A Parameterized Architecture Model in High Level Synthesis for Image Processing Applications

Yazhuo Dong, Yong Dou

Department of Computer Science, National University of Defence Technology
Changsha, P. R. China, 410073
Tel : 86-731-4573647
Fax : 86-731-4573647
e-mail : dongyazhuo@nudt.edu.cn
yongdou@nudt.edu.cn

Abstract - Most image processing applications are computationally intensive and data intensive. Reconfigurable hardware boards provide a convenient and flexible solution to speed up these algorithms. To get a high performance design without going through the time-consuming hardware design process for each different algorithm, we present a universal parameterized architecture in high level synthesis to generate the hardware frames for all image processing applications automatically. The value of the parameters which decide the target architecture can be obtained from the compiler. The algorithm how to get these parameters is also discussed in this paper.

I. Introduction

The extreme flexibility of Field- Programmable- Gate-Arrays (FPGAs) coupled with the wide spread acceptance of hardware description languages have made FPGAs popular used. Unfortunately, developing programs that execute on FPGAs is extremely cumbersome, demanding that software developers also assume the role of hardware designers [1]. In order to deal with the problem, developers try to explore design automation and tool support. It is desirable to implement the system using behavioral level languages, as opposed to register transfer level languages.

High Level Synthesis (HLS) tools provide a bridge between the algorithm written in a high level language (Matlab, C, C++, etc) and a lower level Hardware Description Language (HDL). Some also partition into hardware and software components. We can classify different design methods into two approaches: the annotation and constraint-driven approach and the source-directed compilation approach. The first approach preserves the source programs in C or C++ as much as possible and makes use of annotation and constraint files to drive the compilation process, such as SPARK[2], Sea Cucumber[3], SPC[4], Streams-C[5], Catapult C[6] and DEFACTO[7]. The second approach modifies the source language to let the designer to specify, for instance, the amount of parallelism or the size of variables, such as ASC[8], C2Verilog[9], Handel-C[10], Handy-C[11], Bach-C[12] and SpecC[13] etc. All of these design automation tools aim to raise the level of design.

Of particular interests to this research are image processing applications where there are three kinds of operations: point operations such as GST(Gray Scale Transformation), window operations such as edge detectors, and some complicated perpendicular transforms such as DFT(Discrete Fourier Transform). All these operations have similar calculation patterns: a loop or a loop nest operates with array variable. There are multiple references to an array element in the same or a subsequent iteration, so the memory structure can be designed abortively, and data reuse can be exploited.

To this day, a standard strategy to explore data reuse is to identify multiple memory accesses to the same memory location this reuse identical data, keep these data in a group of registers named smart buffer until the data will never be reference any more[14][15][16][17][18][19]. Current architecture generation efforts have several shortcomings. Firstly, they would demand a large number of registers if the reuse distance be potentially large. Secondly, while some tools now incorporate internal RAM modules and the mapping of array variables to them, they have mostly ignored system level issues when dealing with external memories. Thirdly, it will take a long time to initialize array variables into the internal RAM before starting processing, which is unnecessary.

In this paper we will put forward a universal parameterized architecture for signal processing applications. And we also describe the application of how to use data dependence analysis to develop a compilation and synthesis strategy which will generate different parameters for different image processing applications. This analysis aims at exploring a wide range of program transformations with the goal of reducing the number of required memory accesses to speed up the processing, and at the same time employing a small size of internal RAM blocks and smart buffer.

The main difference between our approach with others' is that we don't wait all array variables for initializing into the internal RAM to start the processing, but execute in a data-driven mode which means starting current operations as soon as possible. There are still pipelined off-chip memory accesses controls during the whole period of performing, thus it is possible to use a less number of internal RAM blocks. And we also don't keep all reused data in smart buffer, but hold them in internal RAM blocks to debase the size of smart buffer, so it is necessary to develop a more complicated compilation strategy to send the right data to smart buffer at the right time.

