We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

17

A Hierarchical C2RTL Framework for Hardware Configurable Embedded Systems

Yongpan Liu[1], Shuangchen Li[1], Huazhong Yang[1] and Pei Zhang[2]
[1]*Tsinghua University, Beijing,*
[2]*Y Explorations, Inc., San Jose, CA*
[1]*P.R.China*
[2]*USA*

1. Introduction

Embedded systems have been widely used in the mobile computing applications. The mobility requires high performance under strict power consumption, which leads to a big challenge for the traditional single-processor architecture. Hardware accelerators provide an energy efficient solution but lack the flexibility for different applications. Therefore, the hardware configurable embedded systems become the promising direction in future. For example, Intel just announced a system on chip (SoC) product, combining the ATOM processor with a FPGA in one package (Intel Inc., 2011).

The configurability puts more requirements on the hardware design productivity. It worsens the existing gap between the transistor resources and the design outcomes. To reduce the gap, design community is seeking a higher abstraction rather than the register transfer level(RTL). Compared with the manual RTL approach, the C language to RTL (C2RTL) flow provides magnitudes of improvements in productivity to better meet the new features in modern SoC designs, such as extensive use of embedded processors, huge silicon capacity, reuse of behavior IPs, extensive adoption of accelerators and more time-to-market pressure. Recently, people (Cong et al., 2011) observed a rapid rising demand for the high quality C2RTL tools.

In reality, designers have successfully developed various applications using C2RTL tools with much shorter design time, such as face detection (Schafer et al., 2010), 3G/4G wireless communication (Guo & McCain, 2006), digital video broadcasting (Rossler et al., 2009) and so on. However, the output quality of the C2RTL tools is inferior to that of the human-designed ones especially for large behavior descriptions. Recently, people proposed more scalable design architectures including different small modules connected by first-in first-out (FIFO) channels. It provides a natural way to generate a design hierarchically to solve the complexity problem.

However, there exist several major challenges of the FIFO-connected architecture in practice. First of all, the current tools leave the user to determine the FIFO capacity between modules, which is nontrivial. As shown in Section 2, the FIFO capacity has a great impact on the system performance and memory resources. Though determining the FIFO capacity via extensive

RTL-level simulations may work for several modules, the exploration space will become prohibitive large in the multiple-module case. Therefore, previous RTL-level simulating method is neither time-efficient nor optimal. Second, the processing rate among modules may bring a large mismatch, which causes a serious performance degradation. Block level parallelism should be introduced to solve the mismatches between modules. Finally, the C program partition is another challenge for the hierarchical design methodology.

This chapter proposed a novel C2RTL framework for configurable embedded systems. It supports a hierarchical way to implement complex streaming applications. The designers can determine the FIFO capacity automatically and adopt the block level parallelism. Our contributions are listed as below: 1) Unlike treating the whole algorithm as one module in the flatten design, we cut the complex streaming algorithm into modules and connect them with FIFOs. Experimental results showed that the hierarchical implementation provides up to 10.43 times speedup compared to the flatten design. 2) We formulate the parameters of modules in streaming applications and design a behavior level simulator to determine the optimal FIFO capacity very fast. Furthermore, we provide an algorithm to realize the block level parallelism under certain area requirement. 3) We demonstrate the proposed method in seven real applications with good results. Compared to the uniform FIFO capacity, our method can save memory resources by 14.46 times. Furthermore, the algorithm can optimize FIFO capacity in seconds, while extensive RTL level simulations may need hours. Finally, we show that proper block level parallelism can provide up to 22.94 times speedup in performance with reasonable area overheads.

The rest of the chapter is organized as follows. Section 2 describes the motivation of our work. We present our model framework in Section 3. The algorithm for optimal FIFO size and block level parallelism is formulated in Section 4 and 5. Section 6 presents experimental results. Section 7 illustrates the previous work in this domain. Section 8 concludes this paper.

2. Motivation

This section provides the motivation of the proposed hierarchical C2RTL framework for FIFO-connected streaming applications. We first compare the hierarchical approach with the flatten one. And then we point out the importance of the research of block level parallelism and FIFO sizing.

2.1 Hierarchical vs flatten approach

The flatten C2RTL approach automatically transforms the whole C algorithm into a large module. However, it faces two challenges in practice. 1) The translating time is unacceptable when the algorithm reaches hundreds of lines. In our experiments, compiling algorithms over one thousand lines into the hardware description language (HDL) codes may lead to several days to run or even failed. 2) The synthesized quality for larger algorithms is not so good as the small ones. Though the user may adjust the code style, unroll the loop or inline the functions, the effect is usually limited.

Unlike the flatten method, the hierarchical approach splits a large algorithm into several small ones and synthesizes them separately. Those modules are then connected by FIFOs.

It provides a flexible architecture as well as small modules with better performance. For example, we synthesized the JPEG encode algorithm into HDLs using eXCite (Y Exploration Inc., 2011) directly compared to the proposed solution. The flatten one costs 42'475'202 clock cycles with a max clock frequency of 69.74MHz to complete one computation, while the hierarchical method spends 4'070'603 clock cycles with a max clock frequency of 74.2MHz. It implies a 10.43 times performance speedup and a 7.2% clock frequency enhancement.

2.2 Performance with different block number

Among multiple blocks in a hierarchical design, there exist processing rate mismatches. It will have a great impact on the system performance. For example, Figure 1 shows the IDCT module parallelism. It is in the slowest block in the JPEG decoder. The JPEG decoder can be boosted by duplicating the IDCT module. However, block level parallelism may lead to nontrivial area overheads. It should be careful to find a balance point between the area and the performance.

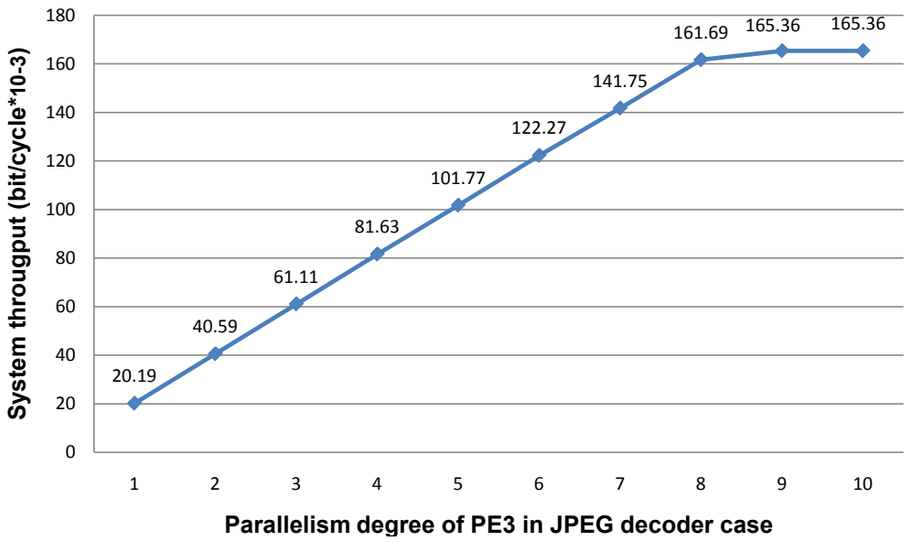


Fig. 1. System throughput under different parallelism degrees

2.3 Performance with different FIFO capacity

What's more, determining the FIFO size becomes relevant in the hierarchial method. We demonstrate the clock cycles of a JPEG encoder under different FIFO sizes in Figure 2. As we can see, the FIFO size will lead to an over 50% performance difference. It is interesting to see that the throughput cannot be boosted after a threshold. The threshold varies from several to hundreds of bits for different applications as described in Section 6. However, it is impractical to always use large enough FIFOs (several hundreds) due to the area overheads. Furthermore, designers need to decide the FIFO size in an iterative way when exploring different function partitions in the architecture level. Considering several FIFOs in a design, the optimal FIFO sizes may interact with each other. Thus, determining the proper FIFO size accurately and efficiently is important but complicated. More efficient methods are preferred.

