w_k\}$)	Total size ($bytes$)	Difference
ADPCM	This work	{10,2}	12	0%
	Simulation	{10,2}	12	
AES decryption	This work	{2,2,2,2}	8	0%
	Simulation	{2,2,2,2}	8	
AES encryption	This work	{154,75,63,173}	465	0.868%
	Simulation	{154,73,62,172}	461	
JPEG decoder	This work	{44}	44	0%
	Simulation	{44}	44	
JPEG encoder	This work	{9,13,13,13,23,8}	79	5.33%
	Simulation	{9,12,12,12,22,8}	75	
GSM	This work	{161,82,2,2,2,8}	257	0%
	Simulation	{161,82,2,2,2,8}	257	
Filter group	This work	{94,2,2,2,2,257,1024}	1,383	0.0724%
	Simulation	{93,2,2,2,2,257,1024}	1,382	

7.3. Evaluation for the Entire Framework

We evaluate the entire framework with several algorithms: a traditional C2RTL tool with flat synthesis (eXCite, Baseline), a single FIFO sizing framework [Li et al. 2012a], a method of simultaneous parallelization and partition without FIFO sizing [Li et al. 2013], and our proposed algorithm (This work). For comparison, the final gate netlists are synthesized based on the optimized architecture and they are functionally verified in ModelSim. All the control circuits are using regular structures, which are automatically generated. The initiation interval and area of the hardware are obtained from ModelSim and Quartus II.

Table VI shows the initiation intervals of seven benchmarks obtained from different algorithms under certain area constraints. The maximum frequencies of the Cyclone II FPGA in a DE2-70 board (f_{max}) obtained from a flat synthesis method (Baseline) are presented in the third column. All other algorithms achieve better f_{max} than the baseline based on the synthesized results from eXCite. Compared with the baseline, the proposed algorithm has the minimum initiation interval and achieves a speedup of two or three orders of magnitude in each benchmark. Compared with [Li et al. 2013], the proposed algorithm achieves a speedup of initiation interval by up to 12.8%. Table VII shows the corresponding area of seven benchmarks obtained from these algorithms. Compared with [Li et al. 2013], the proposed algorithm reduces the area by up to 9.46%.

Table VI. Comparison of initiation interval among several algorithms

Benchmark	eXCite (Baseline)		[Li et al. 2012a]		[Li et al. 2013]		This work		Improvement on
	$cycles$	f_{max}(MHz)	$cycles$	Speedup	$cycles$	Speedup	$cycles$	Speedup	[Li et al. 2013]
ADPCM	35,691	53.3	12,464	2.86	167	214	149	240	12.1%
AES decryption	2,185,802	75.6	915,222	2.39	3,892	562	3,640	601	6.93%
AES encryption	1,904,802	71.2	719,263	2.65	4,963	384	4,963	383	0%
JPEG decoder	623,090	71.2	456,821	1.36	1,789	348	1,670	373	7.12%
JPEG encoder	42,475,202	69.7	4,070,603	10.4	4,190	1,014	4,190	1,014	0%
GSM	620,802	55.7	204,356	3.04	936	663	830	748	12.8%
Filter groups	6,537,416	93.4	1,702,406	3.84	20,533	318	19,332	338	6.19%

Table VII. Comparison of area among several algorithms

Benchmark	eXCite ($LUTs$)	[Li et al. 2012a]		[Li et al. 2013]		This work		Reduction on
	(Baseline)	$LUTs$	Overhead	$LUTs$	Overhead	$LUTs$	Overhead	[Li et al. 2013]
ADPCM	12,608	12,889	2.23%	13,416	6.88%	13,366	6.01%	-0.374%
AES decryption	11,609	11,861	2.17%	15,917	37.1%	15,049	29.6%	-5.77%
AES encryption	11,197	11,501	2.72%	16,463	47.0%	15,577	39.1%	-5.68%
JPEG decoder	5,761	5,947	3.23%	8,362	45.7%	8,101	40.6%	-3.22%
JPEG encoder	13,627	14,546	6.74%	16,309	19.7%	16,074	18.0%	-1.46%
GSM	12,672	14,929	17.8%	18,775	48.2%	17,681	39.5%	-6.19%
Filter groups	13,063	13,316	1.90%	19,537	49.6%	17,848	36.6%	-9.46%

In general, our proposed algorithm achieves a huge speedup of initiation interval in each benchmark, while only suffering from a 29.9% area overhead on average, compared with the baseline and [Li et al. 2012a]. It shows the great effectiveness of our algorithms in the hierarchical synthesis flow. In the following two sections, we compare and analyze the algorithms for simultaneous partition and parallelization, as well as FIFO sizing.

