

Efficient Hardware Code Generation for FPGAs

ZHI GUO, WALID NAJJAR, BETUL BUYUKKURT
University of California, Riverside

The wider acceptance of FPGAs as a computing device requires a higher level of programming abstraction. ROCCC is an optimizing C to HDL compiler. We describe the code generation approach in ROCCC. The smart buffer is a component that reuses input data between adjacent iterations. It significantly improves the performance of the circuit and simplifies loop control. The ROCCC-generated data-path can execute one loop iteration per clock-cycle when there is no loop-dependency or there is only scalar recurrence variable dependency. ROCCC's approach to supporting while-loops operating on scalars makes the compiler able to move scalar iterative computation into hardware.

Categories and Subject Descriptors: B.5 [Register-transfer-level Implementation]; B.5.2 [Design Aids]; C.3 [Signal Processing Systems]

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Reconfigurable Computing, High-level Synthesis, Data Reuse, FPGA, VHDL

1. INTRODUCTION

Continued increases in integrated circuit chip capacity have led to the recent introduction of the Configurable System-on-a-Chip (CSoC), which has one or more microprocessors integrated with a field-programmable gate array (FPGA) and memory blocks on a single chip [23][2][24]. The capabilities of these platforms span a wide range, having the flexibility of software along with the efficiency of hardware. They combine on one chip the *sequential* and the *spatial* computation models: The sequential parts of an application execute on the microprocessor, while the compute intensive computations, which typically consist of parallel loops, are mapped as circuits on the FPGA. In other words, the FPGA acts as a configurable hardware accelerator or co-processor to the microprocessor itself. Speedups ranging from 10x to 1000x over microprocessors have been reported for a variety of applications including image and signal processing, DNA string matching and protein folding [25][16][4]. Such speedups are the result of two main factors: large-scale parallelism and customized circuits. Applications such as signal, image and video processing exhibit very large amounts of parallelism, so mapping such a computation to a circuit can drastically improve its efficiency as compared to a traditional microprocessor. These factors have been described and quantitatively evaluated in [7].

The main problem standing in the way of wider acceptance of CSoC platforms is their programmability. Currently, application developers must have extensive hardware expertise, in addition to their application area expertise, to develop efficient designs that can fully exploit the potential of CSoC. Designing and mapping large applications onto

FPGAs is a long and tedious task that involves a large amount of low-level design in a Hardware Description Language (HDL). To bring CSoCs into the mainstream, tools are needed that would map applications expressed in a High-Level Language to an efficient circuit in HDL.

Optimizing compilers for traditional processors have benefited from several decades of extensive research that has led to extremely powerful tools. Similarly, electronic design automation (EDA) tools have also benefited from several decades of research and development leading to powerful tools that can translate VHDL and Verilog code, and recently SystemC [27] code, into efficient circuits. However, little work has been done to combine these two approaches. Several projects have implemented various types of HLL to HDL translations (GARP [3], Streams-C [6], SA-C [19], DEFACTO [30], SPARK [26], Handel-C [11] etc.). Two papers [5][8] have reported on the performance gap between compiler-generated VHDL and hand-crafted VHDL for medium size codes. In both cases it is reported that the hand-crafted versions ran twice as fast.

ROCCC (Riverside Optimizing Configurable Computing Compiler) is a second-generation compilation tool targeting CSoC leveraging on our prior experience with SA-C [19]. It takes high-level code, such as C or FORTRAN, as input and generates RTL VHDL code for FPGAs. One of its objectives is to bridge the above-described performance gap. Compiling to FPGAs is challenging. Traditional CPUs, including VLIW, have a fixed hardware structure with pre-determined resources, such as ALUs and registers, and a protocol to use these resources, the instruction set architecture (ISA). FPGAs, on the other hand, are completely amorphous. The task of an FPGA compiler is to generate both the data-path and the sequence of operations (control flow). This lack of architectural structure, however, presents a number of advantages:

- (1) The parallelism is very high and limited only by the size of the FPGA device or by the data memory bandwidth in and out the FPGA. Therefore loop transformations that can maximize the parallelism are of paramount importance.
- (2) On-chip storage can be configured at will: registers are created by the compiler and distributed throughout the data-path where needed, thereby increasing data reuse and reducing accesses to memory.
- (3) Circuit customization: the data-path and sequence controller are tailored to the specific computation being mapped to hardware. Examples include pipelining and customized data-path bit-width.

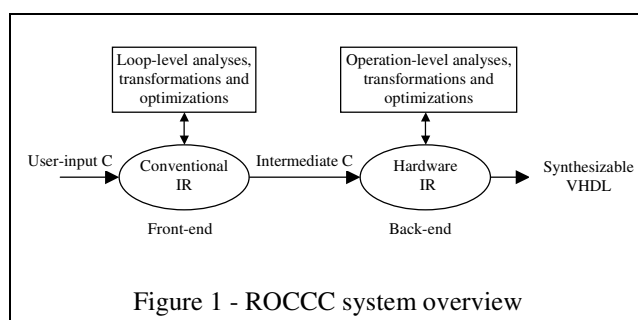
In this paper we focus on the last two points. In previous work [10], we described our approach for the generation of pipelined data-paths for do-all for-loops. In our implementation a new instance of the loop body is started each cycle in the data-path. In order to sustain that throughput we must have a storage mechanism that is capable of feeding the data-path with the required data. In [9], we introduced the *smart buffer*, an interface between on-chip memory and the loop data-path, whose objective is to minimize the number of data re-fetches from memory.

This paper complements and extends our previous work [9][10]. In addition to the code generation of parallel (*for*) loops we present and demonstrate through examples our approach for sequential (*while*) loops' data-path generation. We describe a novel and improved implementation of the smart buffer [9] that (1) supports loops having multiple input and output arrays, (2) is more area and clock cycle efficient. The smart buffer, by reusing the previously fetched data, makes the most of the memory bandwidth and minimizes the stall cycles of the pipelined data-path.

The rest of this paper is organized as follows: the next section presents an overview of the ROCCC compiler framework. Section 3 presents the data-path generation for do-all for-loops and while-loops. Results for each loop type are given in the subsections where the corresponding data-path generation is described. Section 4 introduces the smart buffer. Section 5 discusses related work. Section 6 concludes the paper.

2. ROCCC OVERVIEW

ROCCC is built on the SUIF2 [1][20] and Machine-SUIF [21][18] platforms. Figure 1 shows ROCCC's system overview. It compiles code written in C/C++ or Fortran to VHDL code for mapping onto the FPGA fabric of a CSoC device. In the execution model underlying ROCCC, sequential computations are carried out on the microprocessor in the CSoC, while the compute intensive code segments are mapped onto the FPGA. These typically consist of loop nests, most often parallel loops, operating on large arrays or



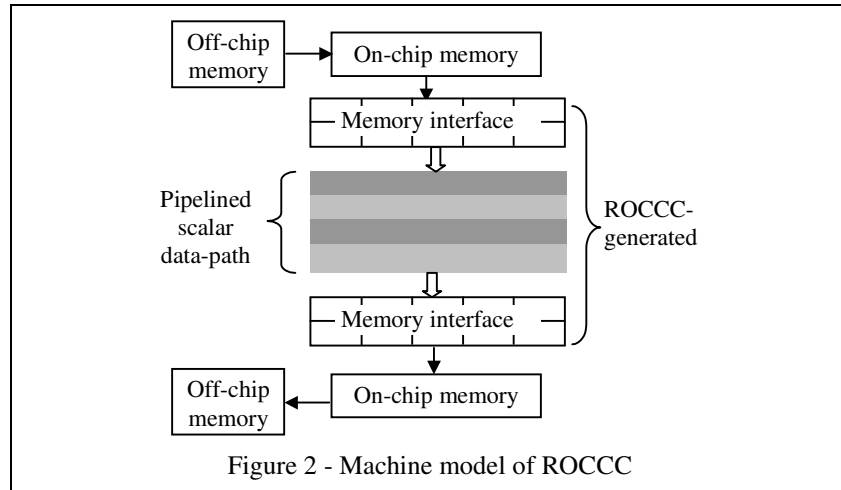


Figure 2 - Machine model of ROCCC

streams of data. The front-end of ROCCC performs a very extensive set of loop analysis and transformations aiming at maximizing parallelism and minimizing the area. The transformations include loop unrolling and strip-mining, loop fusion and common sub-expression elimination across multiple loop iterations¹. Most of the information needed to design high-level components, such as controllers and address generators, is extracted from this level's IRs.