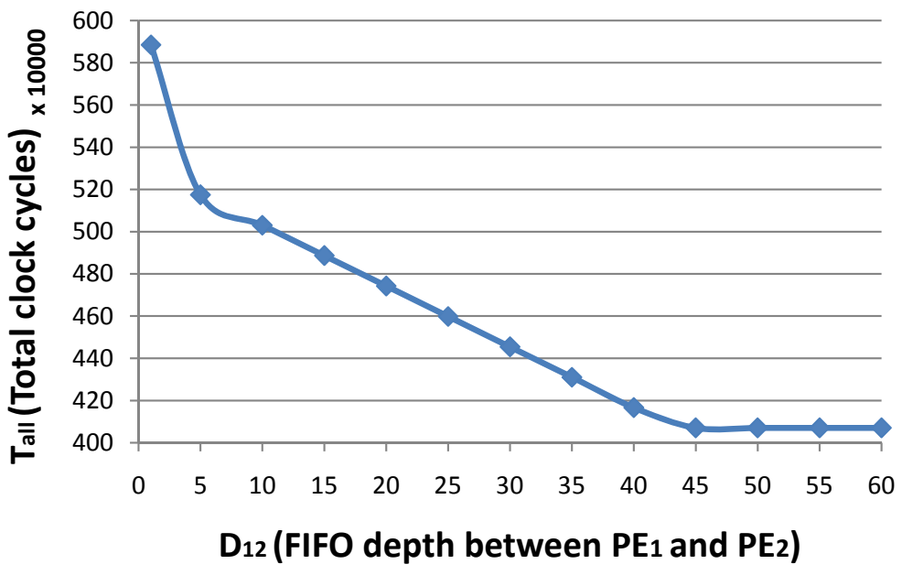


Fig. 2. Computing cycles under different FIFO sizes

3. Hierarchical C2RTL framework

This section first shows the diagram of the proposed hierarchical C2RTL framework. We then define four major stages: function partition, parameter extraction, block level parallelism and FIFO interconnection.

3.1 System diagram

The framework consists of four steps in Figure 3. In Step 1, we partition C codes into appropriate-size functions. In Step 2, we use C2RTL tools to transform each function into a hardware process element (PE), which has a FIFO interface. We also extract timing parameters of each PE to evaluate the partition in Step 1. If a partition violates the timing constraints, a design iteration will be done. In Step 3, we decide which PEs should be parallelized as well as the parallelism degree. In Step 4, we connect those PEs with proper sized FIFOs. Given a large-scale streaming algorithm, the framework will generate the corresponding hardware module efficiently. The synthesizing time is much shorter than that in the flatten approach. The hardware module can be encapsulated as an accelerator or a component in other designs. Its interface supports handshaking, bus, memory or FIFO. We denote several parameters for the module as below: the number of PEs in the module as N, the module's throughput as TH_{all}, the clock cycles to finish one computation as T_{all}, the clock frequency as CLK_{all} and the design area as A_{all}.

As C2RTL tools can handle the small-sized C codes synthesis (Step 2) efficiently, four main problems exist: how to partition the large-scale algorithm into proper-sized functions (Step 1), what parameters to be extracted from each PE(In Step 2), how to determine the parallelized PEs and their numbers (Step 3) and how to decide the optimal FIFO size between PEs (Step 4). We will discuss them separately.

3.2 Function partition

The C code partition greatly impacts the final performance. On one hand, the partition will affect the speed of the final hardware. For example, a very big function may lead to a very slow PE. The whole design will be slowed down, since the system's throughput is decided by the slowest PE. Therefore, we need to adjust the slowest PE's partition. The simplest method is to split it into two modules. In fact, we observe that the ideal and most efficient partition leads to an identical throughput of each PE. On the other hand, the partition will also affect the

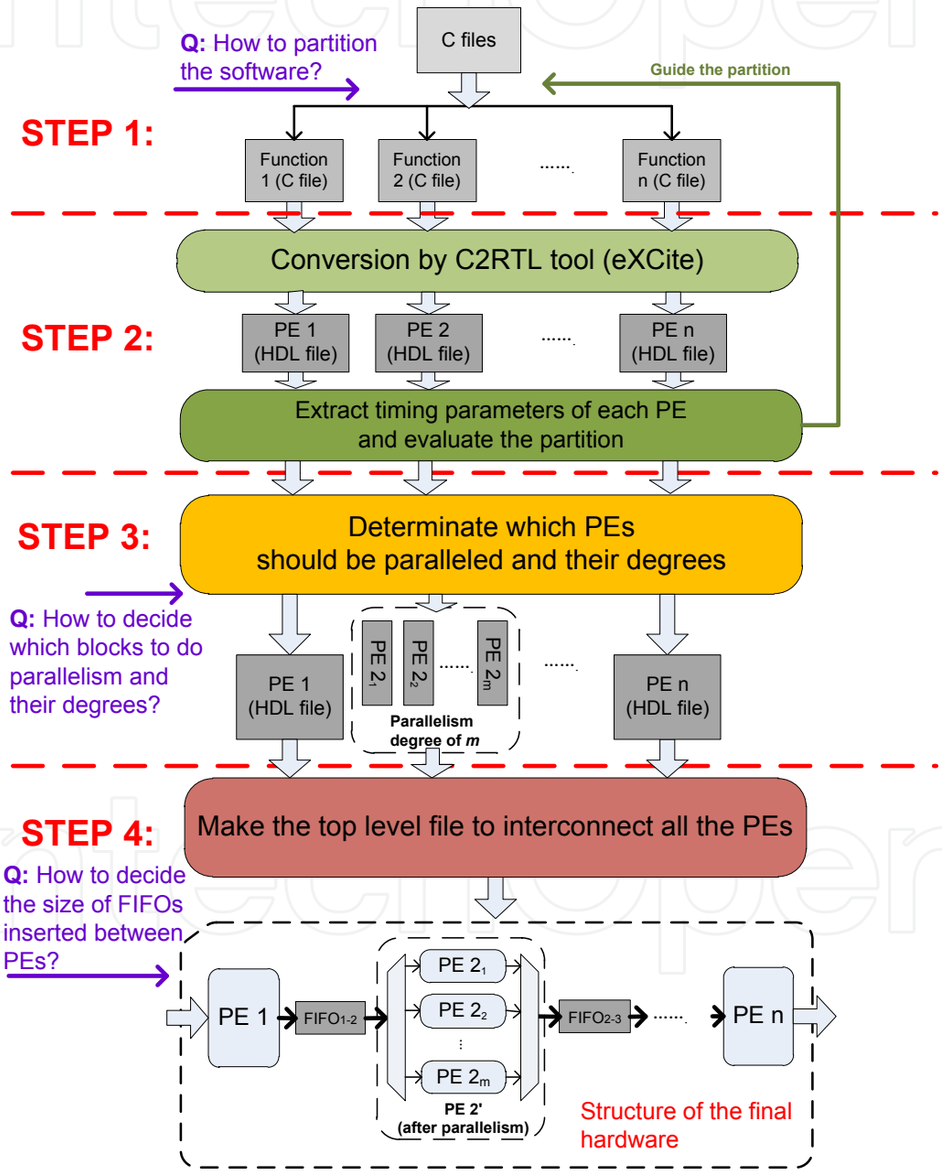


Fig. 3. Hierarchical C2RTL Flow

Name	Description	Examples[2]
Type	Interface type,I or II	II
$TH_{ni/o}$	Throughput of input or output interface	0.0755
$t_{ni/o}$	Input or output time in T_n (cycles)	128
T_n	Period of PE_n (cycles)	848
A_n	Area of PE_n (LE)	4957
f_n	$TH_{no}/TH_{ni/i}$	1
SoP_n(m)[1]	State of PE_n at m^{th} cycle	
	0:Processing;1:Reading; 2:Writing;3:Reading and writing	

[1] m means m^{th} cycle.
[2] Output of PE_2 in the JPEG encode case, as shown in Figre 4

Table 1. The parameter of the n^{th} PE's input/output interfaces

area. Too fine-grained partitions lead to many independent PEs, which will not only reduce the resource sharing but also increase the communication costs.

