Input Data Reuse in Compiling Window Operations onto Reconfigurable Hardware

Zhi Guo
Electrical Engineering
University of California Riverside

Betul Buyukkurt, Walid Najjar
Computer Science and Engineering
University of California Riverside

{zguo, abuyukku, najjar}@cs.ucr.edu

ABSTRACT

Balancing computation with I/O has been considered as a critical factor of the overall performance for embedded systems in general and reconfigurable computing systems in particular. Data I/O often dominates the overall computation performance for window operation, which are frequently used in image processing, image compression, pattern recognition and digital signal processing. This problem is more acute in reconfigurable systems since the compiler must generate the data path and the sequence of operations. The challenge is to intelligently exploit data reuse on the reconfigurable fabric (FPGA) to minimize the required memory or I/O bandwidth while maximizing parallelism.

In this paper, we present a compile-time approach to reuse data in window-based codes. The compiler, called ROCCC, first analyzes and optimizes the window operation in C. It then computes the size of the hardware buffer and defines three sets of data values for each window: the *window set*, the *managed set* and the *killed set*. This compile-time analysis simplifies the HDL code generation and improves the resulting hardware performance. We also discuss in-place window operations.

Categories and Subject Descriptors
D.3.4 [**Processors**]: Retargetable Compilers; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids; J.6 [**Computer-aided Engineering**]: Computer-aided design (CAD)

General Terms
Design, Performance, Experimentation, Languages

Keywords
Reconfigurable computing, Reuse Analysis, High-level Synthesis, Compilation, VHDL

1. INTRODUCTION

Signal, image, and video processing are among the primary target applications of reconfigurable computing. Window operators are frequently used in these applications. Examples include FIR (finite impulse response) filters in signal processing [1], edge detectors, erosion/dilation operators, texture measures, and spectral operations in image/video processing [2][3]. All these window operators have similar calculation patterns — a loop or a loop nest operates on a window of data (in other word, a pixel and its neighbors), while the window slides over an array, as shown in Figure 1. Figure 2 shows a five-tap FIR filter example code in C. FIR filters are one of the most basic building blocks used in digital signal processing. B[i] is the filter's output and A[i], the input. C0 … C4 are the filter's constant coefficients. If the reconfigurable computing compiler performs a straightforward hardware generation, the functional unit would need to access all five input data values in the current window. This would require a large amount of memory bandwidth and involve pipeline bubbles in the data path.

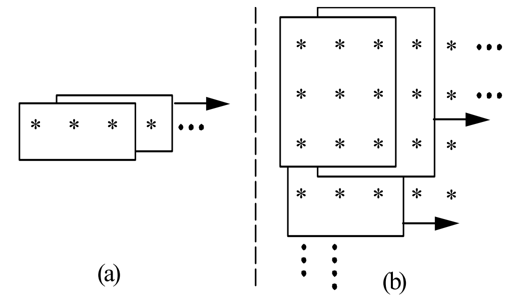

Figure 1 - Window Operations
(a) The Window Operation in An FIR Filter
(b) The Window Operation in An Image

Balancing computation with I/O has been considered as a critical factor of the overall performance for quite some time [4]. When a high-density computation is performed on a large amount

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

Figure 2 - An FIR Filter Example Code In C

of input data, as the case in window operations, data I/O often dominates the overall computation performance. For instance, for the window operations reported in [5], the memory load ratios of reconfigurable computing system over general-purpose processors range from 64 to 112. In other words, the general purpose CPU performed 64 to 112 more load operations than a hand-crafted circuit on an FPGA. Therefore, in order to get high performance, a reconfigurable computing compiler needs to generate *smart* hardware in HDL (hardware description language) to reduce the memory bandwidth pressure by exploiting data reuse when possible. Note that hardware designers routinely do that when handcrafting circuits. The objective of this paper is to automate this process. The challenge is that on FPGAs there is no data path and no pre-designed register files. The compiler must instantiate the data buffer(s) and schedule their accesses, reads and writes. This flexibility is the main reason behind the large reduction in memory operations reported in [5].

However, a reconfigurable computing compiler can't perform an HDL code generation in the way a hardware engineer writes HDL code. For the compiler, the challenge is to be able to intelligently exploit the possibility of data reuse in window operations and automatically generate efficient HDL code tailored for the given input C source code.