7.4. Algorithm Comparison and Analysis for Simultaneous Partition and Parallelization

To compare the algorithms of simultaneous partition and parallelization, we use a single benchmark (the GSM case) with different constraints and multiple benchmarks with certain constraints. The algorithms include: the MILP-based solution for optimal results in Section 4.2 (MILP), which is used as an upper bound; the up-to-date AP algorithm (AP, [Li et al. 2013]) and the proposed AP+PT algorithm (AP+PT, This work).

7.4.1. Comparison of Different Algorithms. First, we compare the algorithms with a single benchmark (the GSM case). Fig. 19 presents two sets of the results under different constraints. In Fig. 19, the MILP-based solution is always superior, with up to 44.5% better results than the AP method in [Li et al. 2013]. However, the MILP-based solution fails to work on two points of Fig. 19(B), as the solver for MILP with those complexities is unstable. Moreover, the results obtained from the proposed AP+PT algorithm are almost the same as those obtained from the MILP-based solution, and are better than those obtained from the AP algorithm.

We then compare the algorithms with seven benchmarks under the different constraints in Fig. 20. In Fig. 20, the results are similar to those in Fig. 19. Compared with the MILP-based solution, the average differences of the optimized results obtained from the AP and AP+PT algorithms are 10.5% and 1.03%. This shows the advantage of the proposed AP+PT algorithm.

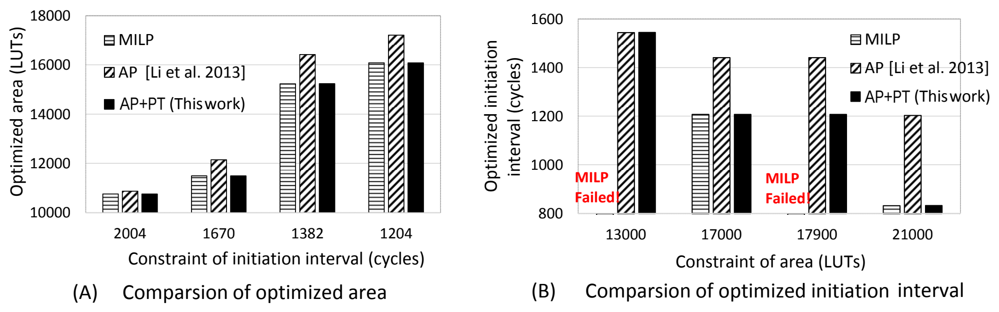


Fig. 19. Comparison of optimization effects with a single benchmark (the GSM case).