In this design flow, we use a manual partition strategy, because no timing information in C language makes the automatic partition difficult. In this framework, we introduce an iterative design flow. Based on the timing parameters[1] extracted by the PEs from the C2RTL tools, the designers can determine the C code partition. However, automatizing this partition flow is an interesting work which will be addressed in our future work.

3.3 Parameter extraction

We get the PE's timing information after the C2RTL conversion. In streaming applications, each PE has a working period T_n, under which the PE will never be stopped by overflows or underflows of an FIFO. During the period T_n, the PE will read, process, and write data. We denote the input time as t_{ni} and the output time as t_{no}. In summary, we formulate the parameters of the n^{th} PE interface in Table 1. Based on a large number of PEs converted by *eXCite*, we have observed two types of interface parameters. Figure 4 shows the waveform of the type II. As we can see, t_n is less than T_n in this case. In type I, t_n equals to T_n, which indicates the idle time is zero.

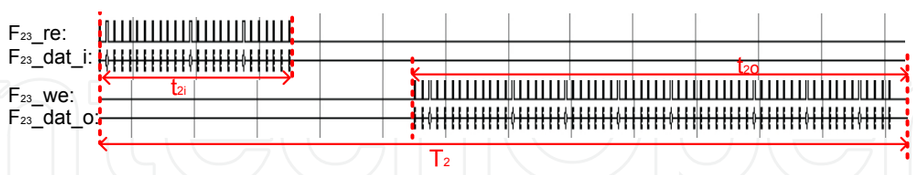


Fig. 4. Type II case: Output of PE_2 in the JPEG encoder

3.4 Block level parallelism

To implement block level parallelism, we denote the n^{th} PE's parallelism degree as P_n.[2] Thus, P_n=1 means that the design does not parallelize this PE. When $P_n > 1$, we can implement block level parallelism using a MUX, a DEMUX, and a simple controller in Figure 5.

[1] We will define those parameters in the next section.
[2] We assume that no data dependence exists among PE_n's task.

Figure 6 illustrates the working mechanism of the n^{th} parallelized PE. It shows a case with two-level block parallelism with $t_{ni}>t_{no}$. In this case, the input and the output of the parallelized blocks work serially. It means that the PE_{n_2} block must be delayed for t_{ni} by the controller, so as to wait for the PE_{n_1} to load its input data. However, when another work period T_n starts, the PE_{n_1} can start its work immediately without waiting for the PE_{n_2}.

As we can see, the interface of the new PE_n after parallelism remains the same as Table 1. However, the values of the input and the output parameters should be updated due to the parallelism. It will be discussed in Section 4.2.

3.5 FIFO interconnection

To deal with the FIFO interconnection, we first define the parameters of a FIFO. They will be used to analyze the performance in the next section. Figure 7 shows the signals of a FIFO. F_clk denotes the clock signal of the FIFO F. F_we and F_re denote the enable signals of writing and reading. F_dat_i and F_dat_o are the input and the output data bus. F_ful and F_emp indicate the full and empty state, which are active high. Given a FIFO, its parameters are shown in Table 2. To connect modules with FIFOs, we need to determine $D_{(n-1)n}$ and $W_{(n-1)n}$.

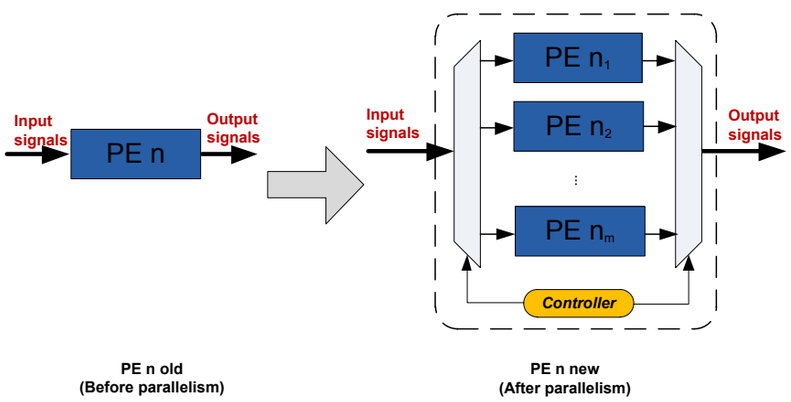


Fig. 5. Realization of block level parallelism

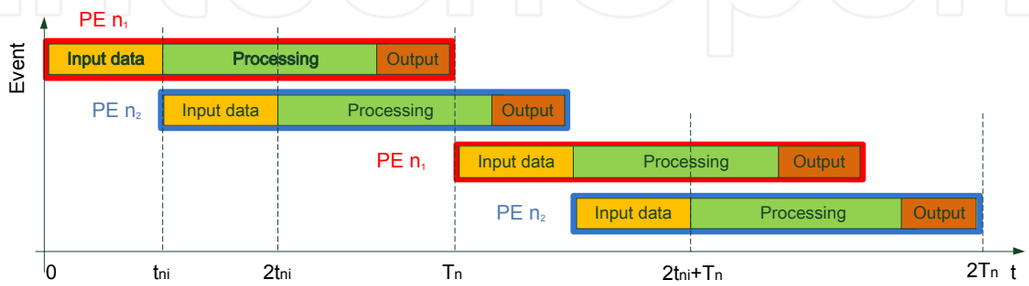


Fig. 6. Working mechanism of block level parallelism($P_n \leq \lfloor T_n / t_{ni} \rfloor$)

Name	Description	Examples[2]
$Fclk_{(n-1)n}$	Clock frequency (MHz)	50
$W_{(n-1)n}$	Data bus width	16
$A_{FIFO(n-1)n}$	Area: memory resource used (bit)	704
$D_{(n-1)n}$	FIFO depth	44
$f_{(n-1)n}(m)$ [1]	Number of data in FIFO at m^{th} cycle	
$SoF_{(n-1)n}(m)$	State of FIFO at m^{th} cycle; 1:Full; -1:Empty; 0:Other state	

[1] m means m^{th} cycle.
[2] This example comes from the FIFO between PE_1 and PE_2 in the JPEG encode case.

Table 2. The parameter of FIFO between PE_{n-1} and PE_n

4. Algorithm for block level parallelism

This section formulates the block level parallelism problem. After that, we propose an algorithm to solve the problem for multiple PEs in the system level.

4.1 Block level parallelism formulation

Given a design with N PEs, the throughput constraint TH_{ref} and the area constraint A_{ref}[3], we decide the n^{th} PE's parallelism degree P_n. That is

$$MIN.P_n, \quad \forall n \in [1, N] \tag{1}$$

$$s.t.TH_{all} \geq TH_{ref} \quad and \quad \sum_{n=1}^{N} \widehat{A}_n \leq A_{ref} \tag{2}$$

where TH_{all} denotes the entire throughput and $\widehat{A}_n$ is the PE_n's area after the block level parallelism. Without losing generality, we assume that the capacity of all FIFOs is infinite and A_{ref}=∞. We leave the FIFO sizing in the next section.

