Optimized Generation of Memory Structure in Compiling Window Operations onto Reconfigurable Hardware

Yazhuo Dong, Yong Dou, and Jie Zhou

School of Computer Science, National University of Defense Technology, Changsha, Hunan, China, 410073
{dongyazhuo,yongdou,zhoujie}@nudt.edu.cn

Abstract. Window operations which are computationally intensive and data intensive are frequently used in image compression, pattern recognition and digital signal processing. The efficiency of memory accessing often dominates the overall computation performance, and the problem becomes increasingly crucial in reconfigurable systems. The challenge is to intelligently exploit data reuse on the reconfigurable fabric (FPGA) to minimize the required memory or memory bandwidth while maximizing parallelism. In this paper, we present a universal memory structure for high level synthesis to automatically generate the hardware frames for all window processing applications. Comparing with related works, our approach can enhance the frequency from 69MHZ to 238.7MHZ.

1 Introduction

FPGA has become the medium of choice for fast hardware prototyping and a popular vehicle for the realization of custom computing machines that target multi-media applications. But developing programs that execute on FPGAs are extremely cumbersome [1]. To deal with the problem, high level synthesis (HLS) tools are developed to implement the hardware system using behavioral level languages, as opposed to register transfer level languages.

HLS tools can be classified 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.

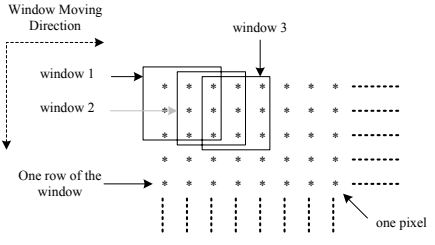
This paper concentrates on one class of applications called window operations. This kind of applications are widely used in signal, image and video processing

P.C. Diniz et al. (Eds.): ARC 2007, LNCS 4419, pp. 110–121, 2007.

and require much computation and data manipulation. Fig.1(a) shows the Sobel edge detection example code in C and the window operations are depicted in fig.1(b). All these operations have similar calculation patterns: a loop or a loop nest operates with array variables. There are multiple references to an array element in the same or a subsequent iteration. Thus, the memory structure can be designed to exploit data reuse.

```
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-1][j+1]+2*img[i][j+1]+img[i+1][j+1]-(img[i-1][j-1]
            +2*img[i][j-1]+img[i+1][j-1];
      uh2= img[i+1][j-1]+2*img[i+1][j]+img[i+1][j+1])-(img[i-1][j-1]
            +2*img[i-1][j]+img[i-1][j+1];
   if ((abs(uh1)+abs(uh2))<threshold)
         edge[i][j]="0xff";
      else
         edge[i][j]="0x00;
   }
}
```

(a)



(b)

Fig. 1. Sobel edge detection algorithm in C code and the window operations

One of general-purposed approaches to exploit data reuse is to identify multiple memory accesses to the same memory location as reused data, and keep these data in a group of registers called smart buffer. It is an important issue how to organize the smart buffer and data layout effectively.

A number of efforts have been carried out to deal with the problem [14][15] [16][17][18]. Some traditional methods use a set of either vertical or exclusively horizontal queues to realize data reuse. They demand a good deal of registers which are the critical resources in FPGA. Since most of them ignored system level scheduling when dealing with external memories, there is a long memory latency to initialize array elements into the internal RAM blocks before starting the processing.

In order to overcome these problems, ROCCC [20][21][22] which is a reconfigurable computing compiler system, presents a new approach to the reuse of data when compiling window operations [19]. It use less number of registers, but it did not realize data reuse completely. Since ROCCC has to access the same location of off-chip memory multiple times, there exist a space for optimization in clock frequency.

In this paper we put forward a new approach to deal with these problems. We execute data accessing to off-chip and on-chip memory overlapping with calculation. We exploit data reuse fully, and at the same time keep the RAM blocks and smart buffer as small as possible.

This paper makes the following contributions:

– It presents a universal parameterized memory structure and a novel data scheduling scheme for window operations.

– It gives the algorithms about how to generate VHDL automatically according to some parameters which are obtained from the compiler.
– It illuminates experimental simulation results for the automatic translation of a set of window processing applications onto FPGA.

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 memory structure of our approach. Section 4 describes the VHDL code generation and the compiler support. Section 5 presents simulation experimental results for a set of window applications. In section 6 we give a conclusion.

2 Background

We now illustrate the use of different storage and control structures in the automatic mapping of an example computation onto a FPGA-based computing engine. The computation is Sobel edge detection written in C as depicted in fig.1(a).

A traditional strategy to reduce the number of required memory accesses is shown in fig.2. Smart buffer holds the data input queues to exploit the fact that consecutive iterations of the inner loop use data that previous iterations have fetched. This strategy layout all reused data in smart buffer, so it uses a large number of registers.

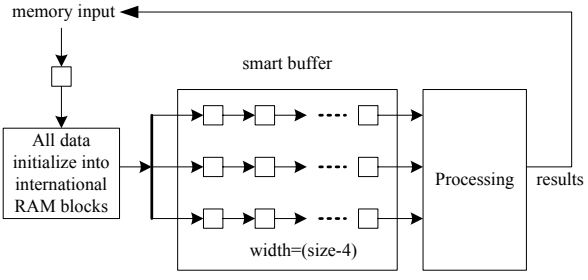


Fig. 2. Traditional strategy to exploit data reuse

The memory structure generated in ROCCC is illustrated in fig.3(a). The smart buffer receives data from the input memory directly, which use less number of registers. The input data used by each outer-loop iteration are loaded only once, but there are overlaps between adjacent outer iterations since they can not afford a smart buffer to hold whole rows of data. It has to access the same location of input memory multiple times which makes the processing speed slow. Fig.3(b) shows the FSM of smart buffer. We notice that ROCCC arranging data in smart buffer in different order at each cycle incurs the complex connection between smart buffer and processing elements. There are twenty-one states needed for Sobel program.

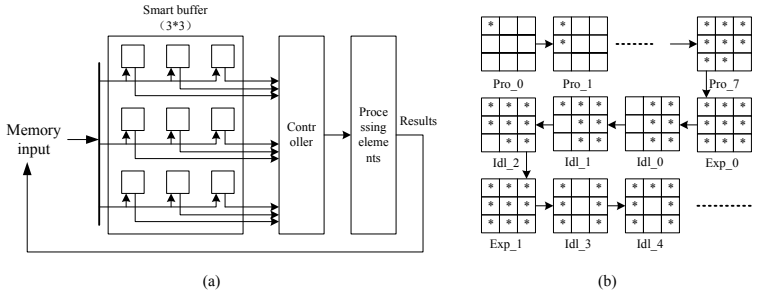


Fig. 3. ROCCC's approach to exploit data reuse and the FSM of smart buffer

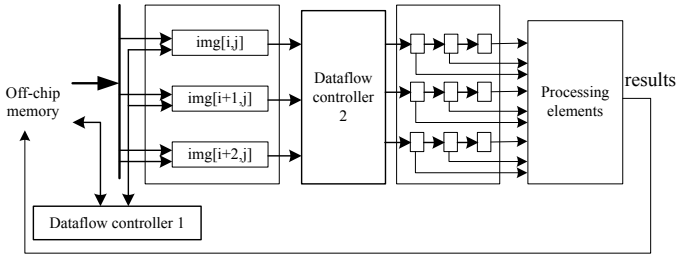


Fig. 4. Our approach to exploit data reuse

In this paper, we propose a novel memory structure, which uses less number of registers than traditional ways and obtains higher speed than ROCCC. Fig.4 shows our approach to resolve Sobel edge detection problem.

There are three RAM blocks with the depth of the inner loop dimension (SIZE-4) designed in the target architecture. One of them holds 1-dimensional img array data. Fig.5 illustrates FSM of the smart buffer. When the data of 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 new input data of row i+3 will take the place of row i. We use shift registers during the whole processing. It can be noticed that we do not initialize all array elements into the on-chip memory before starting the processing. The structure execute in a data driven mode which means starting current operations as soon as possible. There are still pipelined off-chip memory accessing during the whole processing.

Comparing with traditional ways, we use less number of RAM blocks and registers to exploit data reuse. Comparing with ROCCC, we put all reused data in internal RAM blocks. When the data is used later, we can get it from the on-chip RAM blocks but not have to access the off-chip memory. Thus, the processing speed of our approach is faster than ROCCC.

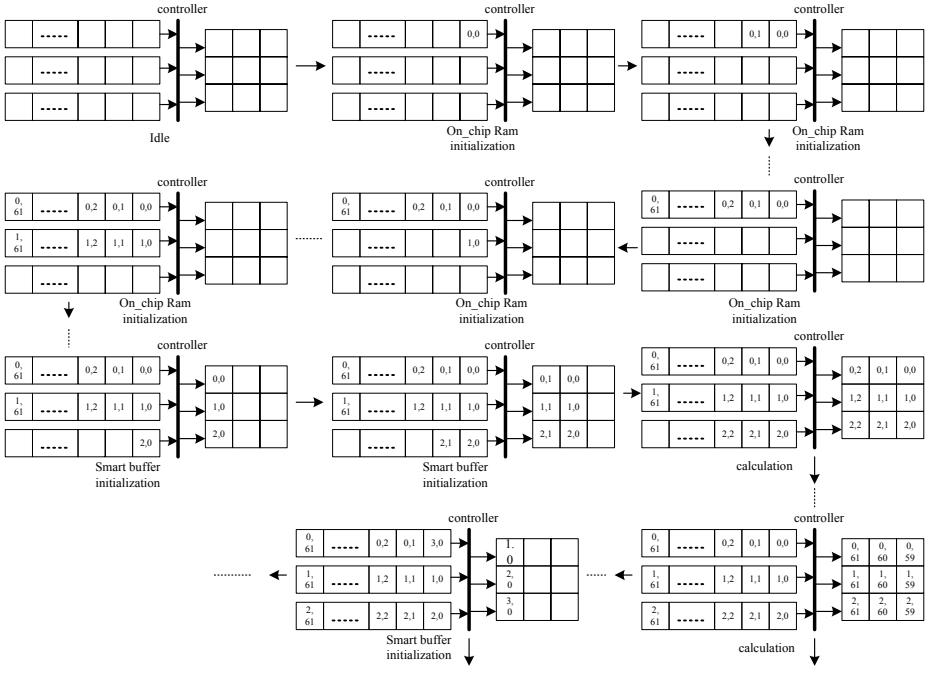


Fig. 5. FSM of on-chip RAM and smart buffer in our approach

3 Memory Architecture

In window operations, the input and output arrays are separate and therefore there is no loop-carried dependency on a single array. Fig.6 presents the universal layout of the target memory structure with which compiler generates VHDL codes for window operations.

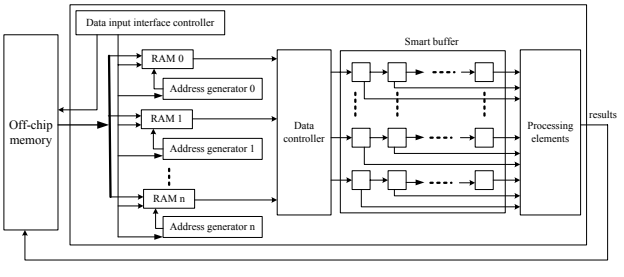


Fig. 6. Overall window operations execution architecture

Data that will be used in the following interations be kept in the RAM blocks until it will never be used again. Data to be used in the current iteration shift in

the registers of smart buffer. Data input interface controller and data scheduling controller keep track of which iterations of the loop are currently in execution, and generate the appropriate control signal to realize the pipelined memory accesses. The address generation unit is a programmable one with auto-increment and auto-decrement capabilities.

4 VHDL Code Generation

In this section, we present our approach to generate efficient VHDL code for the controller and related components. The goal is to minimize run-time control calculation and maximize input data reuse.

4.1 Compiler Support

Window operations have one or more windows sliding over one or more arrays. Both addresses of the read and the write are calculated at compiler time. According to the memory load reference and the loop unrolling parameters, the following parameters are known at compiler time:

1. Starting and ending addresses;
2. The size of array, *width* and *height*;
3. The size of unrolled window, *buffer_width* and *buffer_height*;
4. The unrolled window's strides in each dimension, *buffer_span_width* and *buffer_span_height*;
5. The number of RAM blocks and the depth, *Ram_num*, *Ram_depth*;
6. The number of results generated for once interation, *Result_num*;
7. The number of unrolled times of outer loop, *Control_num*;

Fig.7 gives the unrolled C code of 2D_lowpass_filter. We unroll the loop twice in both horizontal and vertical directions. Therefore, each iteration computes four of these 3*3 windows, and produces a 2*2 output window.

```
for(i=1;i<62;i=i+2) {
    for(j=1;j<62;j=j+2) {
        C[i-1][j-1]=(A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]+A[i][j+1]+A[i+1][j-1]+A[i+1][j]+
A[i+1][j+1])>>3+(A[i][j]>>1)-B[i-1][j-1];
        C[i-1][j]=(A[i-1][j]+A[i-1][j+1]+A[i-1][j+2]+A[i][j+2]+A[i+1][j]+A[i+1][j+1]+
A[i+1][j+2])>>3+(A[i][j+1]>>1)-B[i-1][j];
        C[i][j-1]=(A[i][j-1]+A[i][j]+A[i][j+1]+A[i+1][j+1]+A[i+2][j-1]+A[i+2][j]+
A[i+2][j+1])>>3+(A[i+1][j]>>1)-B[i][j-1];
        C[i][j]=(A[i][j]+A[i][j+1]+A[i][j+2]+A[i+1][j+2]+A[i+2][j]+A[i+2][j+1]+A[i+2][j+2])
>>3+(A[i+1][j+1]>>1)-B[i][j];
        }
}
```

Fig. 7. Motion detection C code, 2*2 unrolled loop

For the C code in fig.7, *width*=62, *height*=62. And for data array A, there are (4*4) registers in smart buffer, *buffer_width*=4, and *buffer_height*=4. The

program is 2*2 unrolled, *buffer_span_width*=2, *buffer_span_height*=2. There are 4 RAM blocks needed, *Ram_num*=4. The depth of RAM is as same as the width of data array A, *Ram_width*=*width*=62. For once interation, there would generate four results, *Result_num*=4. We assume the memory bus in twice the width of the pixel bit-size, then each memory load reads in two pixels. In this example, Ram0 and Ram1 receive data at the same clock period, and Ram2 and Ram3 receive data at the same clock period. The outer loop is unrolled two times, *Control_num*=2. Fig.8 shows the status of array A's smart buffer at different clock cycles.

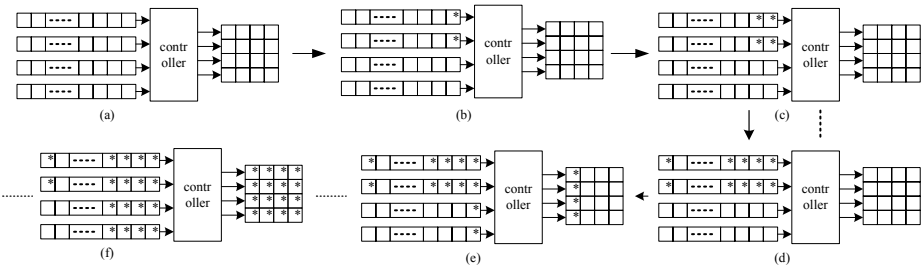


Fig. 8. FSM Status of 2D_Lowpass_filter smart buffer

Fig.9 shows the 5-tap FIR in C which is 1D window operations. In this case, data is only reused in the same loop iteration. We did not design RAM blocks for 1D applications. Smart buffer is enough to keep the reused data. *width*=62, *height*=1, *buffer_width*=5, *buffer_height*=1, *buffer_span_width*=1, *buffer_span_height*=0, *Ram_num*=0, *Ram_depth*=0, *Result_num*=1, *Control_num*=0.

```
For(i=0;i<62;i=i+1){
        B[i]=C0*A[i]+C1*A[i+1]+C2*A[i+2]+C3*A[i+3]+C4*A[i+4];
}
```

Fig. 9. A 5-tap FIR in C

4.2 FSM Generation

The FSM is in charge of the whole processing, determines when the data initialization is finished, traces which register is expired and would be overwritten by new data, determines when a window of data is ready to processing elements, and manages the counterpart relationship between the on-chip RAM and registers in smart buffer. The FSM is assigned one of the four states as shown in fig.5.

- Idle: Waiting for the start signal. When a procedure is started, go to on-chip RAM initialization;
- On-chip RAM initialization: On-chip RAM blocks are in a warm-up state;
- Smart buffer initialization: The smart buffer is collecting data to form the first window. Once all the new data have arrived, the smart buffer goes to processing;
- Processing: In this state, when a window of data is ready, send data in smart buffer to processing elements to calculate. In the next clock cycle, the window data is updated again. The processing keeps going until current row is finished, then the FSM changes to the smart buffer initialization state again to collect new data of the next row.

In the four states, processing state needs some complex controls, because we want to do data transferring and calculation concurrently to speed up the processing. Fig.10 illustrates the parameterized FSM of data transmission between off-chip and on-chip memory. The parameters can be obtained from the compiler as we have discussed in subsection 4.1. Fig.11 gives the parameterized FSM of sending data from on-chip RAM blocks to smart buffer, then to processing elements, and at the same time receiving the results. The FSM also ensures that the whole calculation processing carry through accurately.

5 Experiments

This section presents experimental results that characterize the impact of different methods. We use four window operations as benchmarks: 5-tap FIR, image sharpening, Sobel edge detection (SOBEL), and 2D_Lowpass_filter. These benchmarks are selected for the diversity of size of smart buffer and control structure. 5-tap FIR is a constant-coefficient finite-impulse (FIR) filter. Its source code is given in Fig.9. The size of smart buffer is (1*5). Image sharpening program deals with 2D data array. There are two RAM blocks designed, and the window size is (2*2). Sobel edge detection is shown in fig.1, (3*3) registers are used in smart buffer. The code of 2D_Lowpass_filter is given in fig.7, and the size of smart buffer is (4*4). The input data set of all 1D examples is 256 and the input data set size of all 2D examples is 64*64.

Table 1 shows the number of registers used in smart buffer to exploit data reuse in three methods.

Table 1. Number of registers in smart buffer

Benchmarks	5-tap FIR	Image sharpening	Sobel	2D_Lowpass filter
Traditional approaches	5	128	192	256
ROCCC	5	4	9	16
Ours	5	4	9	16

```
case (sending states)
    1: Primal state: set the Token-Ring to the RAM block  of number
Ring_counter, if the current RAM is the number (height-Control_num), set the
Token-Ring to the first RAM block again;
    2: Send data request signal to the off-chip memory. Ready to receive a new
data;
    3: Receive a data and keep it in the current on-chip RAM block who has the
Token-Ring. The counter increase to note how many data has been received for the
current RAM: counter_num<=counter_num+1;
    4:  Check.
        if (counter_num< width) goto2;
        else the current RAM block is full,
        Ring_counter<=Ring_counter+Control_num, goto 1;
endcase
```

Fig. 10. The FSM of sending data from off-chip memory to on-chip memory

```
case (control states)
    1:Idle state.
        Wait for (height-Control_num) rows of RAM blocks being initialized;
    2: Send data.
        Send data of smart buffer to the processing elements;
    3: Wait for the results;
    4: Send results.
        Increase the result counter,
        result_counter<=result_counter+Result_num;
            If (result_counter== width*buffer_span_height)
                result_counter<=0;
            Increase the row counter which registers how many rows have
been done;
                row_done<=row_done+buffer_span_height;
    5: Increase counter and shift the registers in smart buffer.
        RAM_current_do<=RAM_current_do+1;
        // RAM-current-do records which RAM blocks data is currently being
    dealt with.
                If (RAM_current_do==Ram_num/buffer_span_height)
                    RAM_current_done=1;
    6. Receive the next data.
                If(row_done==height) finish,
                goto 1;
                else goto 2;
endcase
```

Fig. 11. The FSM of data transmission from on-chip RAM blocks to smart buffer and then to processing elements

Traditional appraches hold all reused data in smart buffer. Thus, they need a big smart buffer with a good many registers. ROCCC and our approach only keep calculational data in smart buffer, so the smart buffer is small.