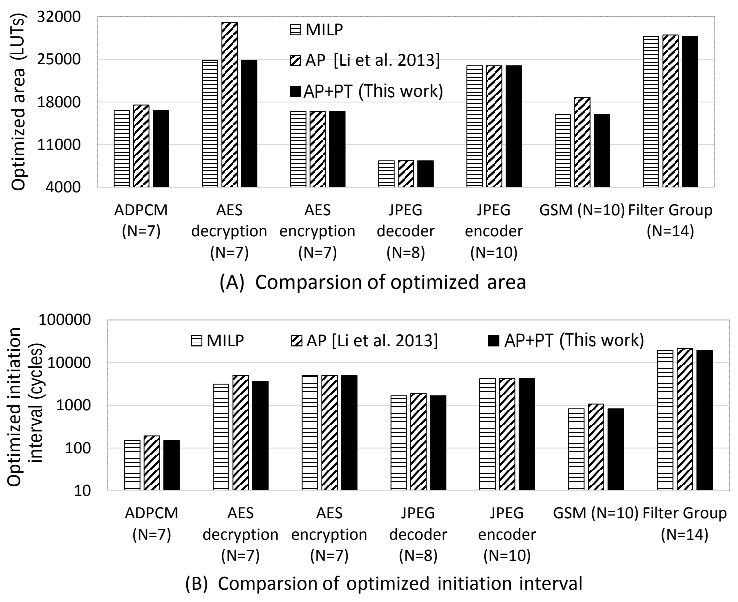
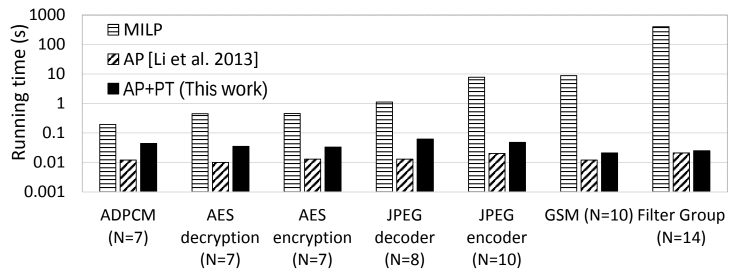


Fig. 20. Comparison of optimization effects with multiple benchmarks.

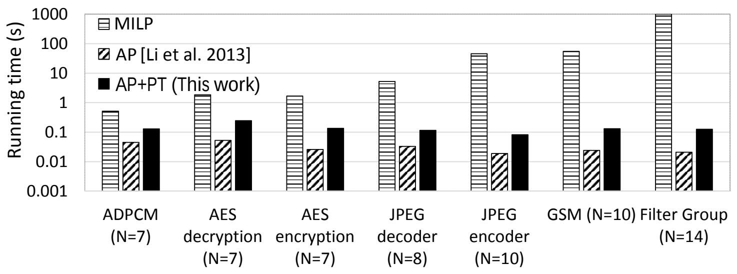
Although the AP and AP+PT algorithms cannot always get the optimal results, their running time is much shorter than that of the MILP-based solution. Fig. 21 presents the running time of different algorithms under the different constraints. The Y-axis is the running time, while the X-axis is the benchmark. Compared with the MILP-based solution, the AP and AP+PT algorithms achieve up to 1,000x speedup in running time. The proposed AP+PT algorithm is a good alternative when the MILP-based solution fails. In reality, a complex system makes the number of functions much larger, where the advantage of the AP+PT algorithm is highlighted.

In general, the proposed AP+PT algorithm, which is almost as good as the optimal MILP-based solution, is better than the AP algorithm. What's more, the proposed AP+PT algorithm has a much lower complexity than the MILP-based solution. Therefore, it is suitable for simultaneous parallelization and partition.

7.4.2. Analysis of the AP+PT Algorithm. Furthermore, we analyze the whole-system and block-level parallelization approaches with the AES decryption case in Fig. 22. The Y-axis is the area. The X-axis is the constraint of the normalized initiation interval



(A) Comparison of running time under constraints of initiation interval



(B) Comparison of running time under constraints of area

Fig. 21. Comparison of running time with multiple benchmarks.

based on the initiation interval of the benchmark without parallelization. Compared with the whole-system parallelization, the block-level parallelization decreases the initiation interval with less area overhead (up to 44.4%), because the block-level approach does not duplicate every block and is able to explore a more fine-grained design space. Obviously, the block-level parallelization makes a tradeoff between the initiation interval and the area more efficiently.

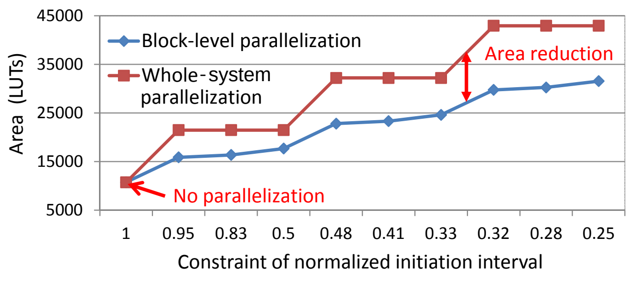


Fig. 22. Initiation interval and area with different parallelization approaches.