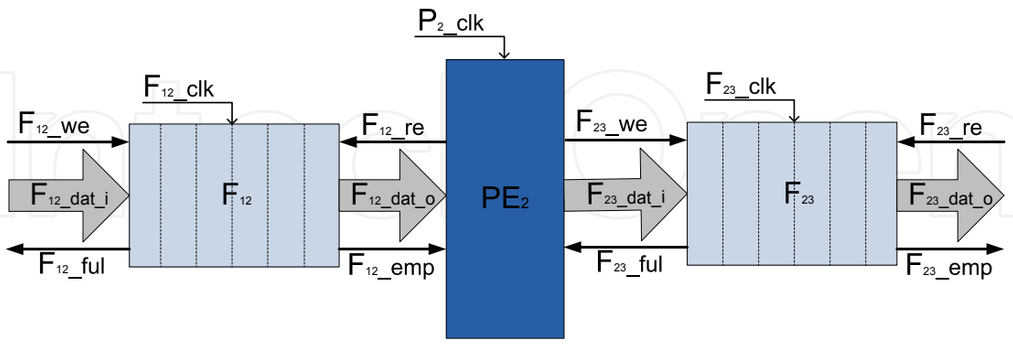


Fig. 7. Circuit diagram of FIFO blocks connecting to PE_2

[3] This area constraint doesn't consider the FIFO area.

4.2 Parameter extraction after block level parallelism

Before determining the parallelism degree of each PE, we first discuss how to extract new interface parameters for each PE after parallelism. That is to update the following parameters: $\widehat{TH}_{ni/o}$, $\widehat{A}_n$, $\widehat{T}_n$, $\widehat{f}_n$, and $\widehat{SoP}_n$, which are calculated based on P_n, $TH_{ni/o}$, A_n, T_n, f_n, and SoP_n.

First of all, we calculate $TH_{ni/o}$. As Figure 8 shows, larger parallelism degree won't always increase the throughput. It is limited by the input time t_{ni}. Assuming $t_{ni}>t_{no}$ and $P_n \leq \lfloor T_n/t_{ni} \rfloor$, we have

$$\widehat{TH}_{ni/o} = P_n * TH_{ni/o} \quad when \quad P_n \leq \lfloor T_n/t_{ni} \rfloor \tag{3}$$

For example, as shown in Figure 6, $\widehat{TH}_{ni/o}$=2*$TH_{ni/o}$ because P_n=2< $\lfloor T_n/t_{ni} \rfloor$=3. When $P_n \geq$

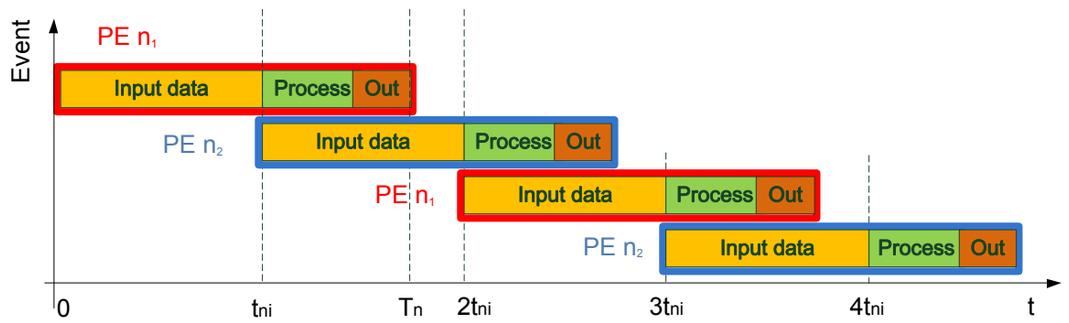


Fig. 8. Working mechanism of block level parallelism($P_n \geq \lceil T_n/t_{ni} \rceil$)

$\lceil T_n/t_{ni} \rceil$, we have

$$\widehat{TH}_{ni/o} = T_n/t_{ni} * TH_{ni/o} \quad when \quad P_n \geq \lceil T_n/t_{ni} \rceil \tag{4}$$

where the throughput is limited by the input time t_{ni}. More parallelism degree is useless in this case. For example, as shown in Figure 8, $\widehat{TH}_{ni/o}$=T_n/t_{ni}*$TH_{ni/o}$, because P_n=2=$\lceil T_n/t_{ni} \rceil$. When t_{ni}<t_{no} we have the similar conclusions. In summary, we have

$$\widehat{TH}_{ni/o} = \begin{cases} P_n * TH_{ni/o} & P_n < p_n \\ T_n/max\{t_{ni}, t_{no}\} * TH_{ni/o} & others \end{cases} \tag{5}$$

where

$$p_n = \lceil T_n/max\{t_{ni}, t_{no}\} \rceil \tag{6}$$

Second, we can solve $\widehat{A}_n$, $\widehat{T}_n$, and $\widehat{f}_n$. Ignoring the area of the controller, we have

$$\widehat{A}_n = P_n * A_n \tag{7}$$

Based on Figure 6 and 8, we conclude

$$\widehat{T}_n = \begin{cases} T_n + (P_n - 1) * max\{t_{ni}, t_{no}\} & P_n \leq p_n \\ P_n * max\{t_{ni}, t_{no}\} & others \end{cases} \tag{8}$$

Equation 5 shows that $\widehat{TH}_{ni}$ and $\widehat{TH}_{no}$ change at the same rate. Therefore,

$$\hat{f}_n = \widehat{TH}_{no} / \widehat{TH}_{ni} = TH_{ni} / TH_{ni} = f_n \tag{9}$$

Furthermore, we calculate $\widehat{SoP}_n$. $\widehat{SoP}_n$ is the combination of each sub-block's SoP. Therefore

$$\widehat{SoP}_n = \begin{cases} \sum_{i=0}^{P_n} SoP_n(m - i * t_{ni}) & t_{ni} \geq t_{no} \\ \sum_{i=0}^{P_n} SoP_n(m - i * (T_n - t_{no})) & t_{ni} < t_{no} \end{cases} \tag{10}$$

Finally, we can obtain all new parameters of a PE after parallelism. We will use those parameters to decide the parallelism degree in Section 4.3 and Section 5.

4.3 Block level parallelism degree optimization

To solve the optimization question in Section 4.1, we need to understand the relationship between TH_{all} and $\widehat{TH}_{ni/o}$. When PE_n is connected to the chain from PE_1 to $\text{PE}_{(n-1)}$, we define the output interface's throughput of PE_n as TH'_{no}. This parameter is different from $\widehat{TH}_{ni/o}$ because it has considered the rate mismatch effects from previous PEs. We have

$$TH'_{no} = \begin{cases} \widehat{TH}_{no} & TH'_{(n-1)o} > \widehat{TH}_{ni} \\ \hat{f}_n * TH'_{(n-1)o} & others \end{cases} \tag{11}$$

In fact, TH_{all}=TH'_{No}. Therefore, we can express TH_{all} in the following format

$$TH_{all} = \widehat{TH}_{bo} \prod_{i=b+1}^{N} f_i \tag{12}$$

where b is the index of the slowest PE_b. It is the bottleneck of the system.

To do the optimization of parallelism degrees, we purpose an algorithm shown in Algorithm 1. In the algorithm, the inputs are the number of PE N, the parameters of each PE $ParaG[N]$, each PE's maxim parallelism degree by Equation 6, and the design constraint TH_ref=TH_{ref}. $ParaG[N]$ includes $TH_{ni/o}$,$t_{ni/o}$,T_n,SoP_n shown in Table 1[4].