This paper makes the following contributions:

- It presents a universal parameterized architecture for image processing applications.
- It presents an analysis algorithm for the automatic derivation of these control structures. This algorithm

1-4244-0630-7/07/$20.00 ©2007 IEEE.

uses the data access patterns in the loop nest to choose a design with the lowest number of required memory accesses, and the least size of RAM blocks and smart buffer.

- It describes a data scheme strategy between on-chip and off-chip RAM blocks, and a data transmission approach between on-chip RAM blocks with smart buffers.
- It presents preliminary experimental simulation results for the automatic translation of a set of image processing applications onto FPGA using our synthesis strategy. These results indicate these control structures can serve as the basis of a successful compilation and synthesis tool.

The rest of the paper is structured as follows. We compare different mapping approaches via an example in section 2. Next we describe the universal parameterized architecture, the control structures and their rationale. Section 4 describes our compiler analysis algorithms for realizing the hardware design automatically. Section 5 presents preliminary simulation experimental results for a set of image processing applications. In section 6 we give a conclusion.

II. Example

```
char img[SIZE][SIZE], edge[SIZE][SIZE];
int uh1, uh2, threshold;
for (i=0; i < SIZE - 4; i++) {
    for (j=0; j < SIZE - 4; j++) {
        uh1=(((-img[i][j])+(-(2*img[i+1][j]))+(-img[i+2][j]))+
    ((img[i][j+2])+(2*img[i+1][j+2])+(img[i+2][j+2])));
        uh2=(((-img[i][j])+(img[i+2][j]))+(-(2*img[i][j+1]))+
    (2*img[i+2][j+1])+((- img[i][j+2])+(img[i+2][j+2])));
        if ((abs(uh1)+abs(uh2))<threshold)
            edge[i][j]=”0xff”;
        else
            edge[i][j]=”0x00;
    }
}
```

Fig. 1. Sobel Edge detection computation example

We now illustrate the use of different storage and control structures in the automatic mapping of an example computation onto an FPGA-based computing engine. The computation is written in C as depicted in Fig.1. It consists of a single loop nest and computes the Sobel edge detection algorithm over an 8 bit gray-scale image. The image is stored as the 2-dimensional img array of characters. The output is stored as the 2-dimensional edge array.

A native implementation of this computation is presented in Fig.2. The reconfigurable computing compiler performs a straightforward hardware generation, and the functional unit would need to access all nine input data values in the current window which would require a large amount of memory bandwidth and involve pipeline bubbles in the data path.

A prevalent mapping strategy to reduce the number of required memory accesses is shown in Fig.3. Smart buffer hold the data input queues to exploit the fact that consecutive iterations of the inner loop use data that previous iterations have fetched. This strategy betakes more registers but significantly reduces the number of memory accesses per iteration.

Fig.4 below illustrates the target architecture design of our strategy for the Sobel edge detection computation.