We use the Xilinx ISE 7.1i tool chain to do synthesis and place-and-route reports. The generated VHDL codes are simulated using ModelSim 5.8. The target architecture of all synthesis is Xilinx XC2V8000-5. ROCCC also take 5-tap FIR and 2D_lowpass_filter as benchmarks. Table 2 below gives the simulation results of ROCCC and ours.

Table 2. The synthsis and simulation results of 5-tap FIR and 2D_Lowpass filter

	5-tap FIR		2D_Lowpass_filter		Image sharping	Sobel
	ROCCC	Ours	ROCCC	Ours	Ours	Ours
Smart buffer's statues	14	5	18	4	2	3
Control area (slices)	210	262	542	514	131	210
Clock rate(MHz)	94	238.664	69	238.664	238.664	238.664
Execution time(cycles)	262	263	5980	2057	4042	4108
Throughput(results number/cycles)	0.96	0.96	0.64	1.87	0.98	0.94

Area is the number of slices obtained from place-and-route reports. State is the number of states in the smart buffer's FSM. Clock rate is the clock rate of the whole placed-and routed circuit. Execution time is the number of cycles obtained from the simulation waveforms.

The shift registers in smart buffer of our approach make the connection between smart buffer and processing elements simpler. Thus, the smart buffer's status of ours is less than ROCCC. The whole control area of our approach and ROCCC are almost equal. Our approach can exploit data reuse completely, so the processing speed of ours is much faster than ROCCC, and the clock frequency is much higher than ROCCC. The clock frequency of our approach for the four benchmarks are almost same. It is because we bring forward a universal parameterized memory structure for the window operations. There are only some parameters need to be changed for a different design, and the control components are almost equivalent for all applications.

6 Summary and Conclusions

In this paper, we present a optimization generation memory structure for window operations. We exploit data reuse to reduce the number of accesses to the off-chip memory. We design special control unit to dominate the dataflow which makes it possible to store a small part of the data in internal RAM blocks and smart buffer while still providing sufficient memory bandwidth for the custom data path. We have applied our technique to a set of window processing tasks, and

do some comparisons with related works. The results show that the generated memory structure can speed up the processing using less number of memory resources.

Acknowledgement

This work is sponsored by the National Science Foundation of China under the grant NO. 60633050.

References

1. Byoungro So, Mary W. Hall, Pedro C. Diniz: 'A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems'. PLDI 2002, 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. http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm.
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 Domer, 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'. CGO 2004, 291-302.

15. Gwenole Corre, Eric Senn, Pierre Bomel, Nathalie Julien, Eric Martin:'Memory Accesses Management During High Level Synthesis' CODES+ISSS 2004: 42-47.
16. Pedro C. Diniz, Joonseok Park: 'Automatic Synthesis of Data Storage and Control Structures for FPGA-Based Computing Engines'. FCCM 2000: 91-100.
17. Nastaran Baradaran, Pedro C. Diniz, Joonseok Park: 'Extending the Applicability of Scalar Replacement to Multiple Induction Variables'. LCPC 2004: 455-469.
18. Andersson P.H and Kuchcinski K.H 'Automatic Local Memory Architecture Generation for Data Reuse in Custom Data Paths', in Proc. of Engineering of Reconfigurable Systems and Algorithms, 2004.
19. Z. Guo, B. Buyukkurt and W. Najjar. "Input Data Reuse In Compiling Window Operations Onto Reconfigurable Hardware", Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES 2004), Washington, DC, June 2004.
20. Z. Guo, B. Buyukkurt, W. Najjar and K. Vissers. "Optimized Generation of Datapath from C Codes for FPGAs", Int. ACM/IEEE Design, Automation and Test in Europe Conference (DATE 2005), Munich, Germany, March, 2005.
21. A. Mitra, Z. Guo and W. Najjar. ""Dynamic Co-Processor Architecture for Software Acceleration on CSoCs", Int. Conference on Computer Design (ICCD 2006), San Jose, California, 2006.
22. Z. Guo, W. Najjar and B. Buyukkurt. "Efficient Hardware Code Generation for FPGAs", ACM Transaction on Architecture and Code Optimizations (TACO), (Accepted 2006).