The output is each PE's optimal parallelism degree $P[N]$. Lines $1-7$ are to check if the optimization object is possible. Lines $8-14$ are the initializing process. Lines $15-20$ are the main loop. $pTH[N]$ equals to $\widehat{TH}_{ni/o}$ and TH_best denotes the best performance. Function $get\_pTH()$ returns the PE's $\widehat{TH}_{ni/o}$. Function $get\_THall()$ returns TH_now which means the TH_{all} under $\widehat{TH}_{ni/o}$ condition. Line 2 sets all the parallelism degree to its maximum value. After that, we get the fastest TH_{all} in Line 4. If the system can never approach the optimizing target, we will change the target in Line 6. In the main loop, we find the bottleneck in each step in Line 16 and add more parallelism degree to it. We will update $\widehat{TH}_{ni/o}$ in Line 18 and evaluate the system again in Line 19. We end this loop until the design constraints are satisfied.

[4] These parameters are initial ones got by Step 2

Algorithm 1 Block Level Parallelism Degree Optimization Algorithm

Input: $N, ParaG[N], p[N], TH\_ref$
Output: $P[N]$
1: **for** $k = 1 \rightarrow N$ **do**
2: $pTH[k] = get\_pTH(p[k], ParaG[k], p[k]), k = k + 1$
3: **end for**
4: $TH\_best = get\_THall(pTH, ParaG)$
5: **if** $TH\_best > TH\_ref$ **then**
6: $TH\_ref = TH\_best$
7: **end if**
8: **for** $k = 1 \rightarrow N$ **do**
9: $P[k] = 1, k = k + 1$
10: **end for**
11: **for** $k = 1 \rightarrow N$ **do**
12: $pTH[k] = get\_pTH(P[k], ParaG[k], p[k]), k = k + 1$
13: **end for**
14: $TH\_now = get\_THall(pTH, ParaG)$
15: **while** $TH\_now \geq TH\_ref$ **do**
16: $Bottleneck = get\_bottle(pTH, ParaG)$
17: $P[Bottleneck] + +$
18: $k = Bottleneck$
19: $pTH[k] = get\_pTH(P[k], ParaG[k], p[k]), k = k + 1$
20: $TH\_now = get\_THall(pTH, ParaG)$
21: **end while**

5. Algorithm for FIFO-connected blocks

This section formulates the FIFO interconnecting problem. We then demonstrate that this problem can be solved by a binary searching algorithm. Finally, we propose an algorithm to solve the FIFO interconnecting problem of multiple PEs in the system level.

5.1 FIFO interconnection formulation

Given a design consisting of N PEs, we need to determine the depth $D_{(i-1)i}$ of each FIFO[5], which maximizes the entire throughput TH_{all} and minimizes the FIFO area of $A_{FIFO_{all}}$.

$$MIN. \quad \sum_{i=2}^{N} D_{(i-1)i} \tag{13}$$

$$s.t. \quad TH_{all} \geq TH_{ref} \quad and \quad A_{FIFO_{all}} \leq A_{FIFO_{ref}} \tag{14}$$

where TH_{ref} and $A_{FIFO_{ref}}$ can be the user-specified constraints or optimal values of the design. Without losing generality, we set TH_{ref}=$(TH_{all})_{max}$ and A_{FIFO_ref}=∞. We assume that F_{01} never empties and $F_{N(N+1)}$ never fulls. That is, $\forall m$, $SoF_{01}(m) \neq -1$ and $SoF_{N(N+1)}(m) \neq 1$[6].

[5] We assume that the $W_{(i-1)i}$ is decided by the application.
[6] This means that we only consider the operating state of the design instead of the halted state.

5.2 FIFO capacity optimization

We can conclude a brief relationship between $\mathrm{TH}_{ni/o}$ and D_i. For PE_n, we define the real throughput as $\widetilde{TH}_{ni/o}$, when connected with F_{n-1} of D_{n-1} and F_{n+1} of D_{n+1}. Then we set

$$\widetilde{TH}_{ni/o} = f(D_{n-1}, D_{n+1}) \tag{15}$$

We know that a small D_{n-1} or D_{n+1} will cause $\widetilde{TH}_{ni/o}$<$\mathrm{TH}_{ni/o}$. Also, when $\widetilde{TH}_{ni/o}$=$\mathrm{TH}_{ni/o}$, larger D_{n-1} or D_{n+1} will not increase performance any more. Therefore, as it is shown in Figure 2, $f(x)$ is a monotone nondecreasing function with a boundary.

With the fixed relationship between $\mathrm{TH}_{ni/o}$ and D_i, we can solve the FIFO capacity optimization problem by a binary searching algorithm based on the system level simulations. We describe this method to determine the FIFO capacity for multiple PEs ($N > 2$) in Algorithm 2.

Algorithm 2 FIFO Capacity Algorithm for $N \geq 2$

Input: $N, ParaG[N], Inital\_D[N]$
Output: $D[N]$
1: $k = 1, n = 1$
2: **while** $k < N$ **do**
3: $\quad D[k] = Initial\_D[k]$
4: **end while**
5: $TH\_obj = get\_TH(D, ParaG)$
6: $TH\_new = TH\_obj, Upper = D[1], Mid = D[1], Lower = 1$
7: **while** $n < N$ **do**
8: $\quad$ **if** $TH\_new = TH\_obj$ **then**
9: $\quad\quad D[n] = ceil((Mid - Lower)/2)$
10: $\quad\quad Upper = Mid, Mid = D[n]$
11: $\quad$ **else**
12: $\quad\quad D[n] = ceil((Upper - Mid)/2)$
13: $\quad\quad Lower = Mid, Mid = D[n]$
14: $\quad$ **end if**
15: $\quad TH\_new = get\_TH(D, ParaG)$
16: $\quad$ **if** $Upper = Lower$ **then**
17: $\quad\quad n = n + 1$
18: $\quad\quad Upper = D[n], Mid = D[n], Lower = 1$
19: $\quad$ **end if**
20: **end while**

The inputs are the number of PE N, the parameters of each PE $ParaG[N]$ and each FIFO's initial capacity $Initial\_D[N]$. $ParaG[N]$ includes $TH_{ni/o}, t_{ni/o}, T_n, SoP_n$ shown in Table 1[7]. $Initial\_D[n]$ means the initial searching value of $\mathrm{D}_{n(n+1)}$, which is big enough to ensure $\widetilde{TH}_{ni/o}$=TH_{all}. The output is each FIFO's optimal depth $D[N]$. Lines $1-6$ are the initializing process. Lines $7-20$ are the main loop. Function $get\_TH()$ in line 5 and 15 can return the entire throughput under different $D[N]$ settings. Variable TH_obj is the searching object calculated by $Initial\_D[N]$. $Initial\_D[N]$ equals to TH_{all} and TH_new is the current throughput calculated based on $D[N]$. $Upper$, Mid, and $Lower$ decide the binary searching range. In each loop, n means that the capacity of $\mathrm{F}_{n(n+1)}$ is processed. We get the searching

[7] These parameters are updated by Block Level Parallelism step

point and the range according to TH_new in lines $8-14$. We update TH_new in line 15. The end condition is checked in line 16. When $n = N$, it means that all FIFOs have their optimal capacity. As we can see, the most time-consuming part of the algorithm is the $getTH()$ function. It calls for an entire simulation of the hardware. Therefore, we build a system level simulator instead of a RTL level one. It can shorten the optimization greatly. The system level simulator adopts the parameters extracted in Step 2. The C-based system level simulator will be released on our website soon.

6. Experiments

In this section, we first explain our experimental configurations. Then, we compare the flatten approach, the hierarchical method without block level parallelism (BLP) and with BLP under several real benchmarks. After that, we break down the advantages by two aspects: the block level parallelism and the FIFO sizing. We then show the effectiveness of the proposed algorithm to optimize the parallel degree. Finally, we demonstrate the advantages from the FIFO sizing method.