In addition, we analyze the impact of R_{step} in Step2 of the proposed AP+PT algorithm with the GSM case. Fig. 23 presents the initiation interval based on the value of R_{step}. The left Y-axis is the initiation interval while the right Y-axis is the number of iterations in Step2. The X-axis shows the value of R_{step}. In Fig. 23, as R_{step} decreases, the initiation interval becomes smaller, while the number of iterations grows. It means that a smaller R_{step} implies more fine-grained optimization, but requires more iterations. However, on certain points (when R_{step} is equal to 50, 25, 10 or 5), the initiation interval remains, as smaller initiation intervals cannot be obtained under the given area constraint. Therefore, too small an R_{step} is useless and results in too many iterations. Thus, users choose their own R_{step} by trading off between the optimality and the number of iterations.

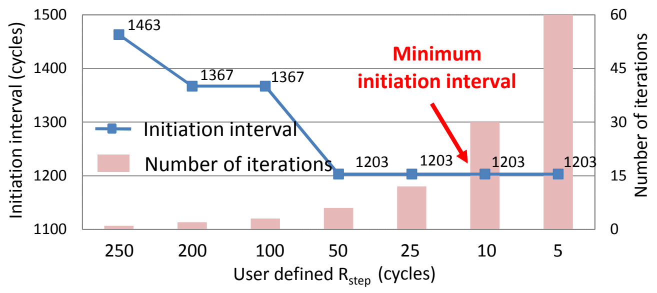


Fig. 23. Impact of R_{step} in Step2 of the AP+PT algorithm.

7.5. Algorithm Comparison and Analysis for FIFO Sizing

7.5.1. Comparison of Different Algorithms. In this section, we compare and analyze the FIFO sizing results. Table VIII presents the FIFO sizing effects of our proposed algorithm with seven benchmarks. The second column lists optimal values $\{x_n\}$ (see the definition in Equation (8a)) after simultaneous partition and parallelization. The area before and after FIFO sizing are given in the third and fourth columns. Compared with the original results before FIFO sizing, our proposed algorithm achieves an area reduction by up to 8.65%. This shows the effectiveness of the algorithm.

Table VIII. FIFO sizing effects with multiple benchmarks

Benchmark	Simultaneous partition & parallelization results ($\{x_n\}$)	Before FIFO sizing a_{all} ($LUTs$)	After FIFO sizing am_{all} ($LUTs$)	Area reduction
ADPCM	{2,0,1,1,0,2}	13416	13366	0.373%
AES decryption	{0,0,2,0,2,0,2}	15917	15049	5.45%
AES encryption	{1,1,1,0,1,0,1}	16463	15577	5.38%
JPEG decoder	{0,0,0,0,1,2,0,1}	8362	8101	3.13%
JPEG encoder	{2,1,0,1,0,1,0,1,1,2}	16309	16074	1.44%
GSM	{2,2,0,2,2,0,2,0,0,1}	18775	17681	5.83%
Filter groups	{0,1,0,0,0,1,1,1,1,1,0,0,1,1}	19537	17848	8.65%

Furthermore, we compare the running time of the parallelism-aware FIFO sizing method (This work) with two methods in Table IX. One is the traditional simulation-based method (Simulation [Wang et al. 2012], Baseline), which is used as the baseline. The other is the analytical method in [Li et al. 2012a]. Compared with the simulation-based method (Baseline), our proposed algorithm (This work) achieves up to an 8,000x speedup in running time. Compared with the analytical method, the proposed algorithm executes under any parallelism degrees of modules. The analytical one fails to work when the parallelism degrees of all the modules are different. In addition, the optimized FIFO sizes of the simulation-based and the proposed algorithms are presented in Table V, where the average difference of the FIFO sizes is only 0.857%. As the analytical method in [Li et al. 2012a] is a part of the proposed algorithm, their optimized FIFO sizes are equal when the parallelism degrees of all the modules are the same.

Table IX. Comparison of running time with multiple benchmarks

Benchmark	Parallelism degrees of all the modules	Simulation [Wang et al. 2012] (Baseline) Time(s)	[Li et al. 2012a]		This work	
			Time(s)	Speedup	Time(s)	Speedup
ADPCM	Different	357	*Failed*	—	0.0542	6,603
AES decryption	Different	3,090	*Failed*	—	1.01	3,059
AES encryption	Same	757	0.0891	8,503	0.0932	8,137
JPEG decoder	Different	256	*Failed*	—	0.145	1,768
JPEG encoder	Different	749	*Failed*	—	0.291	2,572
GSM	Different	1,348	*Failed*	—	1.15	1,171
Filter groups	Same	3,029	16.5	184	18.0	168

7.5.2. Interaction of the FIFO Size and the Performance. Finally, we analyze the relationship between the FIFO size and the performance. Fig. 24 shows the FIFO sizes and the total latency of the JPEG encoder. The Y-axis is the total latency per execution, while the X-axis is the FIFO size (w_1) between the module M_1 and M_2. Based on the different values of the FIFO size (w_2) between the module M_2 and M_3, the latencies obtained by optimizing w_1 have a 6.10% difference. In this case, the optimal FIFO sizes are $w_1 = 44$ and $w_2 = 2$. It shows the superposed influence of different FIFO sizes on the performance.

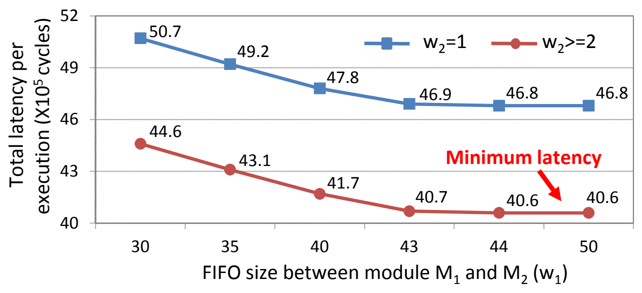


Fig. 24. Interaction of the FIFO size and the performance.

8. RELATED WORK

In this section, we discuss related work in detail. We first show the existing work related to partition and parallelization. After that, the existing work of FIFO sizing is presented.

Many researcher focus on parallelization optimization for streaming applications. [Hagiescu et al. 2009] presented a heuristic algorithm to optimize initiation interval subject to the specified area constraint. The algorithm refined a given stream graph by deciding the number of times each node to be replicated in the graph. This is equivalent to computing the parallelism degree. However, the authors assumed that the partition result was given. In [Cong et al. 2012], an integer linear programming (ILP) method was proposed to select modules in the final design, as well as the parallelism degrees of the selected modules. The method aimed at minimizing area while satisfying the constraint of initiation interval. The authors in [Cong et al. 2014] optimized the parallelization degree together with buffer sizing. [Liu et al. 2012] presented a heuristic technique for finding the parallelization degrees of C-code blocks. A number of other approaches (e.g., [Alexandros et al. 2013a; Alexandros et al. 2013b; Gurumani et al. 2013]) used parallelism descriptions (CUDA) to schedule processing units and optimize computing architectures for the reduction of initiation interval. The work in [Liu et al. 2014] developed a HLS framework that utilizes coarse-grained pipeline parallelism techniques to synthesize specialized accelerator modules from irregular C/C++ programs without requiring any annotations. Though the methods above are effective in performing parallelization, the partitioning part is assumed to be done a priori.

Partition methods in streaming applications have been proposed in a number of papers. [Hara et al. 2010] introduced an ILP-based method to cluster small C functions into blocks under the constraints of performance and area. [Schafer and Wakabayashi 2012] developed a "divide-and-conquer" exploration algorithm (DC-ExpA) to accelerate the design space exploration of given behavioral descriptions (C/SystemC) for high-level synthesis. [Schafer 2013] presented a method for automatically partitioning single-process behavioral descriptions (ANSI-C or SystemC) into separate processes under a given global constraint of initiation interval. [Cong et al. 2011a; Li et al. 2012b; Wang et al. 2013; Wang et al. 2014] presented memory partitioning in high-level synthesis with scheduling by fine-grained modeling and loop optimization. However, their method does not consider block-level parallelization. In summary, none of the existing synthesis work above adequately explore the design space defined by both C-code partition and block-level parallelization in streaming applications.

Partition and parallelization also find their usefulness in mapping multiprocessor systems-on-chip (MPSoC) solutions. For example, in [Haris and Sri 2008], JPEG code was manually partitioned into stages and separately mapped onto heterogeneous processors. In [Ceng et al. 2008], a parallelism C-compiler framework (MAPS) was presented that mapped sequential C programs onto a given heterogeneous MP-SoC platform. A non-speculative compiler parallelization technique was proposed in [Raman et al. 2008] for homogeneous MPSoC. [Hormati et al. 2009] used an ILP formulation to map nodes in stream graphs onto a heterogeneous MPSoC. [Kwon and Ha 2010] used serialized parallelism techniques for efficient code generation on the MPSoC systems. Authors in [Cordes et al. 2011] presented an ILP algorithm, which automatically extracts pipeline parallelism from sequential ANSI-C applications. Though some of the existing MPSoC mapping methods, i.e., software compiling tasks, are capable of performing automatic partitioning, the goal is to identify inherent parallelism within a given program, which is different from the hardware synthesis task that we are interested in.

To design a streaming application, researchers have also investigated input streaming rates to make sure that the FIFOs between modules do not overflow, while meeting the requirements of initiation interval. On-chip traffic analysis of a given SoC architecture was performed in [Lahiri et al. 2001]. However, their simulation-based approach suffered from long executing time and often failed in exploring large design spaces. Based on network calculus, [Maxiaguine et al. 2004] extended the service curves to show how to shape an input stream to meet buffer constraints. Furthermore, [Liu et al. 2006] discussed generalized rate analysis for multimedia processing platforms. [Wiggers et al. 2008] proposed a complex algorithm for buffer size computation that satisfied the constraint of initiation interval for an H.263 video decoder. In this work, the task graph of the application was considered. [Wang et al. 2012] proposed a medium-grained, "access-contiguous" buffer allocation scheme, that minimized total buffer space and pointer overhead. However, the methods above adopt more complicated behavior models for streaming applications, which are not necessary in the hierarchical C2RTL framework. Moreover, [Zhu et al. 2009] presented a two-step approach to minimize the buffer size in a synchronous data flow (SDF) under initiation interval constraints on a hybrid CPU/FPGA structure. Furthermore, [Chen and Zhou 2012] solved the buffer sizing problem by considering simultaneously processor mapping and pipeline stage assignment with a two-level heuristic algorithm, in which a dynamic programming solution with cubic complexity is used. Although their buffer sizing formulation is similar to ours, their approaches require more dimensions to be considered, such as processor mapping, pipeline stage assignment, and hybrid processor architecture. On the contrary, the simpler structure of high-level synthesis in [Li et al. 2012a] led to a more efficient analytical method for buffer sizing, and achieved a linear complexity. However, the method only works for special cases when no block-level parallelization is considered. Thus, the method is not suitable for parallelism-aware FIFO sizing.

9. CONCLUSIONS

In this paper, we study the development of hardware accelerators for streaming applications given as C (or its variants) programs. We adopt a hierarchical synthesis flow, aiming at decreasing the initiation interval and the area of the C2RTL synthesized system. The synthesis flow consists of two tasks: (i) simultaneous partition and parallelization, and (ii) FIFO sizing. They are used to produce the most desirable synthesized results. We propose an MILP-based solution to find the optimal result of simultaneous partition and parallelization for a given C program. This method is capable of minimizing initiation interval or area while satisfying user-specified constraints. We also devise a heuristic algorithm to solve the optimization problem, as the MILP-based solution is too costly for large-scale problems. Moreover, we develop a parallelism-aware FIFO sizing method to determine the FIFO sizes between the adjacent hardware modules with different data rates. The method minimizes the area of the FIFOs while maintaining the desired initiation interval.

Experimental results obtained from seven applications show that our algorithms are effective. Specifically, the heuristic algorithm achieves the optimal results in most cases with less than one second running time. In addition, the method for parallel-aware FIFO sizing achieves as much as 8.65% area savings within just several seconds.