The rest of this paper is organized as follows. We introduce ROCCC (Riverside Optimizing Configurable Computing Compiler) system in section two. Related work is introduced in section three. In section four, we present the analysis and optimization on the window operator's C input and the overall hardware architecture as well. Section five gives out our data reuse scheme and the corresponding VHDL code generation. Section six reports on the experimental result. Section seven discusses in-place window operations and section eight concludes the paper.

2. OVERVIEW OF ROCCC

ROCCC compile system, as shown in Figure 3, takes codes written in high level languages, such as C or Fortran, as input and generates HDL code for reconfigurable devices.

The primary target platforms of ROCCC are Configurable Systems-on-a-Chip. CSoC are platforms that consist of an FPGA chip with one or more embedded microprocessors on the chip; both the FPGA fabric, as well as the embedded microprocessors, are essentially programmed using software. The earliest example is that

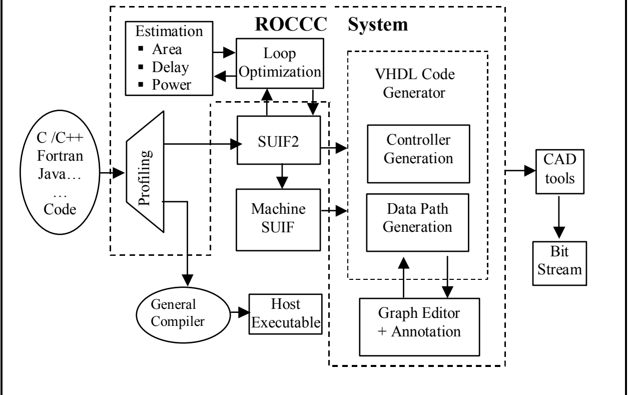

Figure 3 - ROCCC System Overview

of the Triscend E5 followed by the Triscend A7 [7], the Xilinx Virtex II Pro [8], and the Altera Excalibur [9]. The capabilities of these platforms span a wide range. At the low end, the Triscend A7 consists of a 60 MHz ARM CPU with about 20,000 programmable gates. At the high end, the Xilinx Virtex II Pro 2VP125 consists of about 10 million gates, four PowerPC 405 CPUs each running at 400 MHz, 10 Mbits of BlockRAM, 556 18x18-bit multipliers and 3.125 Gbps off-chip bandwidth.

ROCCC's objective is not to compile the whole program, rather to compile the most frequently executing code kernels to FPGAs. It relies on our profiling tool to identify these kernels [6]. The profiling tool uses gcc to obtain a program's basic block counts and identifies, after execution, the most frequently executed loops that form the computation kernel

ROCCC is built on the SUIF2[10][11] and Machine-SUIF[12][13] platforms. The information about loops and memory access is visible in the SUIF's IRs. Therefore, most loop level analysis and optimizations are done at this level. Most of the information needed to design high-level components, such as controllers and address generators is extracted from this level's IRs.

Machine-SUIF is an infrastructure for constructing the back end of a compiler. Machine-SUIF optimizations and analyses do not use machine-specific information directly. Instead, all the machine-specific information is retrieved by calling functions provided by machine-specific libraries. We modify Machine-SUIF's virtual machine (SUIFvm) IR [14] to build our data flow. All arithmetic opcodes in SUIFvm have corresponding functionality in IEEE 1076.3 VHDL with the exception of division. Machine-SUIF's existing passes, like the Control Flow Graph (CFG) library [15], Data Flow Analysis library [16] and Static Single Assignment library [17] provide useful optimization and analysis tools for our compilation system.

A compilation system might be a lifelong project. One emphasis of both SUIF2 and Machine-SUIF is to maximize code reuse. SUIF2 and Machine-SUIF provide frameworks for developing new compiler passes and generating new IRs. They also provide an environment that allows different IRs and different analyses (passes) to be easily combined.

We constrain the source code that will be translated to hardware as follows: no pointer, no floating point, no *break* or *continue* statements.