6.1 Experimental configurations

In our experiments, we use a C2RTL tool called *eXCite* (Y Exploration Inc., 2011). The HDL files are simulated by Mentor Graphics' ModelSim to get the timing information. The area and clock information is obtained by Quartus II from Altera. *Cyclone* II FPGAs are selected as the target hardware. We derive seven large streaming applications from the high-level synthesis benchmark suits CHstone(Hara et al. (2008)). They come from real applications and consist of programs from the areas of image processing, security, telecommunication and digital signal processing.

- *JPEG encode/decode*: JPEG transforms image between JPEG and BMP format.
- *AES encryption/decryption*: AES (Advanced Encryption Standard) is a symmetric key crypto system.
- *GSM*: LPC (Linear Predictive Coding) analysis of GSM (Global System for Mobile Communications).
- *ADPCM*: Adaptive Differential Pulse Code Modulation is an algorithm for voice compression.
- *Filter Group*: The group includes two FIR filters, a FFT and an IFFT block.

6.2 System optimization for real cases

We show the synthesized results for seven benchmarks and compare the flatten approach, the hierarchical approach without and with BLP. Table 3 shows the clock cycles saved by the hierarchical method without and with BLP. The last column in Table 3 shows the BLP vector for each PE. The i^{th} element in the vector denotes the parallel degree of the PE_i. The total speedup represents the clock cycle reductions from the hierarchical approach with BLP. As we can see, the hierarchical method without BLP achieves up to 10.43 times speedup compared with the flatten approach. However, the BLP can provide considerable extra up to another 5 times speedup compared with the hierarchial method without BLP. It should be noted that

	Benchmark	Flatten approach	Hierarchical W.O. BLP(speedup)	Hierarchical W. BLP(speedup)	BLP degree (P_1..P_n)
Min T_{all} (cycles)	JPEG encode	42,475,202	4,070,603 (x10.43)	1,850,907 (x22.94)	(1,3,1)
	JPEG decode	623,090	456,821 (x1.364)	115,622 (x5.389)	(1,1,4,1)
	AES encryption	1,904,802	719,263 (x2.648)	216,393 (x8.803)	(4,2,3,2)
	AES decryption	2,185,802	867,306 (x2.388)	229,570 (x9.521)	(4,2,4,2)
	GSM	620,802	204,356 (x3.038)	55,306 (x11.22)	(4,4,4,1,1,1)
	ADPCM	35,691	12,464 (x2.864)	3,762 (x9.487)	(4,2,2,2,3)
	Filter groups	6,537,416	1,702,406 (x3.84)	511,853 (x12.77)	(2,1,1,4,1,2)

BLP: Block level parallelism.

Table 3. System optimization result of minimal clock cycles

	Benchmark	Flatten approach	Hierarchical W.O. BLP	Hierarchical W. BLP	Total Speedup
Max Clk_{all} (MHz)	JPEG encode	69.74	74.2	74.2	x1.064
	JPEG decode	71.15	71.3	71.3	x1.002
	AES encode	71.24	91.06	91.06	x1.278
	AES decode	75.56	87.35	87.35	x1.156
	GSM	55.73	59.16	59.16	x1.062
	ADPCM	53.29	68.32	68.32	x1.282
	Filter groupe	93.41	96.69	96.69	x1.035

BLP: Block level parallelism.

Table 4. System optimization result of maximal clock frequency

the BLP will lead to area overheads in some extents. We will discuss those challenges in the following experiments. Furthermore, Table 4 shows the maximum clock frequency of three approaches. As we can see, the BLP does not introduce extra delay compared with the pure hierarchical method.

6.3 Block level parallelism

The previous experimental results show the total advantages from the hierarchial method with BLP. This section will discuss the performance and the area overheads of BLP alone. We show the throughput improvement and the area costs in the GSM benchmark in Figure 9[8]. We list the BLP vector as the horizontal axis. As we can see, parallelizing some PEs will increase the throughput. For the BLP vector $(1,2,1,1,1,1)$, we duplicate the second PE_2 by two. It will improve the performance by 4% with 48% area overheads. The result comes from the rate mismatch between PEs. It indicates that duplicating single PE may not increase the throughput effectively and the area overheads may be quite large. Therefore, we should develop an algorithm to find the optimal BLP vector to boost the performance without introducing too many overheads. For example, the BLP vector $(4,4,4,1,1,1)$ leads to over 4 times performance speedup while with only less than 3 times area overheads.

Furthermore, we evaluate the proposed BLP algorithm with the approach duplicating the entire hardware. Figure 10 demonstrates that our algorithm can increase the throughput with less area. It is because the BLP algorithm does not parallelize every PE and can explore more fine-grained design space. Obviously, the BLP method provides a solution to trade off

[8] We observe similar trends in other cases.

performance with area more flexibly and efficiently. In fact, as the modern FPGA can provide more and more logic elements, it makes the area not so urgent as the performance, which is the first-priority metric in most cases.