REFERENCES

P. Alexandros, D. Chen, W. Hwu, J. Cong, and Y. Liang. 2013a. Throughput-oriented Kernel Porting onto FPGAs. In *Proceedings of the Annual Design Automation Conference on Design Automation Conference (DAC'13)*. 1–10.

P. Alexandros, G. Karthik, S. John A, D. Chen, J. Cong, and H. Wen-Mei W. 2013b. Efficient Compilation of CUDA Kernels for High-performance Computing on FPGAs. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 2 (2013), 25:1–26.

J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. 2008. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *Proceedings of the Annual Design Automation Conference (DAC'08)*. 754–759.

Y. Chen and H. Zhou. 2012. Buffer minimization in pipelined SDF scheduling on multi-core platforms. In *17th Asia and South Pacific Design Automation Conference (ASPDAC'12)*. 127–132.

J. Cong, K. Guruaj, M. Huang, S. Li, B. Xiao, and Y. Zou. 2011. Domain-specific Processor with 3D Integration for Medical Image Processing. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'11)*. 247–250.

J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou. 2012. Combining Module Selection and Replication Throughput-Driven Streaming Programs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'12)*. 1018–1023.

J. Cong, M. Huang, and P. Zhang. 2014. Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications. In *Proceedings of the ACM/SIGDA international symposium on Field-programmable gate arrays (FPGA'14)*. 213–222.

J. Cong, W. Jiang, B. Liu, and Y. Zou. 2011a. Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 16, 2 (2011), 15:1–25.

J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011b. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 30, 4 (2011), 473–491.

D. Cordes, A. Heinig, P. Marwedel, and A. Mallik. 2011. Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming. In *International Conference on Parallel and Distributed Systems (ICPADS'11)*. 699–706.

L. Edward and M. David. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

Y. Guo and D. McCain. 2006. Rapid Prototyping and VLSI Exploration for 3g/4G MIMO Wireless Systems Using Integrated Catapult-c Methodology. In *IEEE Wireless Communications and Networking Conference (WCNC'06)*. 958–963.

S. T. Gurumani, C. Hisham, Y. Liang, R. Kyle, and D. Chen. 2013. High-level Synthesis of Multiple Dependent CUDA Kernels on FPGA. In *Asia and South Pacific Design Automation Conference (ASPDAC'13)*. 305–312.

A. Hagiescu, W. Wong, D. F. Bacon, and R. Rabbah. 2009. A Computing Origami: Folding Streams in FPGAs. In *Proceedings of the Annual Design Automation Conference (DAC'09)*. 282–287.

Y. Hara, H. Tomiyama, S. Honda, and H. Takada. 2010. Partitioning of Behavioral Descriptions with Exploiting Function-Level Parallelism. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (IEICE T FUND ELECTR)* E93-A (2010), 488–499.

Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. 2008. CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis. In *IEEE International Symposium on Circuits and Systems (ISCAS'08)*. 1192–1195.

J. Haris and P. Sri. 2008. Synthesis of Heterogeneous Pipelined Multiprocessor Systems Using ILP: Jpeg Case Study. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'08)*. 1–6.

A. H. Hormati, C. Yoonseo, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. 2009. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 214–223.

Xilinx http://www.xilinx.com/. 2015. Vivado High-Level Synthesis.

YXI http://www.yxi.com/. 2013. YXI's eXCite tool.

R. Iyer. 2012. Accelerator-rich Architectures: Implications, Opportunities and Challenges. In *Asia and South Pacific Design Automation Conference (ASPDAC'12)*. 106–107.

S. Kwon and S. Ha. 2010. Serialized Parallel Code Generation Framework for MPSoC. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 15, 2 (2010), 11:1–27.

K. Lahiri, A. Raghunathan, and S. Dey. 2001. System-level Performance Analysis for Designing On-chip Communication Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 20, 6 (2001), 768–783.

P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong. 2012b. Memory Partitioning and Scheduling Co-optimization in Behavioral Synthesis. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'12)*. 488–495.