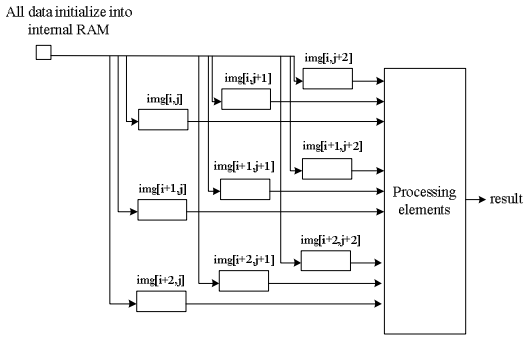

Fig. 2. Native memory structure generation

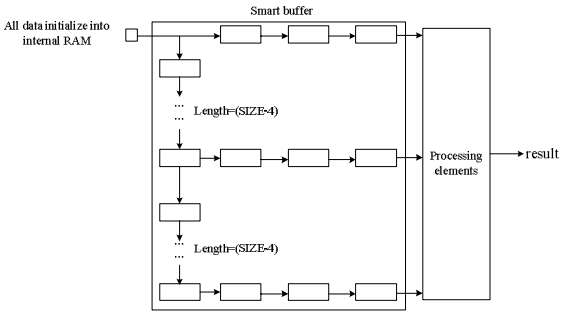

Fig. 3. Prevalent strategy to data reuse

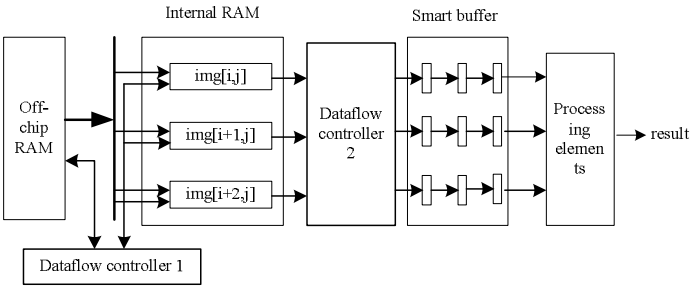

Fig. 4. Our strategy for the Sobel edge detection computation

We contrive three (SIZE-4) deep RAM blocks, one of which holds 1-dimensional img array data values. Once row i, i+1,and the first three data of row i+2 are ready, the whole processing can be started. In case the row i is done, the newer input data of row i+3 will take the place of row i. Dataflow controller 1 answers for the data transmission between off-chip memory and internal RAM. At the same time, transferring data from internal RAM to smart buffer which is dominate by dataflow controller 2 can be carried through simultaneity with sending data to processing elements, so the target design wouldn’t slower the calculation speed, though we use less number of internal memory elements and registers.

III. Parameterized Architecture

Fig.5 below presents the universal layout of the target architecture design that our compiler uses to generate for image processing operations. This architecture has several internal RAM blocks to keep the reused data and smart buffer to keep the computing operands, the rule of our approach to hold data values in internal Ram blocks is that data that will be reused in the following outer loop should be kept in the internal RAM until the data will never be used again, and the data to be used in the current loop will shift in the registers of smart buffer.

There are also some auxiliary control structures to control the execution. Dataflow controller 1 and 2 will keep track of which iterations of the loop are currently in execution and generates the appropriate control signal to realize the pipelined memory accesses. The address generation unit is a programmable array address generation unit the compiler synthesizes with auto-increment and auto-decrement capabilities. This unit is controlled by dataflow controller 2 to steer the appropriate data into the appropriate registers of smart buffer.

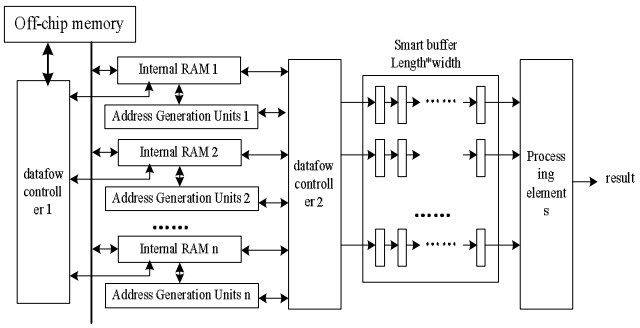

Fig. 5. Target memory architecture for window operations

We have designed a parameterized FSM (finite state machine) in the verilog library to control the whole processing. The FSM has five states for all image processing applications: primal state, data initialization into on-chip RAM, data initialization into smart buffer, start processing and finish.

case (sending states)
1: Primal state: set the Token-Ring to the next on-chip RAM block, if the current RAM is the number $MEM_{i.bank}$, set the Token-Ring to the first block again;
2: Sending data request signal to the off-chip RAM, ready to receive a new data values;
3: Receive a data and keep it in the current on-chip RAM block who has the Token-Ring. The Counter increase to register how many data values have been in the current RAM: counter_num++;
4: Judgment.
if (counter_num< the depth of RAM block $MEM_{i.deep}$) goto2;
else the current RAM block is full, goto 1;
endcase

Fig. 6. The FSM of sending data from off-chip RAM to on-chip RAM

Fig.6 illustrates the parameterized FSM of dataflow controller 1 which is used to deal with the data transmission from off-chip RAM to on-chip RAM. The FSM make it possible to parallel data transferring and calculation to speed up the processing.