The machine model of ROCCC, shown in Figure 2, consists of on-chip memories (BRAM on the Xilinx architecture), memory interfaces and a pipelined scalar data-path. The scalar data-path accesses memory only through memory interfaces. The compiler performs scalar replacement transformation at front-end. Figure 3 shows a simple example. The compiler converts the code segment in Figure 3(a) into the segment in Figure 3(b), separating memory accesses from computations. Figure 3(c) is the hardware implementation of the highlighted segment (the scalar data-path).

ROCCC uses Machine-SUIF virtual machine (SUIFvm) [12] intermediate representation as the back-end IR. The original SUIFvm assembly-like instructions, by themselves, cannot completely cover HDLs' functionality. For example, the statement ($\text{sum} = \text{sum} + x$) in a loop body will result in a loop carried dependency. On FPGAs, this dependency can be simply removed by inserting a feedback variable between two adjacent pipeline stages. But the SUIFvm assembly-like instruction set does not have an equivalent operation to describe this behavior. To compensate for this lack, ROCCC

¹ These transformations are beyond the scope of this paper.

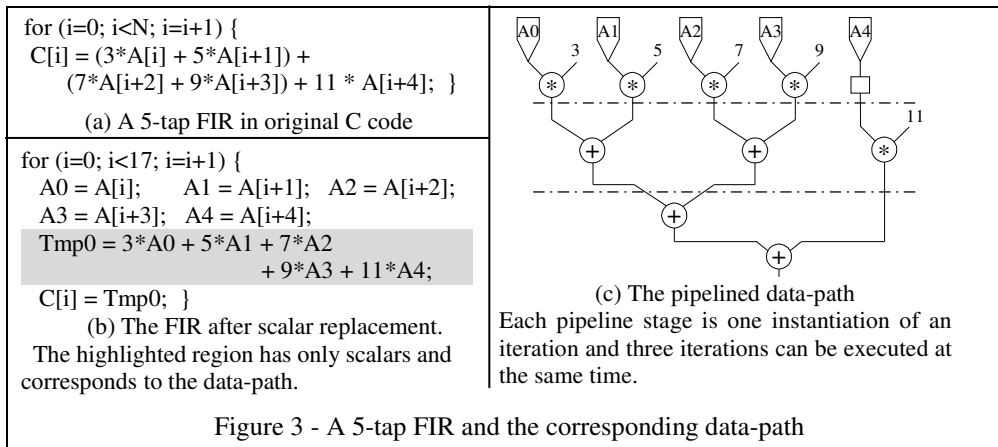


Figure 3 - A 5-tap FIR and the corresponding data-path

performs high-level data flow analysis at the front-end and the analysis information is transferred through pre-defined macros to assist the back-end hardware generation.

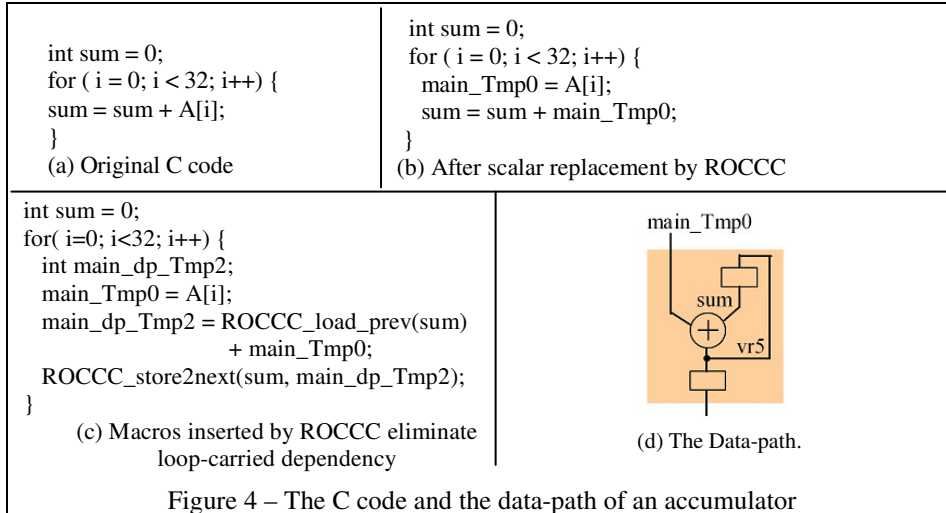
The front-end's optimized output is fed into Machine-SUIF to generate low-level IRs. Machine-SUIF is an infrastructure for constructing the back-end of a compiler. We modify Machine-SUIF's virtual machine (SUIFvm) Intermediate Representation (IR) [12] to build our data flow. All arithmetic opcodes in 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 [13], Data Flow Analysis library [14] and Static Single Assignment (SSA) library [15], provide useful optimization and analysis tools for our compilation system. After applying SSA, control flow graph information is visible and every virtual register is assigned only once.

After back-end analyses and optimizations, the compiler generates VHDL code. We rely on commercial tools, such as Synplicity [22], to synthesize the VHDL code generated by our compiler.

We constrain the source code that will be translated to hardware, which is loop nests, as follows: no pointers, no *break* or *continue* statements, and all memory addresses must be resolvable at compile-time.

3. DATA-PATH GENERATION

In terms of executing a loop in high-level languages, such as C, on reconfigurable fabric, FPGA's most significant advantages are its lack of pre-designed structure and its capacity for parallelism, both at the loop and instruction levels. One of its main weaknesses is the inefficiency of the automatic generation of custom control logic. For parallel loops, the main objectives of our data-path code generation are:



1. Exploiting potential loop-level and instruction-level parallelism.
2. Simple control of the generated data-path.
3. Pipelining to achieve maximum throughput.

This section describes our approach for the compiler’s data-path generation.

3.1 Preparation Passes

After applying scalar replacement and front-end dataflow analysis, the program, such as the code shown in Figure 4(c), is passed to Machine-SUIF. ROCCC performs circuit level optimizations and eventually generates the data-path in a modified version of the Machine-SUIF virtual machine intermediate representation. Figure 4(b) shows an accumulator after applying scalar replacement in C. The variable *sum* is detected as a recurrence variable and will be a feedback signal in hardware. Figure 4(c) shows the resultant code segment in C. The macros *ROCCC_load_prev()* and *ROCCC_store2next()* implement the recurrence.

Macros are converted into ROCCC-specific opcodes. For example, *ROCCC_load_prev()* and *ROCCC_store2next()* in Figure 4(c) are converted into instructions with opcodes *LPR* (load previous) and *SNX* (store next), respectively. *LPR* is implemented as a feedback wire and *SNX* is implemented as a register. This pair of instructions duplicates the variable of the present iteration to the next one and removes the loop carried dependency. Lookup-table macros are also converted into corresponding LUT instructions.

3.2 Building the Data-Path

Each instruction that goes to hardware is assigned a location in the data-path. We add a new field, $[n]$, as shown in Equation 1, into Machine-SUIF IR to record the location of each arithmetic, logic or register copying instruction in the data-path. We call this location the execution level. The higher level an instruction is located, the earlier it is executed.

$$[n][m] \quad \text{add } \$vr1.s32 < -\$vr2.s32, \$vr3.s32 \quad (1)$$

The compiler groups the instructions in each node into different execution levels to exploit instruction (operation) level parallelism. Instructions at the same level are executed simultaneously. Additional *mov* instructions are added where needed as pass-through nodes. Each instruction's location in the data-path satisfies the following requirements:

- If an instruction's source operand(s) is the live-in operand of this node, the instruction must be at the top level of the data-path. If an instruction's destination operand is the live-out operand of this node, the instruction must be at the bottom level of the data-path.
- An instruction's source operands are the destination operands of the instructions one level higher.
- If a live-in operand is also in the live-out operand set, it is copied down to the bottom level.
- Mux nodes are added to implement if-conversion. Latch nodes are added to copy live operands from a branch-node's preceding node down to their succeeding node. Alternative branches of the data-path have the same number of levels.

At this point, every level of the dataflow graph corresponds to the instantiation of one loop iteration. Superfluous *mov* instructions are eliminated by the synthesis tool.