We have modified SUIF2 and Machine-SUIF and have added new analysis and optimization passes. This group of passes target CSoC devices, analyze and optimize the IRs that come from SUIF2 and Machine-SUIF, and then generate VHDL code. Specifically, taking SUIF2 front end IRs as input, our compiler detects and optimizes memory access. Meanwhile, the compiler also takes Machine-SUIF back end IRs as input and generates the data flow. The array access pattern information, which is obtained through memory reference analysis, combined with the pipeline information, which is created during data flow generation, is fed into the controller generation pass to generate controllers in VHDL. One of the passes, the Graph Editor and Annotation pass, is used to visualize the data flow graph and provide a platform for users to edit the IR directly. As of now, we mainly use this unit to visualize data flow parallelism and to annotate the bitwidth of signals in the data.

In order to efficiently use the available area and memory bandwidth of the reconfigurable devices, our compiler performs regular loop unrolling and strip-mining transformations on loop nests. Another important loop optimization technique when

```
for(i = 1; i<= N - 2; i++) {
 for(j = 1; j<=N - 2 ; j++) {
   MASKv = (P[i-1][j+1] - P[i-1][j-1]) +
        (P[i][j+1] - P[i][j-1]) +  (P[i+1][j+1] - P[i+1][j-
1]);
   MASKh =  (P[i+1][j-1] - P[i-1][j-1]) +
        (P[i+1][j] - P[i-1][j]) + (P[i+1][j+1] - P[i-1][j+1]);
   B[i][j] = ( MASKv*MASKv + MASKh*MASKh );
 }
}
```

Figure 4 - The Edge Detection Algorithm in C Code

targeting an FPGA is loop fusion. Combining adjacent loops helps decrease the execution time of the application. It also, particularly for reconfigurable computing implementations, enhances the reuse of data values fetched from memory. Evidently, it also cuts down by half the required memory bandwidth. An automatic loop unrolling/strip-mining optimizations require compiler time area estimation. The work in [28] shows that compile time area estimation can be done within 5% accuracy and in less than one millisecond. In [30] a compiler algorithm determining unroll factors is presented.

We rely on commercial tools, such as Synplicity [18], to synthesize the VHDL code generated by our compiler.

3. RELATED WORK

Several projects have employed various approaches to translate HLL (high level language) into hardware. SystemC [19] is designed to provide roughly the same expressive functionality of VHDL or Verilog and is suitable to designing software-hardware synchronized systems. Handle-C [20], as a low level hardware/software construction with C syntax, supports behavioral descriptions and uses CSP-style (Communicating Sequential Processes) controlling model. The Nimble [21] compiler targets a general-purpose processor with a dynamically reconfigurable data path.

SA-C [22][23] is a single-assignment high-level language for mapping image processing applications to FPGAs. Because of special constructs specific to SA-C (such as window constructs)and its functional nature, its compiler can easily exploit data reuse for window operations. SA-C compiler performs VHDL code generation by using pre-existing parameterized VHDL library codes. There are a number of control signals between the VHDL components, such as the circular buffers. These control signals, including some feedback control signals, require extra clock cycles and reduce the circuit's performance. Our compiler avoids spending clock cycles on handshaking by doing more compile-time analysis. It takes a subset of C as input and does not involve any non-C syntax.

Streams-C [24] follows the CSP model. Streams-C can meet relatively high-density control requirements. It supports multiple input data streams but doesn't access two-dimension arrays in the way of sliding window. For one-dimension input data vector, such as a one-dimension FIR filter, Streams-C doesn't automatically reuse input data. Programmers manually write data reuse in the input C code.

GARP's [25] compiler is designed for the GARP reconfigurable architecture. The compiler generates GARP configuration file instead of standard VHDL. GARP's memory interface consists of three configurable queues. The starting and ending addresses of the queues are configurable. The queues' reading actions can be stalled. GARP compiler doesn't do loop unrolling and the corresponding input data reuse.

In [31] the authors present an algorithm to map GTM (generalized template matching) operations onto reconfigurable computers. The input representations of the mapping process includes VHDL FPGA component library of the operators and the GTM operation specification. One of the assumptions is the permutability of the operations. ROCCC takes C as input and detects whether the loop nest is permutable. ROCCC generates the VHDL codes of data path and on-chip buffer automatically.

SPARK [26] is another C to VHDL compiler. SPARK takes a subset of C as input and output synthesizable VHDL. Its pre-synthesis transformations include loop unrolling/fusion, common sub-expression elimination, copy propagation, dead code elimination and transformations such as loop-invariant code motion etc. However, SPARK does not support two-dimension array accesses.