Fig.7 gives the parameterized FSM of dataflow controller 2 who is master of sending the right data from on-chip RAM blocks to smart buffer at the right time. The FSM also ensures that the whole calculation processing can be carried through accurately.

case (control states)
1:Idle state, waiting for $data_{in}$ data values being initialized into the RAM blocks;
2: Sending $buffer_{length} * buffer_{width}$ data values from smart buffer to the processing elements;
3: Waiting for the result;
4: Sending result,
Increase the result counter, result_counter++;
If (result_count== image width)
result_counter=0;
Increase the row counter which registers how many rows have been done; row_done++;
5: Increase counter;
RAM_current_do++;
// RAM-current-do records which RAM blocks data is currently be deal with and at the same time shift the registers in smart buffer.
If (RAM_current_do= =image width)
RAM_current_done=0;
6. Receive the next data
If(row_done==image length) finish,
goto 1;
else goto 2;
endcase

Fig. 7. The FSM of sending data from on-chip RAM to smart buffer

From the FSM, we can draw a conclusion that in order to generate the target architecture automatically, there are five parameters required: the number of RAM blocks needed $MEM_{i.bank}$, the depth of them $MEM_{i.deep}$, the length and width of smart buffer $buffer_{length}$ and $buffer_{width}$, and the number of data values being initialized into the RAM blocks before starting the processing $data_{in}$. This is the duty of the compiler to achieve the five parameters. We will discuss the algorithm of compiler support detailed in the following section.

IV. Compiler support

We provide a flexible strategy for the compiler to gain the five parameters automatically. Part of the definitions in article [14] will be adopted.

A dependence vector $<d_1, d_2, \cdots, d_n>$ refers to a vector difference of the distance in an n-dimensional loop iteration space where $d_k \geq 0$. A constant dependence vector entity c means that the distance between two dependent array references in the corresponding loop is c. For example, the array references A[i][j] with A[i+2][j] and A[i][j+2] that induce the dependence vectors <2,0> and <0,2> respectively. We usually choose the longest distance, as in the code shown in Fig.8, the dependence vector of data array A is <0, 4>.

loopi_i0, loopi_ip and loopi_inn are used to describe a loop, which is from loopi_i0 to loopi_inn in step of loopi_ip. I_i refers to the number of steps of loop i, where $I_i = |loopi\_inn - loopi\_i0| / |loopi\_ip| + 1$.

In image processing operations, general dependences are either loop-independent or loop-carried. The former occurs between statements in the same loop iteration, and the latter between statements in different iterations. As the example shown in Fig.8, the dependences of array A is loop-independent and the dependences of array B belongs to the second type.

```
for(i=0;i<32,i=i+1)
   for(j=0,j<64,j=j+1)
{
   B[i]=m0*A[i][j]+m1*A[i][j+1]+m2*A[i][j+2]+m3*
A[i][j+3]+m4*A[i][j+4];
   D[i]=m0*C[i][j]+m1*C[i][j+1]+m2*C[i+1][j]+m3*
C[i+1][j+2]+m4*C[i+2][j];
}
```

Fig.8. An example code

In the rest of this subsection, we describe how to compute the five parameters for the two types of reuse categories.

A. Loop-independent

In this situation, the data will only be reused in the same loop. When current loop is done, the data will never be reused again. We can obtain the following expressions:

$$MEM_{i,bank} = 1;$$
$$MEM_{i,deep} = I_m, if\ (d_1, d_2 \ldots, d_{m-1}) = 0\ and\ d_m \neq 0;$$
$$buffer_{length} = d_m + 1,\ if\ (d_1, d_2, \cdots, d_{m-1}) = 0\ and\ d_m \neq 0;$$
$$buffer_{width} = 1;$$
$$data_{in} = M_{i,deep} * (M_{i,bank} - 1) + buffer_{length};$$

As the data array A[i][j] shown in Fig.8, one 64 depth RAM block are designed, and there are 5 registers needed in smart buffer. And 5 (64*(1-1)+5) data values should be initialized before starting the processing.

B. Loop-carried

In this situation, the data will not only be reused in the same loop, but also in some outer loops. So the number of the largest reused distance data array values will be kept in the internal RAM blocks, and the data that is going to be computed will shift in the registers of smart buffer. We can obtain the following expressions:

$$MEM_{i,bank} = d_m + 1, where\ (d_1, d_2 \ldots d_{m-1}) = 0\ and\ d_m \neq 0;$$
$$MEM_{i,deep} = \prod_{l=1}^{m-1} I_l, if\ (d_1, d_2 \ldots, d_{m-1}) = 0\ and\ d_m \neq 0;$$
$$buffer_{length} = \max\{d_m\} + 1;$$
$$buffer_{width} = M_{i,bank} = d_m + 1;$$
$$data_{in} = M_{i,deep} * (M_{i,bank} - 1) + buffer_{length};$$

As the data array C[i][j] shown in Fig.8, three 64 depth RAM blocks are designed, and the length of the smart buffer is 3, at the same time the width of the smart buffer is also 3, so there are 9 registers in the smart buffer. And 131 (64*(3-1)+3) data values should be initialized before starting the processing.

V. Experiments

This section presents experimental results that characterize the impact of different algorithms for a set of image processing applications written in C: image sharpening (SHARP), Sobel edge detection (SOBEL), and MedianFilter(FILTER).

A. Methodology

There are two parts of work in the experiments. Firstly, in order to prove that our Parameterized architecture can use fewer resources to realize data reuse, we compare three data reuse schemes with the supposed limited resources: (1) no data reuse (2) traditional strategy to data reuse which put all reuse data in smart buffer and (3) our approach. We measured three metrics: (1) the number of required memory elements, (2) the number of registers to exploit data reuse, and (3) the speedup over original programs. Secondly, to testify that the architecture is effective, we compare the synthesis results of our automatic approach with the manual code.

The first part heavily depends on the iteration counts of the loops. For each program, thus, we compare three different problem sizes in terms of iteration count of each loop in a nest as shown in Table 1. We assume there are 32 registers and 1100 memory elements available in the target architecture.

Table 1. Problem size

	SHARP			SOBEL			FILTER		
Problem Size	1	2	3	1	2	3	1	2	3
Outer loop	32	32	320	64	64	640	16	16	160
Inner loop	8	16	32	8	16	32	8	16	32

B. Compared with other data reuse strategies

Table 2 below shows the number of registers in smart buffer required to exploit data reuse in three data reuse schemes.

Table 2. Number of registers

	SHARP			SOBEL			FILTER		
Problem Size	1	2	3	1	2	3	1	2	3
No data reuse	4	4	4	10	10	10	10	10	10
Traditional	17	19	19	25	31	31	25	31	31
Our approach	5	5	5	10	10	10	10	10	10

We have assumed the number of memory elements and registers in target architecture are respectively limited to 1100 and 32, so when the problem require more than 1100 memory elements or more than 32 registers, the two methods will have