S. Li, Y. Liu, X. Hu, X. He, Y. Zhang, P. Zhang, and H. Yang. 2013. Optimal Partition with Block-level Parallelization in C-to-rtl Synthesis for Streaming Applications. In *Asia and South Pacific Design Automation Conference (ASPDAC'13)*. 225–230.

S. Li, Y. Liu, D. Zhang, X. He, P. Zhang, and H. Yang. 2012a. A Hierarchical C2RTL Framework for FIFO Connected Stream Applications. In *Asia and South Pacific Design Automation Conference (ASPDAC'12)*. 133–138.

F. Liu, G. Soumyadeep, N. P. Johnson, and D. I. August. 2014. CGPA: Coarse-Grained Pipelined Accelerators. In *Proceedings of the Annual Design Automation Conference on Design Automation Conference (DAC'14)*. 1–6.

Y. Liu, S. Chakraborty, and R. Marculescu. 2006. Generalized Rate Analysis for Media-processing Platforms. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. 305–314.

Y. Liu, S. Li, H. Yang, and P. Zhang. 2012. A Hierarchical C2RTL Framework for Hardware Configurable Embedded Systems. In *Embedded Systems - Theory and Design Methodology*. Intech, 367–386.

A. Maxiaguine, S. Künzli, S. Chakraborty, and L. Thiele. 2004. Rate Analysis for Streaming Applications with On-chip Buffer Constraints. In *Asia and South Pacific Design Automation Conference (ASPDAC'04)*. 131–136.

S. McConnell. 2009. *Code Complete*. O'Reilly Media, Inc.

M. A. Pasha, S. Derrien, and O. Sentieys. 2012. System-Level Synthesis for Wireless Sensor Node Controllers: A Complete Design Flow. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17, 1 (2012), 2:1–24.

E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. 2008. Parallel-stage Decoupled Software Pipelining. In *Proceedings of the Annual IEEE/ACM international symposium on Code generation and optimization (CGO'08)*. 114–123.

M. Rossler, H. Wang, U. Heinkel, N. Engin, and W. Drescher. 2009. Rapid Prototyping of A DVB-SH Turbo Decoder Using High-level-synthesis. In *International Conference on Field Programmable Logic and Applications (FDL'09)*. 1–6.

B. C. Schafer. 2013. Automatic Partitioning of Behavioral Descriptions for High-Level Synthesis with Multiple Internal Throughputs. In *Electronic System Level Synthesis Conference (ESLsyn'13)*. 1–6.

B. C. Schafer, A. Trambadia, and K. Wakabayashi. 2010. Design of Complex Image Processing Systems in ESL. In *Asia and South Pacific Design Automation Conference (ASPDAC'10)*. 809–814.

B. C. Schafer and K. Wakabayashi. 2012. Divide and Conquer High-level Synthesis Design Space Exploration. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17, 3 (2012), 29:1–19.

A. P. Wang, J. Hahn, M. Roumi, and P. H. Chou. 2012. Buffer Optimization and Dispatching Scheme for Embedded Systems with Behavioral Transparency. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17, 4 (2012), 41:1–26.

Y. Wang, P. Li, and J. Cong. 2014. Theory and Algorithm for Generalized Memory Partitioning in High-level Synthesis. In *Proceedings of the ACM/SIGDA international symposium on Field-programmable gate arrays (FPGA'14)*. 199–208.

Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. 2013. Memory Partitioning for Multidimensional Arrays in High-level Synthesis. In *Proceedings of the Annual Design Automation Conference (DAC'13)*. 1–8.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. 2008. Buffer Capacity Computation For Throughput Constrained Streaming Applications With Data-Dependent Inter-Task Communication. In *Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*. 183–194.

J. Zhu, I. Sander, and A. Jantsch. 2009. Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'09)*. 1506–1511.

Y. Zhu, Y. Liu, D. Zhang, S. Li, P. Zhang, and T. Hadley. 2010. Acceleration of Pedestrian Detection Algorithm on Novel C2RTL HW/SW Co-design Platform. In *International Conference on Green Circuits and Systems (ICGCS'10)*. 615–620.

Received November 2014; revised January 2015; accepted June 2015