4. CODE ANALYSIS AND OPTIMIZATION

In [27] Wolf and Lam presented a loop nest representation and an efficient algorithm to maximize the parallelism of a loop nest for parallel machines. We use a similar representation, however, our compiler is designed for generating VHDL codes specifically targeting reconfigurable devices. In this paper our focus is on analyzing and optimizing window operation loop nests. Therefore, if the compiler can successfully optimize the window operation loop nests it generates correct hardware. Otherwise, the compiler treats the loop nest as common loop nest and exploits other forms of parallelism if available.

4.1 Window Operation Pattern Checking

In most window-based operation, the input and output arrays (or streams) are separate and therefore there are no loop carried dependencies on a single array. We therefore assume separate input and output data buffers on the FPGA. In section seven we discuss in-situ array update.

An example C code is shown in Figure 4. This algorithm is used to detect the edges in an image. It's a common window operation, whose 3×3 window slides on the image. The read buffer is array P and write buffer, array B, as shown in Figure 4. The compiler walks through all the memory references in the SUIF IR and confirms that there is no arrays being both read and written. The compiler also checks the following constraints.

1. Loop counters are assigned and updated only in the loop statements.
2. Each loop counter determines the memory address calculation in only one dimension.

4.2 Scalar Replacement and Data Path

Figure 5 shows the code from Figure 4 after scalar replacement. The Machine-SUIF passes take the highlighted region in Figure 5 as input and exploit instruction level parallelism, synchronize alternative branches, and eventually, generate the *fully pipelined* data path in VHDL. The details of the data path code generation are beyond the scope of this paper.

4.3 The Hardware Architecture

One of the most important characteristics of window operations is that the compiler can decouple the memory accesses from the computations and thereby can maximize data reuse. This feature was shown to heavily influence reconfigurable computing systems' speedup over general-purpose processors [5]. A schematic of hardware architecture for the code in Figure 5 is shown in Figure 6. The *input address generator* generates memory load addresses and feeds the addresses to the on-chip block memory. The smart buffer gets the input data stream from on-chip memory, exploits the data reuse and makes the *window data* available to the data path. By *window data* we mean the data covered by the sliding window on an array or a stream of data for a given iteration or set of iterations (when the loop is fully or partially unrolled). The write buffer collects the results from the data path and presents it to the output memory. The *output address generator* generates memory store addresses. In the general case, we assume that the size of input data does not match that of the output data (in bits) and therefore the two address generators operate at different rates.

```
for(i = 0; i<= N - 2; i++) {
 for(j = 0; j<=N - 2 ; j=j+1)  {
   P_im1_jm1 = P[i-1][j-1];   P_im1_j = P[i-1][j];
   P_im1_jp1 = P[i-1][j+1];   P_i_jm1   = P[i][j-1];
   P_i_jp1   = P[i][j+1];     P_ip1_jm1 = P[i+1][j-1];
   P_ip1_j = P[i+1][j];       P_ip1_jp1 = P[i+1][j+1];
   MASKv = (P_im1_jp1 - P_im1_jm1) + (P_i_jp1 -
P_i_jm1) + (P_ip1_jp1 - P_ip1_jm1);
   MASKh = (P_ip1_jm1 - P_im1_jm1) + (P_ip1_j -
P_im1_j) + (P_ip1_jp1 - P_im1_jp1);
   B_i_j = MASKv*MASKv + MASKh*MASKh;
   B[i][j] = B_i_j;
 }
}
```

Figure 5 - The Edge Detection Code After Scalar Replacement

5. CODE GENERATION

In this section we present our compiler's methods to generate efficient VHDL code. The goal is to minimizing run-time control calculation and maximizing input data reuse by utilizing compile time understanding on the loop nest and the compiler's awareness on the resulting circuit at the clock cycle level.

5.1 The Address Generation

Window operations have one or multiple windows sliding on one or multiple arrays. Both the read and the write array access addresses are known are compile time. At the same time, on-chip memory's access time in terms of clock cycle is known as well. It is possible to generate efficient read/write units for window operations by the compiler.