ROCCC automatically places latches to pipeline a data-path. Each execution level is marked as either *latched* or *un-latched*, according to the estimated sum of the signal propagation delay from the most recently latched level, and the special timing requirement of some instructions. Another field, $[m]$, shown in Equation 1, is added to record the latch level of an instruction. At a given execution level all the instructions of that level are either latched or un-latched. All the operations between two latched levels are synthesized as one combinational circuit. Every latched level corresponds to one pipeline stage, and has a delay of one cycle. A parameterized controller is generated to clock the pipeline.

ROCCC generates one VHDL component for each CFG node that goes to hardware. In a node, every virtual register is single assigned and is converted into wires in hardware. Arithmetic, logic and copying instructions become combinational or sequential VHDL statements according to whether they are latched or not. A *LUT* instruction invokes an instantiation of a lookup table component. If the lookup table is a pre-existing one, such as trigonometric or logarithmic function, the compiler automatically inserts the relevant values. Otherwise, the user provides the table entries, for example to describe a probability distribution function. In this case the compiler instantiates the lookup table as a regular *ROM* IP core unit in the VHDL code.

By adding more data types in Machine-SUIF, ROCCC supports any signed and unsigned integer and fixed-point type and size. The compiler infers the inner signals' bit size automatically from the arithmetic operations.

3.3 Comparison with Xilinx IP cores

Two previous works have compared compiler generated to hand-written VHDL codes for SA-C [8] and StreamsC [5]. In both cases it was shown, independently and on different examples, that the hand-written VHDL achieved a clock frequency half as large as the compiler generated codes. Achieving a comparable clock rate is one of the objectives of ROCCC. We therefore compare the hardware performance generated from Xilinx IP cores and ROCCC-generated VHDL code. We use Xilinx ISE 5.1i and IP core 5.1i. All the Xilinx IP cores and ROCCC-generated VHDL code are synthesized targeting a Xilinx Virtex-II xc2v2000-5 FPGA. All the benchmarks in Table 1 are from Xilinx IP core, except the DWT engine that we wrote. The input and output variables of ROCCC equivalents have the same bit sizes as that of the IP cores.

Bit_correlator counts the number of bits of an 8-bit input data that are the same as of a constant mask. *Mul_acc* is a multiplier-accumulator, whose input variables are a pair of 12-bit data. *Udiv* is an 8-bit unsigned divider. *Square_root* calculates a 24-bit data's square root. *Cos*'s input is 10-bit, its output is 16-bit. The arbitrary *LUT*, whose content can be defined by users in a text file before synthesis, has the same port size as that of *cos*. *FIR* is two 5-tap 8-bit constant coefficient finite impulse response filters, whose bus sizes are 16-bit. *DCT* is a one-dimensional 8-data discrete cosine transform. The input data size and output data size are 8-bit and 19-bit, respectively. For Xilinx IP *FIR* and *DCT*, multiplications with constants are implemented using the distributed arithmetic technique, which performs multiplication with lookup-table based schemes. Therefore, we set the

Table 1 - A comparison of hardware performance from Xilinx IPs and ROCCC-generated VHDL code. (*DWT code is handwritten.)

Example	Xilinx IP			ROCCC			%Clock	%Area
	Clock (MHz)	Delay (cycl)	Area (slice)	Clock (MHz)	Delay (cycl)	Area (slice)		
bit_correlator	212	1	9	144	2	19	0.679	2.11
mul_acc	238	1	18	238	1	59	1.00	3.28
udiv	216	11	144	272	25	495	1.26	3.44
square root	167	25	585	220	37	1199	1.32	2.05
cos	170	1	150	170	1	150	1.00	1.00
Arbitrary LUT	170	1	549	170	1	549	1.00	1.00
FIR	185	17	270	194	1	293	1.05	1.09
DCT	181	20	412	133	2	724	0.735	1.76
DWT*	104	1	1464	101	3	2415	0.971	1.65
Average:							1.001	1.93

synthesis option ‘*multiplier style*’ as ‘*LUT*’ for the ROCCC-generated *DCT* and *FIR*. The second through the fourth column of Table 1 show Xilinx IP cores’ clock rate, delay in clock cycle, and device utilization, respectively. The fifth through the seventh column show ROCCC’s corresponding performance. %Clock is the percentage difference in clock rate of ROCCC-generated VHDL compared to Xilinx IP. %Area is the percentage difference in area of ROCCC-generated VHDL compared to Xilinx IP.

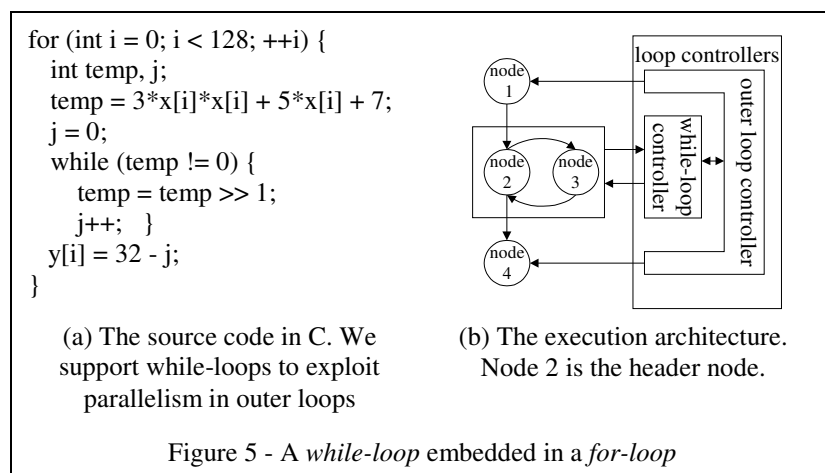
Bit-correlator, *udiv* and *square root* consist of a large number of bit manipulation operations, which for the C language is not well suited to express. This is the major source of the performance difference. Xilinx *mul_acc* IP has a control input signal *nd* (*new data*) whose Boolean value *true* indicates the present data is valid. In C code, we describe the equivalent behavior using an *if-else* statement whose condition evaluates the Boolean input *nd*, requiring extra nodes and latches to be added to support the alternative branch, consuming extra area. (We also tried changing this C code simply by multiplying *nd* with the new input data instead of using the *if-else* statement. Though one more multiplier was used, the overall area and clock rate performance was better than that listed in Table 1. Obviously, that is not a compile level optimization, but it does show one of the advantages of high-level synthesis: ease of algorithm level optimizations). In terms of lookup tables, the ROCCC-generated VHDL code instantiates Xilinx IP cores, so they have exactly the same performance. In Xilinx Virtex-II, 10-bit-input-16-bit-output *cos/sin* lookup table stores only half wave, which is one of the reasons that this *cos/sin* lookup table utilizes less area compared with the arbitrary ROM lookup table with the same port size. *Fir* operates on an array: basically, a 5-data element window slides over the one-dimensional array, and ROCCC generates a smart buffer to reuse the previous

input data. The *FIR*'s data-path consists of multipliers and adders with no branch. ROCCC fits this type of algorithm and gets comparable performance with IP cores. The IP core has several handshaking signals. The ROCCC-generated *FIR* does not have those handshaking signals since its data communication method with outside is known at compile time. Like *FIR*, *DCT* has high computational density and no branch. The throughput of Xilinx DCT IP is one output data element per clock cycle, while ROCCC's throughput is eight output data elements per clock cycle. Therefore, though ROCCC-generated *DCT* runs at a lower speed (73.5%), the overall throughput of the ROCCC-generated circuit is higher. Both ROCCC *DCT* and Xilinx IP *DCT* exploit the symmetry within the cosine coefficients. The last row in Table 1 shows an implementation of a two-dimensional (5, 3) wavelet transform engine, which is the standard lossless JPEG2000 compression transform. This DWT engine includes the address generator, smart buffer and data-path. The ROCCC-generated circuit is compared with a handwritten one.

We derive the bit-width information based only on port size and opcodes. More aggressive bit-width narrowing transformations, performed by users and/or the compiler, may further reduce device area utilization.

3.4 The Data-path Generation and the Control of a While-loop

A while-loop is an inherently sequential construct that is not usually considered a candidate for mapping to hardware. However, often signal and image processing algorithms have a while-loop nested in a parallel for-loop or vice versa. The internal structure of a von Neumann processor is tailored for the execution of sequential codes, and can therefore easily support the execution of a while-loop. A spatial implementation,



however, lacks a program counter, so the compiler must generate a simple yet efficient customized control structure for each while-loop. Figure 5 (a) shows an example in C, in which the main computational burden is in the multiplication that squares $x[i]$. In order to put the whole for-loop in hardware, we need to also support the inner while-loop in hardware. While-loops usually cannot be unrolled and their implementation must support feedback of variables between iterations.

In Figure 5, nodes 2 and 3 correspond to the while-loop: at the bottom of node 2, there is a branch instruction to assert whether the loop body, node 3, should be executed or not. Node 2 is called the header node of a while-loop. The loop controller consists of two sub-controllers, the while-loop controller, and the outer loop controller.

The controller first activates the while-loop's predecessor node(s) - for example, node 1 in Figure 5 (b). In the predecessor nodes, multiple loop iterations are instantiated since each level corresponds to one iteration and we have assumed no loop dependency. Once the earliest iteration reaches the bottom of node 1, the controller halts node 1 and activates node 2. The while-loop controller here does not activate the whole data-path of node 2: rather, it activates the while-loop data-path from top to bottom, stage by stage. The while-loop branch instruction, which is at the bottom level of node 2, generates the assertion *loop_again* signal to the while-loop controller. After the execution of the branch instruction in the header node, the controller evaluates the *loop_again* signal from the data-path to determine whether to enter the loop body or the successor node following it. If the *loop_again* is set, the controller signals the while-loop body to execute, after which, the controller rewinds to execute the header node again. If, instead, the *loop_again* signal is clear, the controller halts the while-loop body and activates the while-loop's successor node as it did to the predecessor nodes. The loop controllers are written in synthesizable VHDL with heavy use of generics. These generics describe the length of the while-loop data-path and that of the outer for-loop, the while-loop's location relative to the outer loop, and the location of the branch instruction of the while-loop.

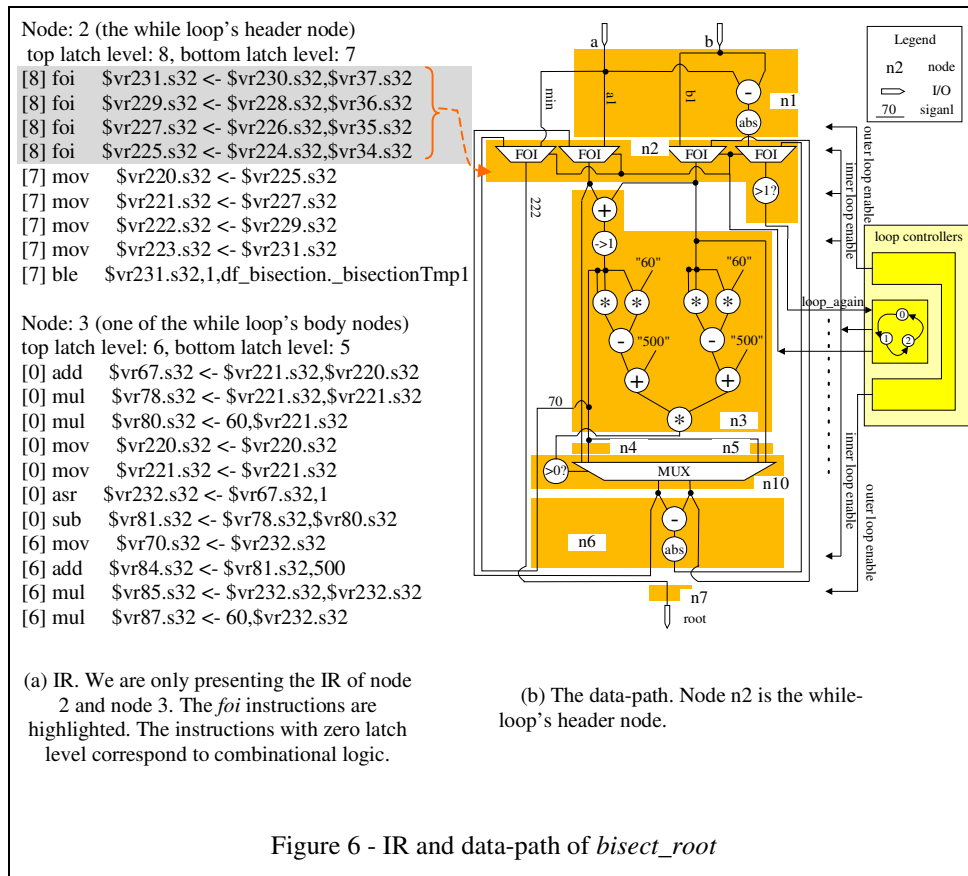
The most significant difference between a while-loop data-path and the rest of the outer for-loop data-path is that the former has to deal with feedback variables from the while-loop body nodes to the header node. In order to simplify the controller, ROCCC forces all feedback variables to be assigned in the bottom while-loop body node. Otherwise, the variable is copied down to the bottom node. We create a new instruction in the Machine-SUIF virtual machine instruction set, FOI (feedback or initialization). The instruction format of FOI is as following.

$$foi \$vr1.s32 <- \$vr2.s32, \$vr3.s32 \quad (2)$$

This instruction only appears on the top of a while-loop's header node. When executed, it evaluates signal FOI_i , which comes from the while-loop controller. If FOI_i is clear, the destination operand equals the first source operand; otherwise, the destination operand equals the second source operand. The hardware implementation of instruction FOI is a multiplexer. The first time a header node is executed, the controller signals the FOI instructions (multiplexers in hardware) to select the upper node's output variables. From then on, the controller signals the FOI instructions to select the feedback variables. The FOI instruction is also used to copy constant variables from a while-loop's predecessor node to the while-loop body.

Table 2 - Description and source code of the *while* loop examples

	Description	Source code
GCD_ifelse	The segment calculates the greatest common divisor using if-else statement.	<pre>a = x; b = y; while (a != b) { if (a > b) a = a - b; else b = b - a; } gcd = a;</pre>
GCD_minmax	The segment calculates the greatest common divisor using min() and max() macros.	<pre>a = x; b = y; min = x; max = y; while (a != b) { min = ROCCC_min(a, b); max = ROCCC_max(a, b); a = min; b = max - min; } gcd = min ;</pre>
Dif_equi	The algorithm numerically obtains y in equation $\frac{d^2y}{dx^2} + 3x\frac{dy}{dx} + 3y = 0$ where x starts from x_{in} to a with step dx.	<pre>u = u_in; x = x_in; y = y_in; u1 = u; x1 = x; y1 = y; while (x < a) { x1 = x + dx; u1 = u - 3*dx*(u*x + y); y1 = y + u *dx; x = x1; y = y1; u = u1; } x_out = x1; y_out = y1; u_out = u1;</pre>
Integ_equi	The algorithm numerically solves equation $\int_a^y \cos^2 x dx = const$	<pre>sum = 0; x = a; while (sum < const) { temp = cos(x); sum = sum + temp*temp; x = x + 1; } y = x;</pre>
Bisect-root	Using bisection method, the algorithm finds the root of equation $x^2 - 60x + 500 = 0$ in the range of [a, b].	<pre>a1 = a; b1 = b; mid = a1; while (ROCCC_abs(b1-a1) > 1) { mid = (b1 + a1) >> 1; right_root=(b1*b1)-(60*b1)+500; mid_root=(mid*mid)-(60*mid)+500; if((right_root*mid_root)>0) b1 = mid; else a1 = mid; } root = mid;</pre>



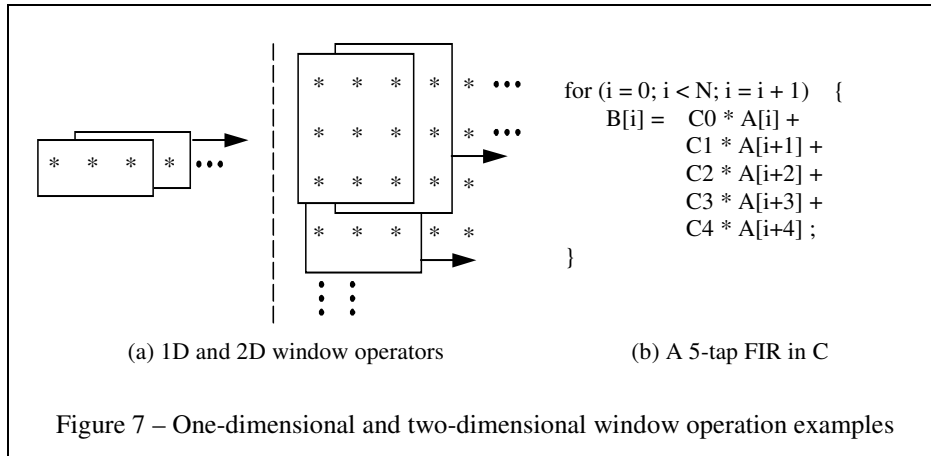
3.5 While-loop examples

Table 2 lists five examples having a while-loop inside a for-loop. Figure 6 shows the IR and data-path of the Bisect.root example.