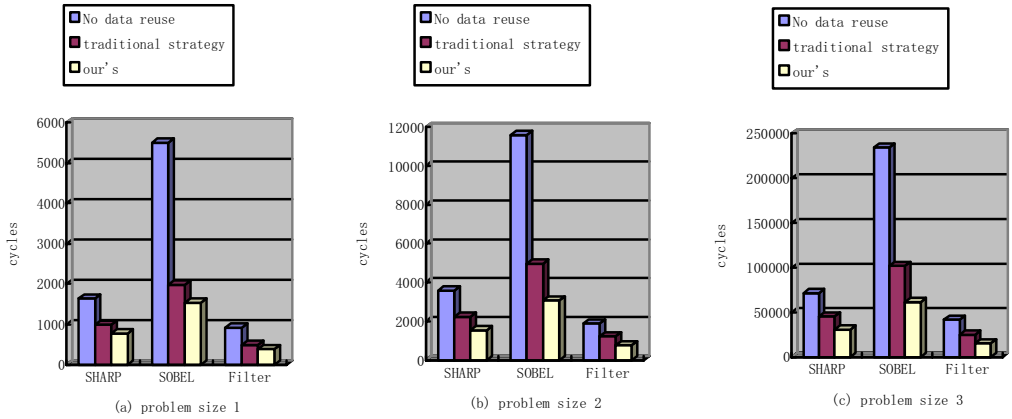

fig.9. Speedups

to partition the data array into smaller blocks. For example, to deal with the problem size 3 of image sharpening, the traditional strategy require 10240 memory elements and 64 registers, so the data array will be partitioned into twelve blocks of size 108×9.

Fig.9 presents the speedup results of overall performance on a single FPGA. Table 3 below shows the number of memory elements required for three data reuse schemes.

Table 3. Number of memory elements

Problem Size	SHARP			SOBEL			FILTER		
	1	2	3	1	2	3	1	2	3
No data reuse	256	512	1056	512	1024	1024	128	256	1088
Traditional	256	288	972	512	640	1090	128	160	820
Our approach	16	32	65	25	48	96	16	48	96

Traditional approach observes speedups from 1.54 to 2.78, and our approach observes speedups from 2.13 to 3.81. The speedup of our approach relative to original and traditional strategy roots in the following sources of benefits:

1. Require less time to initialize internal RAM data values;
2. Design data controller to ensure data transmission and processing perform to execute synchronously.

Another important benefit of our approach is using a less number of registers and memory elements than traditional strategy.

C. *Compared to manual code*

The three test programs have been performed successfully using ModelSim according to our approach and manual code. And the Place and Route (P&R) of the designs is performed with Quartus from ALTERA, and then integrated into a single ALTERA Stratix 80 FPGA. Table 4 below shows the synthesis results of the three test programs using our automatic approach and the manual code, including the resource employed and the clock frequency obtained. The first data in each blank is the result of manual code and the second data is the result of our automatic approach.

In our approach, a data is send to a specific register in smart buffer, such as for Sobel program, the register in smart buffer which a data should be send into is followed strictly, but for SHARP and Filter, the registers to hold the data didn't need to be distinguished definitely, so the synthesis results of the manual code use less resources than our automatic approach but have no much effective on clock speed. From our experimental results, we observe that our approach can achieve a design which is effective as much as manual code.

Table 4. Synthesis results

	SHARP	SOBEL	FILTER
logic elements	636 /651	1082 /1082	1382 /1416
pins	43/43	45/45	44/44
memory bits	16384 /16387	24576 /24576	24586 /24596
clock frequency	103.46 MHz /101.31MHz	139.04 MHz /139.04MHz	111.58 MHz /108.11MHz

VI. Summary and Conclusions

In this paper, we have presented a parameterized architecture to generate the target structure for image processing programs. We employ data reuse to reduce the number of accesses to the data memory. We design special control unit to dominate the dataflow which makes it possible to store a small part of the data values in internal RAM and smart buffer while still providing sufficient memory bandwidth for the custom data path.

We have applied our technique to a set of common signal analysis and image tasks. The results show that the generated memory architecture is able to provide sufficient memory bandwidth for the custom data path using less number of memory elements and registers. It can speed up the processing with less time requiring for initialization.