Take the source code in Figure 4 as an example. According to the memory load references in Figure 5 and the loop optimization parameters, the following parameters are known at compile time.

1. Stating and ending addresses
2. The number of clock cycles between two sequential memory accesses
3. The unrolled window's size
4. The unrolled window's sliding strides at each direction
5. The array's row size
6. The starting address-difference between two adjacent outer iterations.

We have designed a parameterized FSM (finite state machine) in the VHDL library to be used as the address generator. All the parameters above are the FSM's inputs. Notice that these parameters are redundant. For example, parameter 6 can be deduced from others. If we unroll twice the outer loop of the code in Figure 5, the array reference addresses are generated in the order shown in Figure 7. The compiler parameterizes the input and output address generators and exports the corresponding memory access units. The compiler also needs to take care of the fact that the output array's size is a little bit smaller than the input array size. The generated memory access units does not waste any cycle on rewinding address in either column direction or row direction, which is the performance advantage of handcraft VHDL.

5.2 The Smart Buffer Generation

Essentially elements of an array are stored linearly in memory. For window operations, the data elements are fetched sequentially in the order of, for example, shown in Figure 7. In order to reuse the input data, the compiler needs to design a buffer, which has the ability to intelligently fulfill the following tasks:

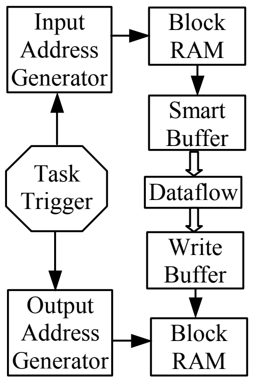

Figure 6 - The Overview Architecture of the Edge Detection Code

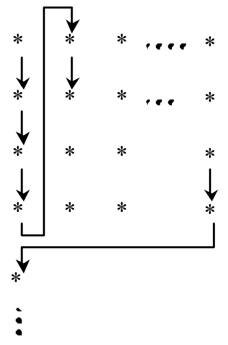

Figure 7 - Reading Order of the Edge Detection Code

- Buffering the input data stream and exporting, to the data path, a complete data window once it becomes available.
- Managing the storage utilization of the buffer by keeping live data and clearing unused data.

Our compiler relies on the six parameters of Section 5.1 to generate the smart buffer. The smart buffer is not implemented as a circular shift register where the valid data is presented in the same location every cycle. Instead, the compilers embodies the control signals in the FSM to export the set of valid data every cycle without having data being shifted.

5.2.1 The Buffer

The smart buffer has the same number of rows as the window in the unrolled inner loop. The number of columns should satisfy the following conditions.

$$\text{Column}_{\text{buffer}} \bmod \text{Stride}_{\text{h}} = 0$$

$$\text{Column}_{\text{buffer}} \bmod \text{WordLength}_{\text{mem}} = 0$$

$$\text{Column}_{\text{buffer}} > \text{Column}_{\text{window}} + \text{WordLength}_{\text{mem}}$$

where

$\text{Column}_{\text{buffer}}$: the number of columns of the smart buffer,

Stride_{h} : the stride of the sliding window in horizontal direction in both Figure 7 and Figure 8.

$\text{WordLength}_{\text{mem}}$: the width of memory I/O.

$\text{Column}_{\text{window}}$: the number of columns of one window.

The unit of all the parameters is the number of pixels. The first condition ensures that the leftmost column in the smart buffer is always the start of the next stride of window. The second condition ensures that input data can be directly stored into a set of bundled buffer elements. Tokens are used to indicate current open locations to the new input data. We will introduce the tokens later on. The third condition makes sure that new input data does not overwrite live buffered data.

Figure 8 shows the *smart buffer* of the edge detection code (inner loop is unrolled twice) in Figure 5. In this figure, we assume that the memory I/O bus is 16 bits wide (a word) and each pixel is 8 bits. According to the reading order in Figure 7, the smart buffer get the number 0 ellipse (word), which has two pixels, then the number 1 ellipse and so on. The number 12 ellipse overwrites the number 0 ellipse.

We use *tokens* in the buffer to have the circuit automatically determine the location of new coming input data. At the very beginning, only the two elements in the number 0 ellipse have tokens. Once these two elements get and store the input data, the tokens are passed to the next ellipse, number 1 ellipse, at the same time. The number 11 ellipse passes the tokens to number 0 ellipse.

5.2.2 The Sliding Windows

The Boolean signal *token* of each buffer element indicates if the next new input data should be stored into the corresponding element. Each buffer element also has another signal, *live,* to indicate if the data in the element is valid or not. It is set when new data is stored .

Based on the window's size and sliding strides, the compiler determines the elements that form a window. For instance, in Figure 8, the elements in ellipse 0, 1, 2, and the left elements in ellipse 4, 5, 6 belong to window 0. These nine elements comprise the *managed set* of window 0. Notice that the *managed set* of window 11 covers three element of the rightmost column and six elements of the two leftmost columns. Once the circuit detects that within one *managed set,* all the elements' *live* signals are set,

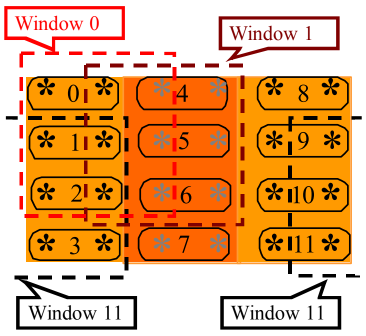

Figure 8 - Smart Loading Buffer of The Edge Detection Code

these elements are exported to the buffer's output ports. Thus, this window's availability is asserted and the data is fed into the data path. Note that the order of the window's availability is know at compile time.

Each window in Figure 8 also has a *kill set,* which consists of the elements that are not needed (dead) once this window's data has been used. For example, window 0's *kill set* consists of only the left element of ellipse 0, while window 11's *kill set* consists of the right three elements of ellipse 9, 10, 11. Once a window is asserted *available*, the *live* signals of the elements in the *kill set* are reset.

```
for (i=0; i<7; i=i+1)
  for(j=0; j<7; j=j+1)
    if (A[j]<A[j+1])
      {
      temp = A[j];
      A[j]=A[j+1];
      A[j+1] = temp;
      }
```

Figure 9 - A Bubble Sort Code in C

```
for (i=0; i<=18; i=i+1)
  for(j=max(0,(i-6)/2); j<=min(6,i/2); j=j+1)
    if (A[i-2j]<A[2-2j+1])
      {
      temp = A[i-2j];
      A[i-2j]=A[i-2j+1];
      A[i-2j+1] = temp;
      }
```

Figure 10 - The bubble Sort Code After Loop Transformations

The compiler first does all this analysis and generates the *managed set* and the *kill set*. Then, the VHDL code generation is performed based on this analysis. The VHDL code, by itself, doesn't need to have the concepts of the *windows* and *sets*. The VHDL code only describes the logical and sequential relationship between signals/registers.

The compiler reduces handshaking signals between components, and therefore, saves clock cycles spent on doing Boolean calculation on the handshaking signals. In other words, this approach's compile time analyses on the result circuits, such as the *windows* and the *sets*, shift run time control burden to compiler.

6. EXPERIMENTAL RESULTS

We use three window operations as benchmarks: FIR filter, edge detection and wavelet transform. FIR (finite impulse response) filter is a one-dimension window operator, whose window size is 1×n, where n is the number of data used in one iteration. The edge detection code is shown in Figure 4. The wavelet transform is also a two-dimension window operator, whose window size is 5×5 and strides are two in both X and Y direction.

For each benchmarks, the compiler generates the smart buffers and the data path in VHDL, as the architecture shown in Figure 6. The compiler also gives the parameters for the parameterized address generators.

Table 1 - Experimental Results of the Smart Buffers

Benchmark	Window Size	Stride		Buffer Size	# of Slice	# of Clock Cycle	Pipeline Latency
		X	Y				
FIR	1×5	N/A	1	8	263	128	5
Edge Detection	3×3	1	1	36	1,355	48384	6
Wavelet Trans.	5×5	2	2	88	2,612	43648	5

In all these three benchmarks, I/O width is 16-bit and data width is 8-bit. The input of FIR is a vector of 256 8-bit data. The input image of both edge detection and wavelet transform examples is a 256×256 8-bit array. We unroll the inner loop four times. In Table 1, the second column shows the original window size before loop unrolling. The third and fourth columns list the strides of the sliding window in both X and Y directions. The number of registers in the smart buffer is shown in the fifth column. The sixth column gives out device utilization as the number of slices used for the buffer on a Xilinx Virtex xcv2000E chip, which account for 1%~13% of the whole chip. Notice that xcv2000E is just a mid-sized FPGA at present. The seventh column list the numbers of clock cycles to read and buffer the input data. Because the data path is fully pipelined and free of bubbles, the number of clock cycles for the entire calculation is equal to the number of clock cycles for memory access plus the pipeline latency, which is listed in the last column.