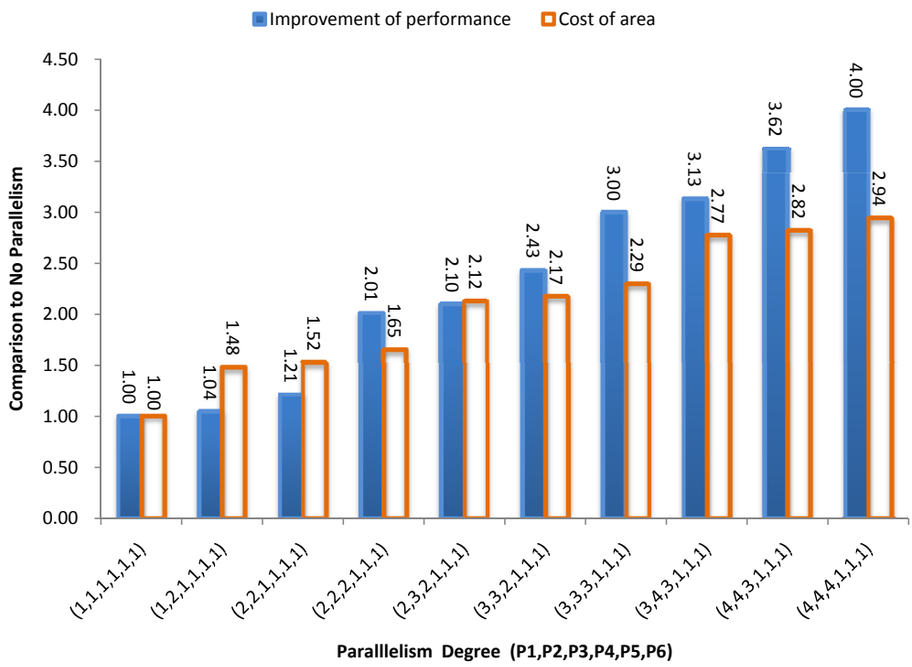


Fig. 9. Speedup and Area cost in GSM case

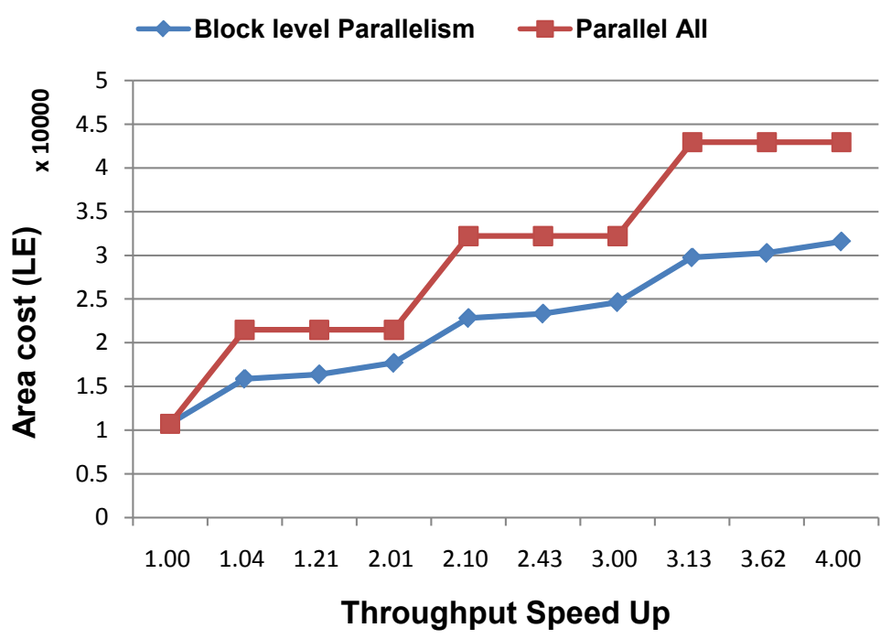


Fig. 10. Advantage of Block Level Parallelism algorithm

Benchmark		D_{12}	D_{23}	D_{34}	D_{45}	D_{56}	T_{all}
JPEG encode	System Level	43	2	-	-	-	4080201
	RTL Level	44	2	-	-	-	4070603
JPEG decode	System Level	2	33	17	2	-	456964
	RTL Level	2	33	18	2	-	456821
AES encryption	System Level	2	2	2	-	-	719364
	RTL Level	3	2	3	-	-	719263
AES decryption	System Level	2	257	2	-	-	867407
	RTL Level	3	249	3	-	-	867306
GSM	System Level	54	2	2	2	2	204554
	RTL Level	55	2	2	2	2	204356
ADPCM	System Level	2	2	2	2	2	12464
	RTL Level	2	2	2	2	1	12464
Filter group	System Level	2	2	86	2	2	1701896
	RTL Level	2	2	87	2	2	1701846

Table 5. Optimal FIFO capacity algorithm experiment result in 7 real cases

6.4 Optimal FIFO capacity

We show the simulated results for real designs with multiple PEs. First of all, we show the relationship between the FIFO size and the running time T_{all}. Figure 11 shows the JPEG encoding case. As we can see, the FIFO size has a great impact on the performance of the design. In this case, the optimal FIFO capacity should be D_{12}=44, D_{23}=2.

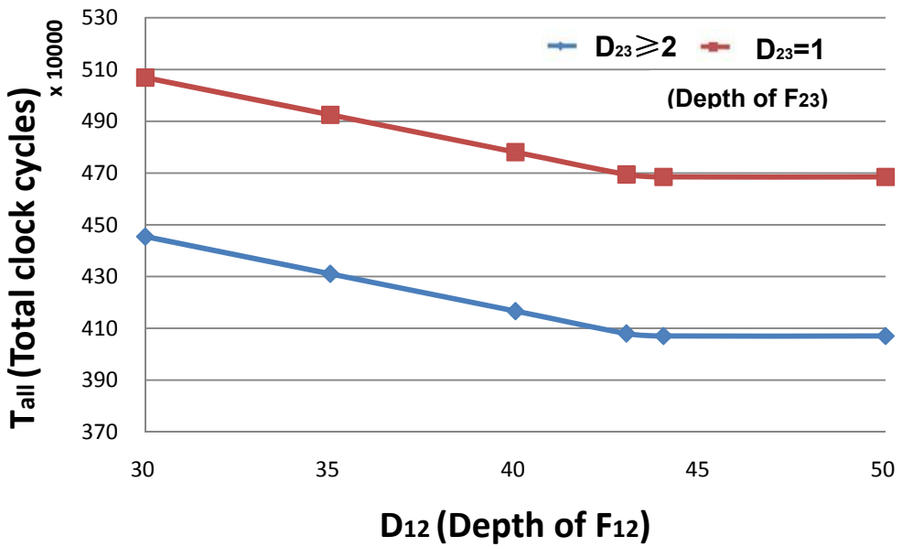


Fig. 11. FIFO capacity in JPEG encode case

Table 5 lists both the system level simulation results and the RTL level experimental ones on FIFO size in seven cases. It shows that our approach is accurate enough for those real cases. Though little mismatch exists, the difference is very small. Compared to the magnitudes of speedup to determine the FIFO size, our approach is quite promising to be used in architecture level design space exploration.

Benchmark	Memory resource used(bit)		Savings
	FIFOs with [1] enough size	FIFOs with optimized size	
JPEG encode	10,048	2,624	x3.83
JPEG decode	38,776	8,376	x4.63
AES encode	92,160	67,968	x1.36
AES decode[2]	92,160	75,808	x1.22
GSM	36,028	8,602	x4.19
ADPCM	54,040	3,736	x14.46
Filter groupe	114,400	76,736	x1.49

[1] We set each FIFO depth as 128.
[2] In this case we set each FIFO depth as 256.

Table 6. Area saved

The memory resource savings by well designing FIFO are listed in Table 6. Compared to the large enough design strategy, the memory savings are significant. Moreover, compared to the method using RTL level simulator to decide FIFO capacity, our work is extremely time efficient. Considering a hardware with N FIFO to design, each FIFO size is fixed using a binary searching algorithm. It will request $log_2(p)$ times simulations with the initial FIFO depth value $D_{(n-1)n} = p$. Assuming that the average time cost by *ModelSim* RTL level simulation is C, the entire exploration time is $N * log_2(p) * C$. Considering the $FilerGroup$ case with $N = 5$, $p = 128$ and $C = 170$ seconds, which are typical values on a normal PC, we have to wait 100 minutes to find the optimal FIFO size. However, our system level solution can finish the exploration in seconds.

7. Related works

Many C2RTL tools (Gokhale et al., 2000; Lhairech-Lebreton et al., 2010; Mencer, 2006; Villarreal et al., 2010) are focusing on streaming applications. They create design architectures including different modules connected by first-in first-out (FIFO) channels. There are some other tools focusing on general purpose applications. For example, Catapult C (Mentor Graphics, 2011) takes different timing and area constraints to generate Pareto-optimal solutions from common C algorithms. However, little control on the architecture leads to suboptimal results. As (Agarwal, 2009) has shown, FIFO-connected architecture can generate much faster and smaller results in streaming applications.

Among C2RTL tools for streaming applications, GAUT (Lhairech-Lebreton et al., 2010) transforms C functions into pipelined modules consisting of processing units, memory units and communication units. Global asynchronous local synchronous interconnections are adopted to connect different modules with multiple clocks. ROCCC (Villarreal et al., 2010) can create efficient pipelined circuits from C to be re-used in other modules or system codes. Impulse C (Gokhale et al., 2000) provides a C language extension to define parallel processes and communication channels among modules. ASC (Mencer, 2006) provides a design environment for users to optimize systems from algorithm level to gate level, all within the same C++ program. However, previous works keep how to determine the FIFO capacity efficiently unsolved. Most recently, (Li et al., 2012) presented a hierarchical C2RTL framework with analytical formulas to determine the FIFO capacity. However, block level parallelism

is not supported and their FIFO sizing method is limited to PEs with certain input/output interfaces.