Table 3 shows the place-and-route results and execution results from simulation. The bit size of *Integ_equi* is 10-bit and all the other examples' signals are 32-bit. The last column is the number of cycles per iteration. In *GCD_ifelse* the *if-else* statement is converted by ROCCC: both branches execute and the predicate selects the outcome.

Table 3 – The hardware performance of the *while-loop* examples

	Area (slices)	Multpl (18x18)	Clock (MHz)	cycle per iteration
GCD_ifelse	275	0	69.9	6
GCD_minmax	191	0	76.7	3
Dif_equi	370	12	40.9	5
Integ_equi	140	1	56.5	4
Bisect_root	590	15	51.1	8

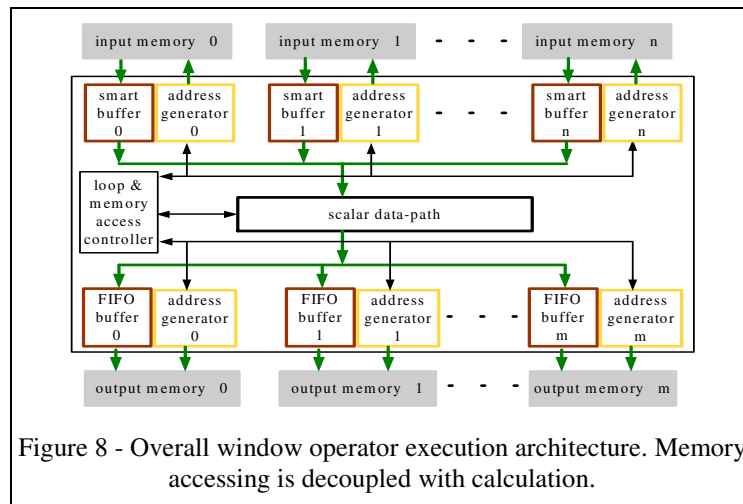


GCD_minmax, on the other hand, utilizes predefined macros to pick *min* and *max* values. These macros, instantiated as RTL VHDL function calls in the generated VHDL code, are more efficient. *GCD_minmax*'s generated data-path is shorter and takes fewer clock cycles compared with *GCD_ifelse* with no pipeline idle cycles.

4. INPUT DATA REUSE

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, edge detectors, erosion/dilation operators and texture measures in image/video processing. 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/sample and its neighbors), while the window slides over an array, as shown in Figure 7 (a). In most cases, these window operators are do-all for-loops and their data-paths can be generated using the approach presented in the previous section.

However, in order to take advantage of the high-throughput of the data-path, input data has to be organized and fed in efficiently. Figure 7(b) shows a five-tap FIR filter example code in C. $B[i]$ is the filter's output and $A[i]$ is the input. C_0 , C_1 , C_2 , C_3 and C_4 are the filter's constant coefficients. In the previous section we already showed that the throughput of the compiler-generated data-path is one-iteration per clock cycle. However, if a 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.

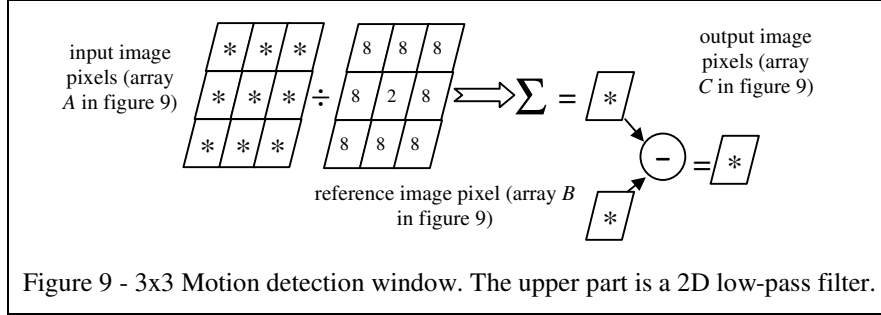


Balancing computation with I/O has been considered a critical factor of the overall performance for quite some time [17]. When a high-density computation is performed on a large amount of input data, as the case in window operations, data I/O often dominates the overall computation performance. For instance, for the window operations reported in [7], the general-purpose CPU performed 64X to 112X more load operations per pixel than a hand-crafted circuit on an FPGA. Therefore, in order to achieve high performance, a reconfigurable computing compiler needs to generate *smart* hardware in HDL to reduce the memory bandwidth pressure by exploiting data reuse whenever possible. We call this piece of synthesizable HDL module *smart buffer*. The compiler must implement the smart buffer tailored for the applications, and schedule the buffer's reads and writes.

4.2 Execution Architecture and Code Analysis

In window-based operations, the input and output arrays (or streams) are separate and therefore there is no loop-carried dependency on a single array.

An execution architecture of window operators is shown in Figure 8. The ROCCC framework is not board-specific in that it does not assume a pre-set number of memory modules connected to the FPGA. The data communication engine between the on-chip block RAMs and the inter-chip data streams is not part of the ROCCC code generation. The loop has n input arrays and m output arrays. If n is equal to one, the compiler generates a single-mode smart buffer. Otherwise, it generates multi-mode smart buffers. Each input/output array of a loop corresponds to an input/output memory. 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. Every



input memory or output memory is connected to a compiler-generated smart buffer, or a compiler-generated FIFO (first-in-first-out) buffer, respectively. The compiler performs scalar replacement on the loop body in the front-end. Therefore in Figure 8 the data-path, by itself, does not access memory directly. The address generators are VHDL libraries. 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 streams from the block memory, exploits data reuse and makes the data of the *current window* available to the data-path. In other words, the smart buffer collects/reuses the input data of one iteration and exports them into the data-path at the same time. In this way, the computation and memory access are decoupled. 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.

A one-dimensional example algorithm and a two-dimensional example algorithm are shown in Figure 7(b) and Figure 9, respectively. We use them to explain how the ROCCC compiler generates smart buffer. For the algorithm in Figure 9, the output pixel is the difference between the output of the low-pass filter (upper part in Figure 9) and the

```

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][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] + 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+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] + 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];
  }
}

```

Figure 10 - Motion detection C code
2 x 2 unrolled loop for the algorithm in Figure 9. Array A and B correspond to input memory banks, and array C correspond to output memory bank.

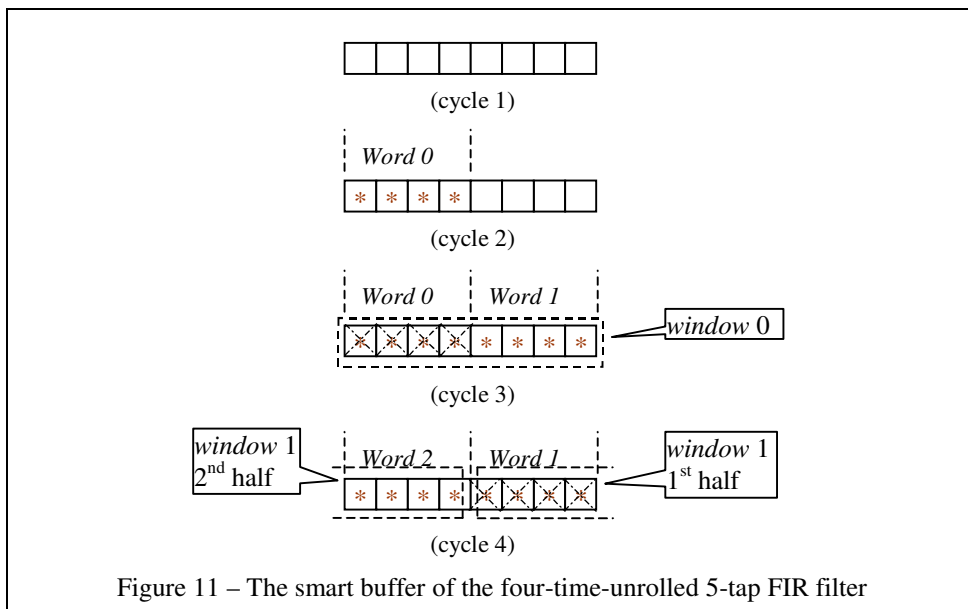
pixel of the reference image. Figure 10 gives the unrolled C code before undergoing scalar replacement. The compiler walks through all the memory references in the SUIF IR and confirms that there is no array being both read and written. The compiler also checks the following constraints.

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.

4.3 VHDL Code Generation

Figure 11 shows the registers of the FIR filter [Figure 7 (b)]’s one-dimensional smart buffer. In this figure, we assume that the memory I/O bus is 32-bit and the data width is 8-bit. In order to fully utilize memory bandwidth, we unroll the for-loop four times. The unrolled for-loop body has four copies of the original loop body. Now each iteration reads eight contiguous array elements, but the stride between two iterations is four ($i = i + 4$). The smart buffer has eight elements. In Figure 11, subfigures (1) through (3) show the smart buffer’s status from clock cycle one through clock cycle four. At cycle two, the smart buffer gets the first four data. At cycle three, the smart buffer collects the eight data elements needed for the current iteration and exports them to the data-path initiatively. Notice that also at cycle three, the left-most four elements are killed to reclaim space for *word 2*. At cycle four, *window 1* starts from the right-most four elements in the buffer.

The ROCCC compiler generates both the registers and the logic to schedule the registers’



action.

In this subsection we present ROCCC's approach to the generation of efficient VHDL code for the smart buffers and related components. The goal is to minimize run-time control calculation and maximize input data reuse.

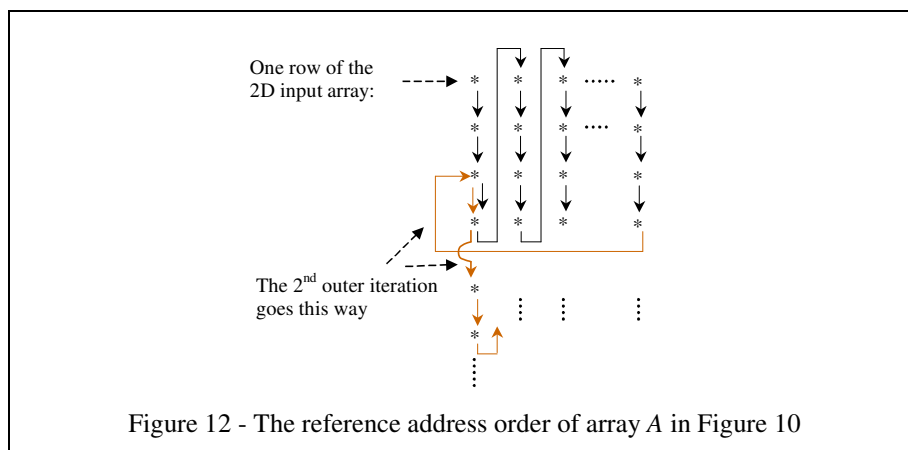
4.3.1 Address Generation

Window operations have one or more windows sliding over one or more arrays. Both the read and the write array addresses are known at compile time. We also assume that the on-chip memory read latency is known².

Consider the code in Figure 10 as an example. According to the memory load references and the loop unrolling parameters, the following parameters are known at compile time:

1. Starting and ending addresses
2. The number of clock cycles between two consecutive memory accesses
3. The unrolled window's size
4. The unrolled window's strides in each dimension
5. The array's row size
6. The starting address-difference between two adjacent outer-loop iterations.

In Figure 8, each smart buffer has an address generator to generate the loading address stream to the input memory bank. These address generators are parameterized finite state machines (FSMs) in VHDL. Figure 12 shows the loading address stream of the two-



² This is not an unusual assumption in FPGA design.

dimensional array A in Figure 10. An address generator does not produce all the loading addresses of the corresponding array for the current iteration; rather, it only produces the new data's addresses. The remaining recently loaded data are already in the smart buffer. In Figure 12, the input data needed by each outer-loop iteration are loaded only once. But there are overlaps between adjacent outer iterations since we cannot afford a smart buffer to hold whole rows of data. Notice that the first inner-loop iteration of each outer-loop iteration might read more columns of data compared with other inner-loop iterations because for the first iteration all the loads are new. We call these extra new columns warm-up columns.

In the case of the data-path producing output data at a higher rate than the loading rate from the input memory, and the output memory bandwidth is limited, the loading address generator has to slow down its address stream to balance the I/O throughput by inserting extra idle cycles after each iteration's loading. All these analyses can be done at compile-time and thereby simplify the controller.

The VHDL generics of an address generator include:

- The number of *warmup_columns* (the number of *warmup_data* for one-dimensional address generators.).
- The number of new columns per iteration (the number of new data per iteration for one-dimensional address generators.).
- The number of cycles to halt between iterations. This parameter might be zero. These halt cycles are inserted to balance the I/O throughput between input memories and output memories.
- The number of loads per outer-loop, the number of data per row, and the address difference between two adjacent outer-loop iterations' upper-left-most loads. These parameters only apply to two-dimensional address generators.

With the above parameters, each input memory's address generator is able to generate the address stream of the current iteration's new data, and each output memory's address generator can generate the corresponding storing addresses for the current iteration's output data. Once the generation of the current iteration's address stream is done, all address generators check if the loop and memory access controller has issued the next iteration, and act accordingly. For two-dimensional-array cases, the above parameters also ensure that the loading address generators are able to determine when a new outer-loop iteration starts, and wait until then to flush the smart buffers. Notice that all loading address generators have to flush the smart buffers though the *loop and memory access*

controller. The latter flushes all the smart buffers at the same cycle to maintain the synchronicity of the data-path's input data.

4.3.2 Smart Buffer Generation

An input array's smart buffer accomplishes the following tasks:

- Collecting the new input data for the current iteration.
- If the new input data are needed by future iterations, storing them to buffer registers whose contents are expired.
- Exporting the window data to the data-path when they are all ready. For multiple-input-array loops, the multi-mode smart buffers hold the window data until the controller signals it to release. This way all smart buffers are synchronized. A single-input-array loop's smart buffer pushes the window data into data-path immediately.

A smart buffer is implemented in VHDL at compile-time. It consists of registers to store the data and a finite state machine (FSM). The FSM traces which register is expired and would be overwritten by new data, determines when a window of data is ready to export, and manages the counterpart relationship between the buffer registers and the data-path's input variables.

We use the two-dimensional array A in Figure 10 as an example. Figure 13 shows the status of array A 's smart buffer at different clock cycles. Each iteration of the nested loop loads a 4x4 window. We assume the memory bus is twice the width of the pixel bit-size, so each memory load reads in two pixels. In this example the window's row size is an integer multiple of the bus-load (one row is two bus-loads). If this is not the case, for example each window row has five pixels, we round the smart buffer row size to an integer multiple of the bus-size, which makes the smart buffer row size six pixels (three bus-loads). Each FSM state is assigned one of the three states:

- *Prologue-state*: The computation is in a warm-up state of the first iteration of loops working on one-dimensional arrays or of the first inner-loop iteration of every outer-loop iteration of loops working on two-dimensional arrays. The smart buffers are collecting data to form the first window.
- *Export-state*: In this state, a window of data is ready. If the loop has only one input array, the smart buffer's FSM initiates the export of the window data to the data-path and keeps going to the next state. For multi-mode smart buffers, the FSM stays in this state until the loop controller signals it to release.

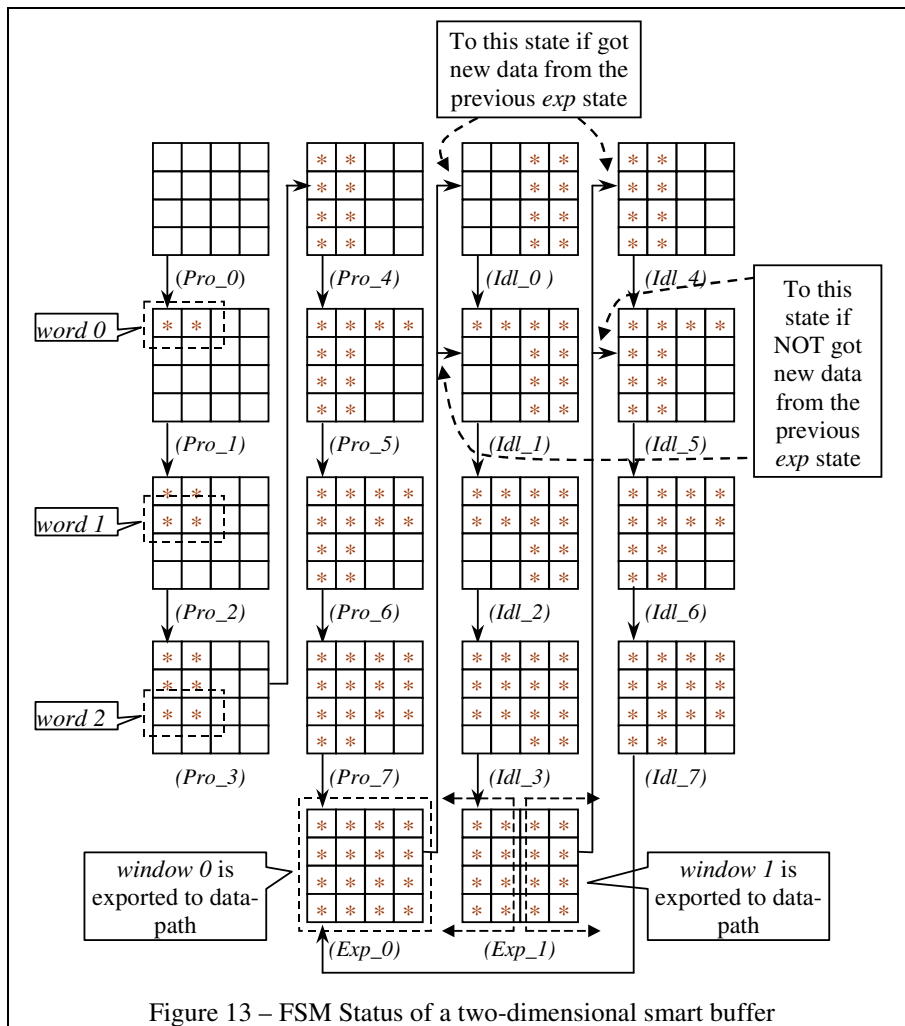


Figure 13 – FSM Status of a two-dimensional smart buffer

- **Idle-state:** In this state, the smart buffer is waiting for the new data that have not been loaded in previous iterations. Once all the new data have arrived, the smart buffer goes to the *export_state*.

Generally, for a two-dimensional smart buffer, the row size of the buffer is the smallest integer multiple of the bus-size that makes the row size larger than the sliding window's row size. The smart buffer's column size is equal to the height of the sliding window. For a one-dimensional smart buffer, the column size is always one.

As shown in Figure 13, the state machine initially starts from state Pro_0. Once it gets export-state Exp_0, window_0 is exported to the input port of the data-path. Since array A is one of the two input arrays in Figure 10, the FSM stays in this state until the loop controller signals it to go to the next iteration. If at Exp_0 state the smart buffer gets

another new bus of data, which would only happen in single-input-array cases, the state machine jumps to `Idle_1`. Otherwise it goes to `Idle_0`. Once the FSM leaves state `Exp_0`, the data elements in the left-most and the second-left-most columns are killed to reclaim the registers for new data. In state `Exp_1`, the smart buffer exports `window_1`. Notice that the first column of this window is the third column in the buffer. The compiler is aware of this relationship and implements it in the export states accordingly. The new data to state `Idle_7` switches the state machine back to `Exp_0`. In fact, the last columns of idle states (`Idle_4` through `Idle_7`) are identical to the last column of prologue states (`Pro_4` through `Pro_7`).

A two-dimensional array's smart buffer is able to be flushed and start the FSM from the first prologue state.

The generated VHDL code, by itself, does not have the concepts of *windows*. The VHDL code only describes the logical and sequential relationship between signals/registers.

Implementing smart buffers using shift registers, while is an option, would require too many counters inside the smart buffer. There also would have to be a great deal of logic between the counters to control the smart buffer's import and export actions. Besides, for the multiple-input-array case, the synchronization requirement increases the complexity of the logic. By using the FSM, the compiler's analysis effort simplifies the generated circuit, and the FSM tracks the smart buffer's status efficiently.

4.4 Experimental Results

We use the five benchmarks listed in Table 4 in our experiments. These benchmarks are selected for the diversity of their numbers of input arrays, dimension of arrays and memory interface bandwidth.

Constant_FIR is a constant-coefficient finite-impulse-response (FIR) filter. Its source code is given in Figure 3. Array *A* is the only input array, whose dimension is one. *Variable_FIR* is a variable-coefficient FIR filter. Its coefficients are also variables stored in Array *B*. This example has two one-dimensional input arrays. Each iteration of *Complex_FIR* produces a complex integer. The complex integer's real part and imaginary part are stored in Array *C* alternately. Therefore every iteration loads in two new data. The loop counter step of *Complex_FIR* is two. *2D_lowpass_filter* is a 3x3 low-pass filter used in image processing. For each 3x3 window in the input image, the nine pixels are divided by the corresponding coefficients (they really are shifting operations). The output

Table 4: The synthesis and simulation results of buffers in five examples. The total number of slices is 46592 on the target FPGA chip.

		constant FIR	variable FIR	complex FIR	2D_lowpass filter	motion detection
Input buffer A	Area (slices)	156	159	132	325	327
	# of regs	5	5	6	16	16
	# of states	14	14	8	18	18
	Bus size (bits)	8	8	16	16	16
Input buffer B	Area (slices)		159			150
	# of regs		5			4
	# of states		14			4
	Bus size (bits)		8			16
Output buffer C	Area (slices)	11	11	12	73	73
	# of regs	1	1	2	2	2
	# of states	1	1	2	2	2
	Bus size (bits)	8	8	8	16	16
Data-path	Area (slices)	43	5 mltpl	99	144	164
	Bit size	8	8	8	8	8
Overall area (slices)		210	329	243	542	714
Clock rate (MHz)		94	68	85	69	42
Execution time (cycles)		262	1019	260	5980	5986
Throughput (iteration/cycle)		0.96	0.25	0.48	0.16	0.16

pixel is the summation of the division’s nine results. We unroll the loop twice in both horizontal and vertical directions. Therefore, each iteration computes four of these 3x3 windows, and produces a 2x2 output window. The input of *2D_lowpass_filter* is a two-dimensional array. *Motion_detection* is the implementation of the code in Figure 10. *Motion_detection*’s input windows are 4x4 and 2x2. These two windows come from the image array to be detected and the reference image array, respectively. There is no data reuse to the reference image array and the smart buffer of it is used only to decouple the data-path with memory accessing. ROCCC supports multiple output arrays, though all these examples have only one output array. We do not show a multiple-output-array example because writing to each array is an independent process, and the implementation is a straightforward replication.

We use the Xilinx ISE 6.2.03i tool chain to do synthesis and place-and-route. The generated VHDL codes are simulated using ModelSim 5.8c. The target architecture of all synthesis is Xilinx XC2V8000-5, whose total number of slices is 46592. We set the synthesis option ‘*multiplier style*’ as ‘*LUT*’ for all constant multiplications.

In Table 4, *Area* is the number of slices obtained from place-and-route reports. The area of a buffer includes the area of the address generator associated with the buffer. *# of reg* is the number of registers that the corresponding buffer uses to store the data. Each register is 8-bit since all the data-path variables of these five examples are 8-bit. *# of state*

Table 5- Smart buffer size and synthesis results of unrolled loops

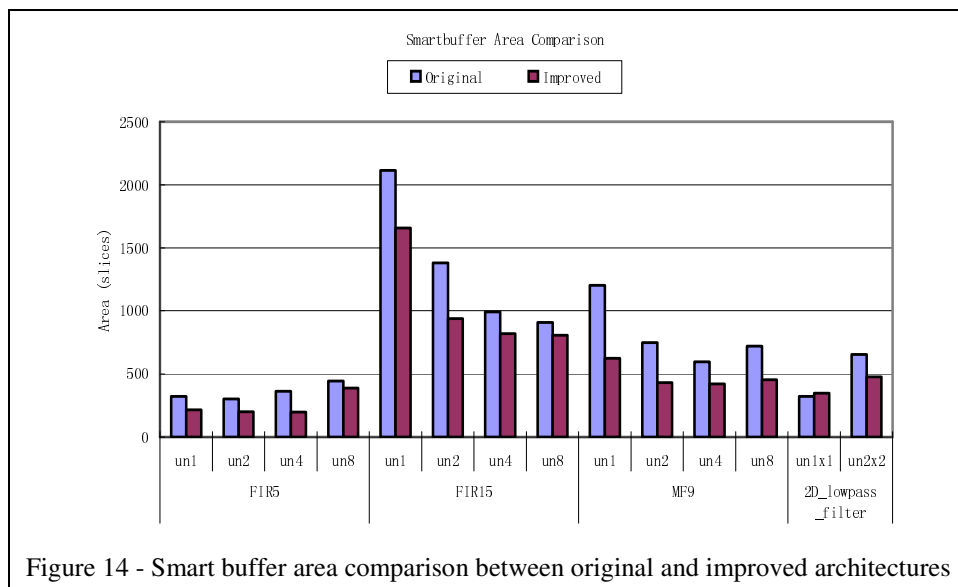
	Un1		Un2		Un4		Un8	
	buffer size(pixel)	slices	buffer size(pixel)	slices	buffer size(pixel)	slices	buffer size(pixel)	slices
constant_FIR5	5	213	6	199	8	197	16	388
constant_FIR15	15	1657	16	938	20	817	24	806
moving_Filter9	9	623	10	431	12	421	16	453
2D_lowpass_filter	9	242			16	376		

is the number of states of the buffer's FSM. *Bus size* is the number of bits of the bus between input/output memory and the corresponding buffer. Clock rate is the clock rate of the whole placed-and-routed circuit. The input data set size of all one-dimensional examples is 256 and the input data set size of all two-dimensional examples is 64x64. Execution time is the number of cycles obtained from the simulation waveforms.

Constant_FIR has only one input array, whose dimension is one. The input smart buffer automatically feeds its window data to the data-path whenever they are ready, without waiting for synchronization. The loop controller keeps issuing new iterations. The total execution time is just a few cycles more than the input data set size. The area cost on the input buffer's address generator of *constant_FIR* is more significant compared with that on the registers (only five 8-bit registers). *Variable_FIR* has two input arrays. Both the input smart buffers and the loop controller have to spend extra clock cycles to do handshaking with each other, so the execution time is longer compared with *Constant_FIR*. For *Complex_FIR*, we intentionally set the input bus-size to be 16-bit and the output bus-size to be 8-bit. Each memory load can read in two input data, which is what one iteration needs since the loop counter step is two. However, the compiler detects that the output FIFO buffer needs two cycles to store two output data of one iteration, the real part and the imaginary part. The compiler adds one halt cycle to the input smart buffer's address generator for each iteration, and this extra halt cycle explains the fact that *Complex_FIR*'s total execution time is almost the same as that of *Constant_FIR*, though the number of iterations of *Complex_FIR* is only half of that of *Constant_FIR*. *2D_lowpass_filter* has a two-dimensional array and the compile chooses the corresponding smart buffer and loop controller. In contrast, *Motion_detection* has two input arrays. Like the case of *Variable_FIR*, synchronizing between smart buffers and loop controller needs extra clock cycles. But these cycles overlap with the intrinsic delay of two-dimensional address generators. This is the reason that *Motion_detection*'s execution time is close to that of *2D_lowpass_filter*.

The window size of *2D_lowpass_filter* is 4x4. Every time the sliding window reaches the right of the image, it rewinds back to the left and starts from the second last row of the previous outer loop, since the outer loop counter's stride is now four. Therefore, by using the smart buffer, we only re-read each pixel once. Without the smart buffer each pixel would be read four times. Obviously, the more a loop is unrolled, the more memory loads can be saved. Four loop unrolling examples are shown in Table 5. *Constant_FIR15* is a 15-tap constant-coefficient finite-impulse-response filter. *Moving_filter9* is a nine-element moving average filter. We assume the data-size is 16-bit. *Buffer size* is the number of the 16-bit storage units in the smart buffer. The benchmarks are unrolled 2, 4 and 8 times, with the exception of *2D_lowpass_filter*, which is unrolled four times (2x2). We also assume that the bus bandwidth scales up with unrolling. Notice that the *buffer size* of *constant_FIR5* on Un8 is 16, though the number of memory loads per iteration is 12. The compiler rounds the buffer size to an integer multiple of the bus-size, which is 8-data per bus for Un8. When the loop is unrolled the buffer size increases, but it holds fewer distinct windows and therefore the cost of control logic decreases while the storage area increases. This explains why the total area of the smart buffer does not increase linearly with the amount of unrolling.

The smart buffer's control logic determines the location of the input data's destination buffer registers, schedules the output data export, and determines the proper connection between the buffer registers and their corresponding data-path input ports. Notice that this connection varies for different iterations as the window slides. For example, the



connection is different between Exp_0 and Exp_1 in Figure 13. In our original work [9], all these control is fulfilled by combinational circuit. The compiler made less analysis effort but generated a less efficient circuit. Compared with our previous work, the current smart buffer's FSM is more area efficient. The new approach requires more sophisticated compile-time analysis to build the FSM. The bar chart in Figure 14 depicts the smart buffer area comparison. With the exception of 2D_lowpass_filter(un1x1), the area savings of the new smart buffers range from 13% to 93%, and are 43% on average.

5. RELATED WORK

Many projects, employing various approaches, have worked on translating high-level languages into hardware. SystemC [27] is designed to provide roughly the same expressive functionality of VHDL or Verilog and is suitable for designing software-hardware synchronized systems. Handle-C [11] is a low-level hardware/software construction language with C syntax, which supports behavioral descriptions and uses a CSP-style (Communicating Sequential Processes) communication model. Both SystemC and Handle-C are timed languages.

GARP [3]'s compiler is designed for the GARP reconfigurable architecture, and generates GARP configuration file instead of standard VHDL. GARP's memory interface consists of three configurable queues, the starting and ending addresses of which are configurable. The queues' reading actions can be stalled.

SA-C [19], Single Assignment C, is a single-assignment high-level synthesizable language. 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 uses pre-existing parameterized VHDL library routines to perform code generation in a way that requires a number of control signals between components, and thereby involves extra clock cycles and delay.

Streams-C [6] relies on the CSP model for communication between processes, both hardware and software, and can meet relatively high-density control requirements. One-dimensional input data reuse can be manually implemented in the source code.

SPARK [26] is another C to VHDL compiler, which takes a subset of C as input and outputs synthesizable VHDL. Its optimizations include code motion, variable renaming, etc. The transformations implemented in SPARK reduce the number of states in the controller FSM, and the number of cycles in the longest path.

GARP, Streams-C and SPARK do not support accesses to two-dimensional arrays, so image processing applications, including video processing, must be mapped manually.

Phoenix project [28] has implemented a compiler called CASH, which represents the input program using Pegasus, a dataflow intermediate representation. CASH targets its asynchronous hardware. SOMA [29] is a synthesis framework for constructing Memory Access Network architecture, and has been integrated into CASH. SOMA features the support to input specifications in which memory references cannot be statically disambiguated.

DEFACTO [30] system takes C as input and generates VHDL code. In its generated circuit, memory units are connected to the data-path through the memory channel to reuse input data. The memory channel architecture has its FIFO queue and a memory-scheduling controller.

One main difference between ROCCC and these research projects is that ROCCC performs aggressive input data reuse.

6. CONCLUSION

In this paper we have presented the code generation part of ROCCC, an open framework built on the SUIF platform that compiles C programs to VHDL for mapping reconfigurable devices. At the front-end, the compiler performs high-level data flow analysis as well as an extensive set of loop transformations. It transfers the analysis information through preserved macros. At the back-end, the compiler explores low-level parallelism, pipelines the data-path and narrows the bit sizes of the inner signals.

When compiling parallel loops, the ROCCC compiler generates a pipelined data-path in which each pipeline stage corresponds to one iteration, so that the throughput is one iteration per clock cycle. We present our approach to maintaining the same throughput when there is only scalar recurrence between iterations. ROCCC supports lookup tables through automatically instantiating pre-existing lookup table IPs or ROM IPs. The synthesis result shows that ROCCC-generated circuits take around $2x \sim 3x$ the area but run at a clock rate comparable to that of existing Xilinx IPs. In many cases the throughput of ROCCC generated code is higher than that of the original IP. As expected, ROCCC performs better on high computational density examples than on high control density ones. When compiling sequential (*while*) loops, the compiler pipelines the data-path in a similar way, but only one iteration is executed at a time. We created a new instruction to