We have compared our results to another C to VHDL compiler: SPARK [26] but only for the FIR case. SPARK does not support two-dimensional arrays. SPARK's resulting VHDL code requires 1765 clock cycles to compute on the same size of input data set. ROCCC gets 13.8 (1765÷128) times speedup. If we factor out the input bus width difference (ROCCC is 16-bit while SPARK is 8-bit), the speedup is 6.9. ROCCC compiler's analysis and optimizations are aware of the bus and data widths and exploit this information to maximize parallelism. This same feature is also in Streams-C. The essential reason of the speedup is that ROCCC tailors the optimizations according to the C source code's characteristic and therefore utilizes the potential parallelism of window operations, while SPARK treats the input code as a general one. For instance, SPARK doesn't reuse the input data. The number of clock cycles of SPARK's resulting VHDL code varies according to the FIR's number of taps, while our compiler's resulting VHDL code doesn't spend more cycles even if we increase the number of taps since our resulting circuit reuses the data in the input stream.

7. DISCUSSION

This paper mainly reports our compiler's approaches and performance of dealing with the situation that the window operators read input data from one array and write the output to other array. But the ROCCC compiler doesn't exclude window operations updating array in place if there is potential parallelism. ROCCC follows the approaches in [27].

Figure 9 shows an example code of bubble sort. Every innermost iteration works on a 1 × 2 window in place and the window slides on the one-dimension array to be sorted. A[j] = f{A[j], A[j+1]} gives rise to distance vectors (1, 0) and (1, -1) and, A[j+1] = f{A[j], A[j+1]} gives rise to distance vectors (0, 1) and (1, 0). Therefore, the original code's distance vectors are

$$D = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & -1 \end{bmatrix}.$$

Using skewing and waveform transformations we change the original distance vectors to

$$D' = TD = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 1 \end{bmatrix},$$

$$\text{where } T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}.$$

Matrix T consists of two transformations: $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ skews the shape of the iteration space and $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ performs a frontwave transformation to obtain one degree of parallelism.

The transformed loop nest is shown in Figure 10. We observe that there is no data dependence between any innermost loops. Therefore, in each outermost loop, the address generator in Figure 6 can keep on feeding reading addresses to the memory without any pipeline bubble. Notice that this time, the reading memory and writing memory are the same bank of dual-port memory. The legality of the loop transformations and the compiler's awareness of the data path at clock cycle level ensure that the dual-port memory is free of accessing hazard.

8. CONCLUSION

Reconfigurable computing is an emerging and promising technology that allows part of a computation to be mapped onto a reconfigurable fabric such as an FPGA. The most challenging aspect of reconfigurable computing is the compilation of high-level language programs to HDL. So far, hand-crafted VHDL code is twice as fast as most compiler generated ones [5][29].

In this paper, we present our reconfigurable computing compiler system, ROCCC. ROCCC, an open framework built on the SUIF platform, takes high-level languages, such as C/Fortran, as input and generated VHDL codes for reconfigurable devices.

We have identified the reuse of data elements that are fetched from memory to the FPGA as one area where hand-crafted VHDL codes perform much better. This paper presents a new approach to the reuse of data when compiling window operations. We describe the compiler's analysis and optimization on the memory accesses of the C input codes. We propose a compile-time scheme that generates a *smart buffer* for storing all the fetched data elements based on window size, stride, data size, memory access size etc. The smart buffer design does not rely on shifting data or circular buffers. Instead, a controller keeps track of the managed set and the kill set of data in every iteration. The newly fetched data replaces the kill set.

The experimental results show that our scheme does not imply any extra idle cycles. In other words, the computation requires exactly as many cycles as fetching the data from memory does. This paper also discusses the situations of window operations working on an array in-situ.

9. ACKNOWLEDGMENTS

This work was supported in part by NSF ITR Award No. 0083080 and by the Los Alamos National Laboratory.

10. REFERENCES

[1] A. V. Oppenheim, R. W. Schafer, Discrete-Time Signal Processing. Prentice-Hall , Inc. 1989

[2] R. C. Gonzales, R. E. Woods, Digital Image Processing. Prentice-Hall Inc. 2002

[3] A. M. Tekalp. Digital Video Processing. Prentice-Hall Inc. 1995.

[4] H. T. Kung. Why Systolic Architectures? IEEE Computer. Vol. 15, No. 1 (Jan. 1982), pp. 37-46

[5] Z. Guo, W. Najjar, F. Vahid and K. Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs over Processors, Int. Symp. Field-Programmable gate Arrays (FPGA), Monterrey, CA, February 2004.

[6] D. C. Suresh, W. A. Najjar J. Villareal, G. Stitt and F. Vahid. Profiling Tools for Hardware/Software Partitioning of Embedded Applications. Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES 2003), San Diego, CA, June 2003.

[7] Triscend Corporation, "Triscend A7 Configurable System on a Chip Family." http://www.triscend.com/products/a7.htm 2004

[8] Xilinx Corp. "IBM and Xilinx Team." http://www.xilinx.com/prs_rls/ibmpartner.htm 2004

[9] Altera Corp. "Excalibur: System-on-a-Programmable." http://www.altera.com 2004

[10] SUIF Compiler System. http://suif.stanford.edu, 2004

[11] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, C. Sapuntzakis. An Overview of the SUIF2 Compiler Infrastructure. Computer Systems Laboratory, Stanford University.

[12] Machine-SUIF. http://www.eecs.harvard.edu/hube/research/machsuif.html, 2004

[13] M. D. Smith and G. Holloway. An introduction to machine SUIF and its portable libraries for analysis and optimization. Division of Engineering and Applied Sciences, Harvard University.

[14] G. Holloway and M. D. Smith. Machine-SUIF SUIFvm Library. Division of Engineering and Applied Sciences, Harvard University 2002.

[15] G. Holloway and M. D. Smith. Machine SUIF Control Flow Graph Library. Division of Engineering and Applied Sciences, Harvard University 2002.

[16] G. Holloway and A. Dimock. The Machine SUIF Bit-Vector Data-Flow-Analysis Library. Division of Engineering and Applied Sciences, Harvard University 2002.

[17] G. Holloway. The Machine-SUIF Static Single Assignment Library. Division of Engineering and Applied Sciences, Harvard University 2002.

[18] Synplicity, Inc. http://www.synplicity.com/ 2004

[19] SystemC Consortium. http://www.systemc.org 2004

[20] Handel-C Language Overview. Celoxica, Inc. http://www.celoxica.com 2004

[21] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In Design Automation Conf. (DAC), 1999.

[22] W. Najjar, W. Böhm, B. Draper, J. Hammes, R. Rinker, R. Beveridge, M. Chawathe and C. Ross. From Algorithms to

Hardware - A High-Level Language Abstraction for Reconfigurable Computing. IEEE Computer, August 2003.

[23] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems, The Journal of Supercomputing, Volume 21, pages 117-130, 2002.

[24] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Lalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In IEEE Symp. on FPGAs for Custom Computing Machines (FCCM), 2000.

[25] T. J. Callahan, J. R. Hauser, J. Wawrzynek. The Garp Architecture and C Compiler. IEEE Computer, April 2000.

[26] SPARK. http://www.cecs.uci.edu/~spark/ 2004

[27] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. IEEE Transactions on Parallel and Distributed Systems, 2(4): 452--470, October 1991.

[28] D. Kulkarni, W. Najjar, R. Rinker, and F. Kurdahi, Fast Area Estimation to Support Compiler Optimizations in FPGA-based Reconfigurable Systems, IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, April 2002.

[29] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective. Ninth ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, 2001.

[30] B. So, M. W. Hall and P. C. Diniz, "A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems", Int. Symp. On Programmong Language Design and Implementation (PLDI) 2002

[31] X. Liang, J. Jean, Mapping of Generalized Template Mapping on Reconfigurable Computers. IEEE Trans. on VLSI System, 11(3): 485-498, 2003