Acknowledgements

This work was supported by NSFC (National Natural Science Foundation of China) and PCSIRT (Program for Changjiang Scholars and Innovative Research Team in University).

References

[1] Byoungro So, HMary W. HallH, HPedro C. DinizH: 'A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems'. HPLDI 2002H, 165-176.

[2] Gupta, S., Dutt, N.D., Gupta, R.K., and Nicolau, A.: 'SPARK: a high-level synthesis framework for applying parallelizing compiler transformations'. Proc. Int. Conf. on VLSI Design, January 2003.

[3] Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings: 'Sea Cucumber: A Synthesizing Compiler for FPGAs'. M. Glesner, P.Zipf, and M. Renovell(Eds.), FPL 2002, LNCS 2438, pp. 875-885, 2002. Springer-Verlag Berlin Herdelberg 2002.

[4] Weinhardt, M., and Luk, W.: 'Pipeline vectorization', IEEE Trans. Comput.-Aided Des., 2001, 20, (2), pp. 234–248.

[5] Jan Frigo, Maya Gokhale, Dominique Lavenier: 'Evaluation of the StreamsC C to FPGA Compiler: An Applications Perspective'. FPGA 2001, February 11-13, 2001, Monterey, CA.

[6] Hhttp://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfmH.

[7] Heidi Ziegler and Mary Hall: 'Evaluating Heuristics in Automatically Mapping Multi-Loop Applications to FPGAs'. FPGA'05, February 20-22, 2005, Monterey, California, USA.

[8] Mencer, O., Pearce, D.J., Howes, L.W., and Luk, W.: 'Design space exploration with a stream compiler'. Proc. IEEE Int. Conf. on Field Programmable Technology, 2003

[9] Donald Soderman and Yuri Panchul: 'Implementing C algorithms in reconfigurable hardware using C2Verilog'. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), pages 339–342, Los Alamitos, CA, April 1998.

[10] Celoxica, 'Handel-C Language Reference Manual for DK2.0', Document RM-1003-4.0, 2003.

[11] De Figueiredo Coutinho, J.G., and Luk, W.: 'Source-directed transformations for hardware compilation'. Proc. IEEE Int. Conf. on Field-Programmable Technology, 2003.

[12] Takashi Kambe, Akihisa Yamada, Koichi Nishida, Kazuhisa Okada, Mitsuhisa Ohnishi, Andrew Kay, Paul Boca, Vince Zammit, Toshio Nomura,: 'A C-based Synthesis System, Bach, and its Application'.

[13] Daniel D. Gajski, Jianwen Zhu, Rainer D¨omer, Andreas Gerstlauer, and Shuqing Zhao. 'SpecC: Specification Language and Methodology'. Kluwer, Boston, Massachusetts, 2000.

[14] Byoungro So, HMary W. HallH, HHeidi E. ZieglerH: 'Custom Data Layout for Memory Parallelism'. HCGO 2004, 291-302.

[15] Pedro C. Diniz, HJoonseok ParkH: 'Automatic Synthesis of Data Storage and Control Structures for FPGA-Based Computing Engines'. HFCCM 2000H: 91-100.

[16] HJoonseok ParkH, Pedro C. Diniz: 'Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines'. HISSS 2001H: 221-226.

[17] HNastaran BaradaranH, Pedro C. Diniz, HJoonseok ParkH: 'Extending the Applicability of Scalar Replacement to Multiple Induction Variables'. HLCPC 2004H: 455-469.

[18] Z. Guo, B. Buyukkurt, W. Najjar. 'Input Data Reuse In Compiling Window Operations Onto Reconfigurable HardwareH'. Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES), Washington, DC, June 2004.

[19] HAndersson P.H and HKuchcinski K.H 'Automatic Local Memory Architecture Generation for Data Reuse in Custom Data Paths', in Proc. of Engineering of Reconfigurable Systems and Algorithms, 2004.