During the hierarchical C2RTL flow, a key step is to partition a large C program into several functions. Plenty of works have been done in this field. Many C-based high level synthesis tools, such as SPARK (Gupta et al., 2004), eXcite (Y Exploration Inc., 2011), Cyber (NEC Inc., 2011) and CCAP (Nishimura et al., 2006), can partition the input code into several functions. Each function has a corresponding hardware module. However, it leads to a nontrivial datapath area overhead because it eliminates the resource sharing among modules. On the contrary, function inline technique can reduce the datapath area via resource sharing. The fast increasing complexity of the controller makes the method inefficient. Appropriate function clustering (Okada et al., 2002) in a sub module provides a more elegant way to solve the partition problem. But it is hard to find a proper clustering rule. For example, too many functions in one cluster will also lead to a prohibitive complexity in controllers. In practise, architects often help the partition program to divide the C algorithms manually.

Similar to the hierarchical C2RTL, multiple FIFO-connected processing elements (PE) are used to process audio and video streams in the mobile embedded devices. Researchers had investigated on the input streaming rates to make sure that the FIFO between PEs will not overflow, while the real-time processing requirements are met. On-chip traffic analysis of the SoC architecture (Lahiri et al., 2001) had been explored. However, their simulation-based approaches suffer from a long executing time and fail in exploring large design space. A mathematical framework of rate analysis for streaming applications have been proposed in reference (Cruz, 1995). Based on the network calculus, reference (Maxiaguine et al., 2004) extended the service curves to show how to shape an input stream to meet buffer constraints. Furthermore, reference (Liu et al., 2006) discussed the generalized rate analysis for multimedia processing platforms. However, all of them adopts a more complicated behavior model for PE streams, which is not necessary in the hierarchical C2RTL framework.

8. Conclusion

Improving the booming design methodology of C2RTL to make it more widely used is the goal of many researchers. Our work of the framework does have achieved the improvement. We first propose a hierarchical C2RTL design flow to increase the performance of a traditional flatten one. Moreover, we propose a method to increase throughput by making block level parallelism and an algorithm to decide the degree. Finally, we develop an heuristic algorithm to find the optimal FIFO capacity in a multiple-module design. Experimental results show that hierarchical approach can improve performance by up to 10.43 times speedup, and block level parallelism can make extra 4 times speedup with 194% area overhead. What's more, it determines the optimal FIFO capacity accurately and fast. The future work includes automatical C code partition in the hierarchical C2RTL framework and adopting our optimizing algorithm in more complex architectures with feedback and branches.

9. Acknowledgement

The authors would like to thank reviewers for their helpful suggestions to improve the chapter. This work was supported in part by the NSFC under grant 60976032 and 61021001,

National Science and Technology Major Project under contract 2010ZX03006-003-01, and High-Tech Research and Development (863) Program under contract 2009AA01Z130.

10. References

Agarwal, A. (2009). *Comparison of high level design methodologies for algorithmic IPs: Bluespec and C-based synthesis*, PhD thesis, Massachusetts Institute of Technology.

Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K. & Zhang, Z. (2011). High-level synthesis for fpgas: From prototyping to deployment, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 30(4): 473–491.

Cruz, R. (1995). Quality of service guarantees in virtual circuit switched networks, *Selected Areas in Communications, IEEE Journal on* 13(6): 1048–1056.

Gokhale, M., Stone, J., Arnold, J. & Kalinowski, M. (2000). Stream-oriented fpga computing in the streams-c high level language, *fccm*, IEEE, p. 49.

Guo, Y. & McCain, D. (2006). Rapid prototyping and vlsi exploration for 3g/4g mimo wireless systems using integrated catapult-c methodology, *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, Vol. 2, IEEE, pp. 958–963.

Gupta, S., Gupta, R. & Dutt, N. (2004). *SPARK: a parallelizing approach to the high-level synthesis of digital circuits*, Vol. 1, Kluwer Academic Pub.

Hara, Y., Tomiyama, H., Honda, S., Takada, H. & Ishii, K. (2008). Chstone: A benchmark program suite for practical c-based high-level synthesis, *IEEE International Symposium on Circuits and Systems*, IEEE, pp. 1192–1195.

Intel Inc. (2011). Stellarton atom processor, *Website: http://www. intel. com* .

Lahiri, K., Raghunathan, A. & Dey, S. (2001). System-level performance analysis for designing on-chip communication architectures, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 20(6): 768–783.

Lhairech-Lebreton, G., Coussy, P. & Martin, E. (2010). Hierarchical and multiple-clock domain high-level synthesis for low-power design on fpga, *2010 International Conference on Field Programmable Logic and Applications*, IEEE, pp. 464–468.

Li, S., Liu, Y., Zhang, D., He, X., Zhang, P. & Yang, H. (2012). A hierarchical c2rtl framework for fifo-connected stream applications, *Proceedings of the 2012 Asia and South Pacific Design Automation Conference*, IEEE Press, pp. 1–4.

Liu, Y., Chakraborty, S. & Marculescu, R. (2006). Generalized rate analysis for media-processing platforms, *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, Vol. 6, Citeseer, pp. 305–314.

Maxiaguine, A., Künzli, S., Chakraborty, S. & Thiele, L. (2004). Rate analysis for streaming applications with on-chip buffer constraints, *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, IEEE Press, pp. 131–136.

Mencer, O. (2006). Asc: a stream compiler for computing with fpgas, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 25(9): 1603–1617.

Mentor Graphics, M. (2011). Catapult c synthesis, *Website: http://www. mentor. com* .

NEC Inc. (2011). CyberWorkBench, *Website: http://www.nec.com/global/prod/cwb/* .

Nishimura, M., Nishiguchi, K., Ishiura, N., Kanbara, H., Tomiyama, H., Takatsukasa, Y. & Kotani, M. (2006). High-level synthesis of variable accesses and function

calls in software compatible hardware synthesizer ccap, *Proc. Synthesis And System Integration of Mixed Information technologies (SASIMI)* pp. 29–34.

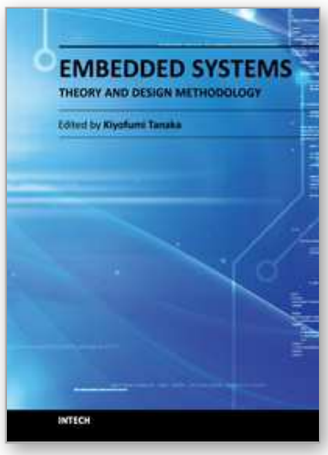
Okada, K., Yamada, A. & Kambe, T. (2002). Hardware algorithm optimization using bach c, *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 85(4): 835–841.

Rossler, M., Wang, H., Heinkel, U., Engin, N. & Drescher, W. (2009). Rapid prototyping of a dvb-sh turbo decoder using high-level-synthesis, *Forum on Specification & Design Languages, 2009.*, IEEE, pp. 1–6.

Schafer, B., Trambadia, A. & Wakabayashi, K. (2010). Design of complex image processing systems in esl, *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, IEEE Press, pp. 809–814.

Villarreal, J., Park, A., Najjar, W. & Halstead, R. (2010). Designing modular hardware accelerators in c with roccc 2.0, *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, IEEE, pp. 127–134.

Y Exploration Inc. (2011). eXCite, *Website: http://www.yxi.com* .



Embedded Systems - Theory and Design Methodology
Edited by Dr. Kiyofumi Tanaka

ISBN 978-953-51-0167-3
Hard cover, 430 pages
Publisher InTech
Published online 02, March, 2012
Published in print edition March, 2012

Nowadays, embedded systems - the computer systems that are embedded in various kinds of devices and play an important role of specific control functions, have permitted various aspects of industry. Therefore, we can hardly discuss our life and society from now onwards without referring to embedded systems. For wide-ranging embedded systems to continue their growth, a number of high-quality fundamental and applied researches are indispensable. This book contains 19 excellent chapters and addresses a wide spectrum of research topics on embedded systems, including basic researches, theoretical studies, and practical work. Embedded systems can be made only after fusing miscellaneous technologies together. Various technologies condensed in this book will be helpful to researchers and engineers around the world.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen