An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware

Robert Rinker, Margaret Carter, Amitkumar Patel, Monica Chawathe, Charlie Ross, Jeffrey Hammes, Walid A. Najjar, Wim Böhm

Abstract— **We describe a system, developed as part of the Cameron project, which compiles programs written in a single-assignment subset of C called SA-C into dataflow graphs, and then into VHDL. The primary application domain is image processing. The system consists of an optimizing compiler which produces dataflow graphs, and a dataflow graph to VHDL translator. The method used for the translation is described here, along with some results on an application. The objective is not to produce yet another design entry tool, but rather to shift the programming paradigm from HDLs to an algorithmic level, thereby extending the realm of hardware design to the application programmer.**

Keywords— **Adaptive-computing, Configurable, Image-processing, Reconfigurable-components, Reconfigurable-computing, Reconfigurable-systems**

I. Introduction

IMAGE processing (IP) applications are ideally suited for reconfigurable computing. They exhibit a large degree of fine and coarse grain parallelism: at the bit (or pixel), instruction, loop and task levels. Moreover, they require the repeated application of the same operation on successive sets of data (e.g. streaming video). Reconfigurable computing systems (RCS's) are therefore interesting candidates for special purpose IP acceleration hardware: they provide a large degree of fine-grained parallelism that can be configured to efficiently fit many simultaneous small-data-size (pixel) operations.

However, most reconfigurable computing systems are based on FPGAs, and therefore are programmed using hardware description languages where the user specifies the logical structure of the intended circuit. This programming paradigm is very different from the algorithmic programming languages that are typically used by IP application developers. Another difficulty is partitioning of the algorithm between a host processor and reconfigurable modules, and devising ways of producing efficient FPGA configurations – both of these steps require intimate knowledge of the hardware and host interface, which is not something that the typical IP programmer understands.

The goal of the Cameron project [1] is to shift the programming paradigm for reconfigurable computers from hardware-centered to software-centered, thereby making them accessible to IP application developers and portable across reconfigurable computing platforms. This is achieved by creating a software infrastructure that translates a high level algorithmic language into a hardware description language. It consists of a graphical programming environment, a high-level language, an optimizing compiler, and debugging and performance monitoring tools for IP on reconfigurable computers. The objective of this system is to integrate the parts into a single design environment, and to allow the design to be carried out entirely by the application programmer, without requiring intimate knowledge of hardware or interface details.

The Cameron project includes the design of a language which is particularly suited for translation into hardware, called SA-C (*Single Assignment C*, pronounced *sassy*). SA-C programs are compiled into a dataflow graph (DFG) format. The compiler applies extensive expression, loop and array optimizations. The objective of the optimizations is to minimize the hardware cost of the program on the FPGA as well as to maximize locality by reusing data and expressions. The DFG format is then compiled into VHDL which is mapped, using commercial tools, onto the reconfigurable hardware. The focus of this paper is on this DFG to hardware mapping.

The rest of this paper is organized as follows. The next section highlights other related reconfigurable computing, projects. Section III provides an overview of the SA-C language and DFGs, particularly those features which facilitate and impact the DFG–to–VHDL translation. Next, the development of an abstract architecture, which defines the target for the translation, is discussed. The actual translation process is described in Section V. Next, an example is described, along with some preliminary performance numbers. The paper concludes with a discussion of future work.

II. Related Work

Reconfigurable computing is an active area of research – both hardware and software projects, and combinations of both, are ongoing. Hardware projects fall into two categories – those that use off-the-shelf components (in particular, FPGAs), and those which use custom designs.

The Splash-2 [2] is an early (circa 1991) implementation of an RCS, built from 17 Xilinx [3] 4010 FPGAs, and connected to a Sun host as a co-processor. Several different types of applications have been implemented on the Splash-2, including searching[4], [5], pattern matching[6], convolution [7] and image processing [8].

A commercial system which is loosely patterned after the Splash-2, but which utilizes larger but fewer FPGA's, is the Annapolis Microsystems Wildforce$^{(TM)}$ board [9],

This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

Computer Science Department, Colorado State University, Ft. Collins, CO 80523-1873, E-mail: {rinkerr, carterm, amit, monica, rossc, hammes, najjar, bohm}@cs.colostate.edu

introduced in 1995 – this system was used in the implementation described in this paper, and is covered in some detail later.

Representing the current state of the art in FPGA-based RCS systems are the AMS WildStar[10] and the SLAAC project [11]. Both utilize Xilinx Virtex [12] FPGA's, which offer over an order of magnitude more programmable logic, and provide a several-fold improvement in clock speed, compared to the earlier chips.

Several projects are developing custom hardware. The Morphosis project [13] marries an on-board RISC processor with an array of reconfigurable cells (RC's). Each RC contains an ALU, shifter, and a small register file. The RAW Project [14] also consists of an array of computing cells, called *tiles*; it differs in that each tile is itself a complete processor, coupled with an intelligent network controller, and a section of FPGA-like configurable logic that is part of the processor data path, more like an on-chip network of workstations; there is no "host" processor. These designs represent a more coarse-grained, or *chunky* architecture [15], compared to FPGA-based logic cells; such architectures promise to be more manageable as complexity increases.

PipeRench [16] consists of a series of *stripes*, each of which is a pipeline stage – an input interconnection network, a lookup-table based PE, a results register, and an output network. During execution, a context loader places pipeline stages to be executed into the next available stripes – in most cases, the context switching can be completely hidden. the application appears to execute in an infinitely deep pipeline.

On the software front, several of the above hardware projects also involve software development. The RAW project includes a significant compiler effort [17] whose goal is to create a C compiler which treats the network of tiles as a single system, rather than as individual processor nodes as in conventional network programming. For PipeRench, a low-level language called DIL [18] has been developed for expressing an application as a series of pipeline stages, which can easily be mapped to stripes.

Several projects (including Cameron) focus on hardware-independent software for reconfigurable computing; the goal – still quite distant – is to make development of RCS applications as easy as for conventional processors, using commonly known languages or application environments. Several projects use C as a starting point for RCS development. Handel-C [19] both extends the C language to express important hardware functionality, such as bit-widths, explicit timing parameters, and parallelism, and limits the language to exclude C features that do not lend themselves to hardware translation, such as random pointers. Streams-C [20] does a similar thing, with particular emphasis on extensions to facilitate the expression of communication between parallel processes. SystemC [21] and Ocapi [22] provide C++ class libraries to add the functionality required of RCS programming to an existing language.

Finally, a couple of projects use higher-level application environments as input. The MATCH project [23], [24] uses MATLAB as its input language – applications that have already been written for MATLAB can be compiled and committed to hardware, eliminating the need for re-coding them in another language. Similarly, CHAMPION [25] is using Khoros [26] for its input – common glyphs have been written in VHDL, so GUI-based applications can be created in Khoros and mapped to hardware.

III. The SA-C Compiler and Dataflow Graphs

Rather than trying to extend or limit an existing language, SA-C[27] is designed specifically to make it easy for the compiler to analyze the code and extract both fine-grain and coarse-grain parallelism. SA-C is an expression–oriented, single assignment (functional) language that is designed to be translated into hardware descriptions. As the name implies, the syntax is loosely based on C; however, there are significant differences as well, mostly due to its use as hardware generation language.

A. Unique features of SA-C

- A flexible type system, including signed and unsigned integers of any bit width, as well as fixed point numbers.
- True multi-dimensional arrays, with a specific size and shape which may be inferred when the array is created.
- No pointers or other indirection, to eliminate side-effects,
- Loop generators, which are usually used in place of the more traditional "loop index used as an array subscript" to perform operations on arrays. Conceptually, there is no specified order to the operations performed on elements of the array, but rather it appears that the entire array is defined at one time. This gives the compiler the freedom to implement array operations in the most efficient way.
- Reduction operators, which perform commonly used operations on the data produced in loop bodies, such as `array sum` and `histogram`. This allows the programmer access to efficient VHDL implementations of these operations.

The language includes several features that make it especially suited for IP applications; however, it is a general purpose language that can be used for other applications as well.

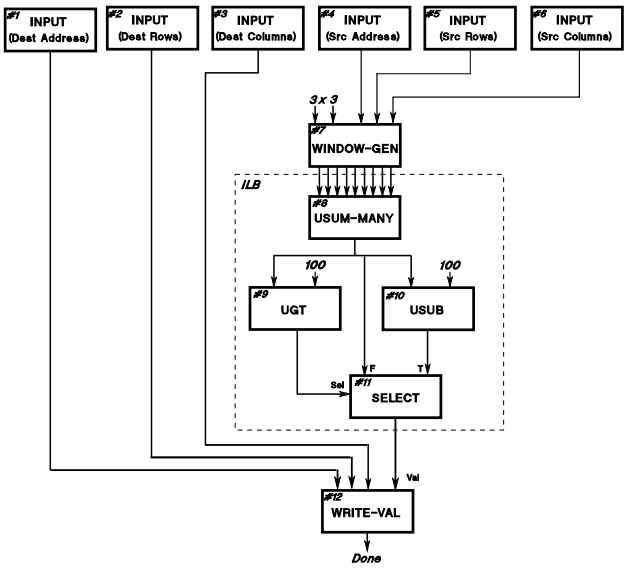
A simple SA-C program is shown in figure 1(a). This program accepts a 2-D array (named `Arr`) of 8-bit unsigned integers (i.e., of type `uint8`) as input. A window generator statement (`for window...`) extracts all 3×3 sub-arrays from the image array, and sums the elements in each sub-array. A new array (named `r`) is formed such that each element is either the sum of the corresponding window or, if the sum is greater than 100, the sum minus 100. This simple program demonstrates several characteristics of the SA-C language; a complete language reference is available in [27]. Upon compilation, an intermediate form of the program, called a dataflow graph (DFG), is generated; a pictorial representation of the DFG for this program is shown in figure 1(b).

B. SA-C Compiler Optimizations

Numerous optimizations are performed by the compiler before creating the DFG. Several traditional optimizations

```
uint8[:,:] main (uint8 Arr[:,:]) {
 uint8 r[:,:] =
   for window W[3,3] in Arr {
     uint8 s = array_sum (W);
     uint8 v = if(s>100) return(s-100)
               else      return(s);
    } return (array (v));
  } return (r);
```

(a)



(b)

Fig. 1. (a) A simple SA-C program, and (b) the resulting dataflow graph

minimize the calculation required by the hardware; these include code motion, constant folding, array and constant value propagation, and common subexpression elimination. Function inlining and loop unrolling provide parallel computation opportunities, and often enable other optimizations.

Other, less traditional optimizations are included to reduce hardware size and/or increase execution speed:

- Array and loop size propagation, facilitated by the use of the loop generator statements, allow the compiler to automatically determine loop unrolling depth.
- Bit width narrowing works in conjunction with loop unrolling to insure that each instance of a loop body uses the smallest data sizes possible to perform a given calculation.
- *Stripmining* - splits a loop into a pair of nested loops, with the outer loop creating chunks of work which are performed by the inner loop. When implemented in hardware, parallelism is introduced by separately instantiating each inner loop body; the outer loop is converted into code which distributes data to these instances.
- *Tiling* - allows a large image to be split into smaller pieces that fit into the RCS memory.
- *Loop fusion* - forms a single loop from two or more consecutive loops in an algorithm. This helps to eliminate extra data communication between processing steps. The compiler can often perform this operation in situations where such fusion is not obvious to the application programmer. Some of the optimizations may be detrimental to performance if applied in the wrong situation, or trade one hardware resource for another; in these cases, their application can be controlled by *pragmas*.

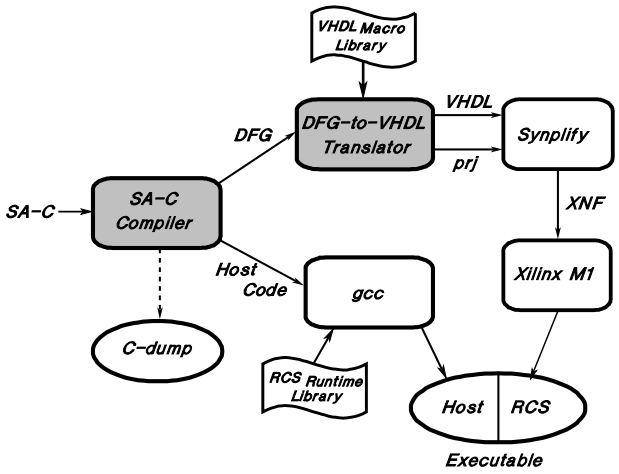


Fig. 2. Overview of the SA-C Compilation process

An overview of the current implementation of the compilation system is shown in figure 2. The compiler translates SA-C programs and performs optimizations as described above. It produces:

- A host-based C program, which controls the hardware, manages data transfer to/from the RCS, and performs execution tasks that cannot be performed by the hardware, such as file I/O.
- A DFG of that portion of the program which will execute on the RCS.
- Optionally, a *C-dump*, a C-version of the entire program which can be used for debugging and verification before committing the program to hardware.

By default, the compiler tries to move as much of the program as possible from host to hardware; in the vast majority of cases, its decision is reasonable. However, for those cases where it makes a bad decision, a *pragma* can be used to force the compiler to keep more of the program on the host.

The second component, a VHDL-to-DFG translator, extracts information from the generated DFG and produces a VHDL implementation of the program. This code is processed by VHDL synthesis and place–and–route tools to produce hardware configuration files for the RCS.

IV. An Abstract Architecture for Reconfigurable Computing

Unlike "standard" instruction set architectures, which provide a relatively small set of well-defined instructions to the user, RCS's are composed of an amorphous mass of logic cells which can be interconnected in any number of ways. To limit the number of possibilities available to the designer, an *abstract architecture* has been defined as a more manageable, hardware independent target. We currently define three types of functions:

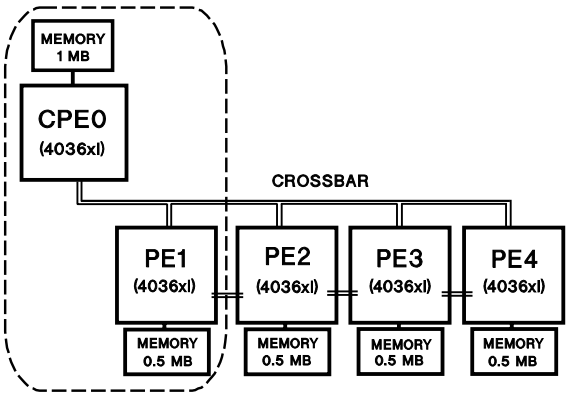


Fig. 3. Architecture of the Wildforce-XL Reconfigurable Computing Board, showing that part of the board being utilized by the DFG-to-VHDL translator

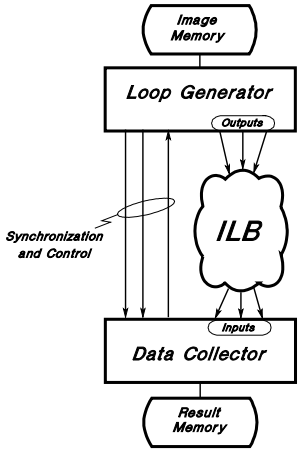


Fig. 4. Structure of the code generated by the DFG-to-VHDL translator

• Data transfer mechanisms. These include streams, block memory transfer, and systolic modes. These mechanisms are used both between host and RCS as well as within sections inside the RCS.
• Arithmetic and Logical Operations. These include common simple operators such as `ADD`, `SHIFT`, and `COMPare`, as well as more complex operations such as `SQRT`, `SUM`, and `MEDIAN`. The set of operations that have been included in SA-C was influenced by the IP application domain.
• Data buffering and storage mechanisms, including FIFOs and arrays of buffers, which are used to implement more complex functions such as shift registers.

A. Implementation on the AMS Wildforce Board

The first version of the compilation system targets the AMS Wildforce board [9]. Figure 3 shows a simplified diagram of the board used as a target. This board consists of 5 FPGAs, connected such that one FPGA, named CPE0 (*Control Processing Element 0*), can broadcast data to the others, PE1-4, via a 36 bit crossbar. Each PE also has access to its own local memory, organized as 32 bit words. The board is connected to a host via a PCI connection – the host is responsible for downloading the configuration codes, and for sending and retrieving data to/from the board.

Our initial implementation of the compilation system uses only that portion of the board shown inside the dotted lines in the figure. This implementation subset was chosen primarily for simplicity – CPE0 retrieves image data from its local memory and sends it in the proper order over the crossbar to PE1, which buffers it, performs the necessary calculations, and stores the results in its local memory. Since two PE's and thus two memories are used, the design is simplified because there is no memory contention between reading and writing. It is expected that the next generation of the translator will use more of the PE's on this board, either for added functionality or to enhance parallelism in the present system. On the other hand, this subset is reasonable in its own right, since the trend in new board designs is to use fewer, larger FPGAs; thus, this scheme seems to be a reasonable model for future work when the system is ported to other hardware.

The choice of PE1 in the implementation is arbitrary – any of the PE's 1–4 could have been used, since they are identical chips and have nearly identical connections. In the following discussion we use the term *PEx* to reference this PE.

V. Translating DFG's to VHDL

At first glance the DFG of figure 1(b) appears to describe a simple top to bottom execution that can be implemented by a combinational circuit; however, the operation of some of the nodes are more complicated. In particular, the `WINDOW-GEN` node retrieves data from RCS memory and presents it in the proper order to the inner loop body (ILB) of the program, and the `WRITE-VAL` node collects the results and stores them into memory to be retrieved later by the host. These operations require multiple clock cycles, several state machines, and coordination between the upper (loop generator) and lower (data collection) nodes. Figure 4 shows the resulting design partitioning.

A. Classification of DFG nodes

As a first step in converting the DFG to VHDL, the translator classifies the dataflow graph nodes into one of four types:
• Run-time input nodes. These nodes are not directly translated into VHDL, but rather specify addresses within CPE0's memory where run-time data has been stored. It includes information only known at run-time, such as the size, shape, and starting address of the data arrays. The translator uses this information to form a table of addresses which will be used by the `ConstGrabber` module to retrieve the data from memory at the start of execution.
• Generator nodes. The information in these nodes includes such things as window size, shape, and step size. In contrast to the run-time constants discussed above, this information is used during compilation to parameterize the instantiation of the various VHDL components.
• Loop body nodes. These nodes specify the operations to be performed by the ILB. These nodes are used to generate the VHDL code for the ILB.

```
entity Computation is                                          -- 1
  port(NODE7OUT : in ByteArr(WinSize-1 downto 0);              -- 2
       NODE12OUT: out Byte);                                   -- 3
end;                                                           -- 4
architecture byblocks of Computation is                        -- 5
  signal USUM_MANY8OUT0:std_logic_vector(7 downto 0);--  6
  signal UGT9OUT0: std_logic;                                  -- 7
  signal USUB10OUT0: std_logic_vector(7 downto 0);             -- 8
  signal SELECTOR11OUT0:std_logic_vector(7 downto 0);-- 9
begin                                                          --10
  NODE9: USUMMANY generic map( nVals => 9,                     --11
                               insize => 8,                    --12
                               outsize=> 8)                    --13
                  port map( vals => NODE7OUT,                  --14
                            result => USUM_MANY8OUT0);--15
  UGT9OUT0 <= '1' when USUM_MANY8OUT0 > 100                    --16
                  else '0';                                    --17
  USUB10OUT0 <= USUM_MANY8OUT0 - 100;                          --18
  SELECTOR11OUT0 <= USUB10OUT0 when UGT9OUT0 = '1'             --19
                    else USUM_MANY8OUT0;                       --20
  NODE12OUT<= SXT(SELECTOR11OUT0, 8);                          --21
end;                                                           --22
```

Fig. 5. The generated VHDL for the ILB of figure 1(b) (some declarations and type casting have been omitted for clarity).

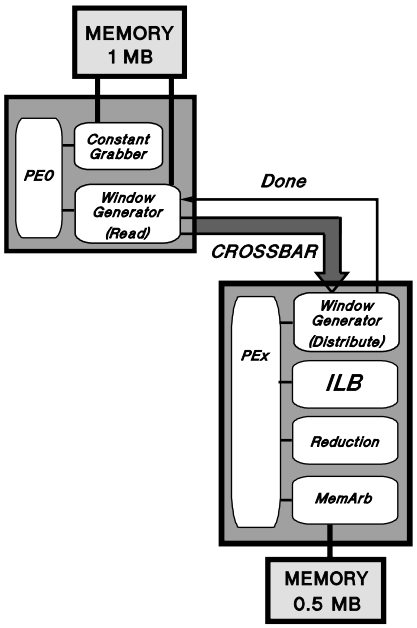


Fig. 6. Location of components in the 2 PE model

• Reduction nodes. Similar to the generator nodes, these specify the parameters needed to select and instantiate the reduction (collection) nodes.

Given this information, the translation process is divided into three main parts. First, the ILB is identified as being that part of the DFG that lies between the *outputs* of the loop generator nodes and the *inputs* of the data collection nodes, and consists entirely of loop body nodes. This section of the DFG is translated directly into a VHDL *component*. Then, the loop generator and collection nodes are implemented by selecting the proper VHDL components from a library, and by supplying these components with a parameters file containing information derived from the pertinent DFG nodes. Finally, the translator specifies the interconnections between the ILB and generator/collection components by generating two top–level VHDL modules, one for CPE0 and one for PEx, and a set of *project* files used by the Synplify VHDL compiler/synthesis tool. The files created in this last step serve to "glue" all the components together into a final design.

B. Translation of the ILB

The translation of the ILB involves a traversal of the dataflow graph. A VHDL *component* is created whose inputs are connected to the outputs of the loop generator, and whose outputs are connected to the inputs to the data collector. As an example, Figure 5 shows the VHDL that is generated for the ILB denoted by the dotted lines in the DFG of figure 1(b).

Many nodes implement simple operations, such as addition or logical operations; for these nodes, there is a one-to-one correspondence between DFG node and VHDL statement; for example, line 18 in figure 5 performs the subtract by 100. For more complicated operations, the translator generates a connection to a VHDL component; for example, lines 11-15 implement the SA-C *array sum* reduction operation. A library of such components has been written directly in VHDL; this allows a SA-C program access to operators that either cannot be expressed in the high level language or that have efficient direct hardware implementations. To facilitate the tracing of signals through the ILB, the names of the signals used to interconnect nodes are derived from the DFG node type and number. For example, the `signal` name `UGT9OUT0` in the VHDL example corresponds to the first (number 0) output of the `UGT` node numbered 9 in the DFG.

At present, the generated ILB is entirely combinational, although it is expected to eventually include multiple-cycle functions such as lookup tables, complex data reduction operations, and pipeline registers.

C. Implementation of the Other Components in a Design

In contrast to the ILB, which is generated from scratch by traversing the DFG, the data generators and collectors are created from VHDL components selected by the translator from a module library, and are parameterized with values extracted from the DFG.

An entire implementation, including the top-level glue modules mentioned earlier, is shown in figure 6. The operation of each component in the figure will be discussed in the following sections. The general flow of information is from top left to bottom right – the following discussion follows roughly the same order.

C.1 The ConstGrabber Component

Prior to initiating execution of the hardware, the host C program, created by the compiler as described earlier, downloads the run-time constants and input data to CPE0's memory and then resets the hardware. The `ConstGrabber` component then reads the run-time constants from memory and makes them available to the rest

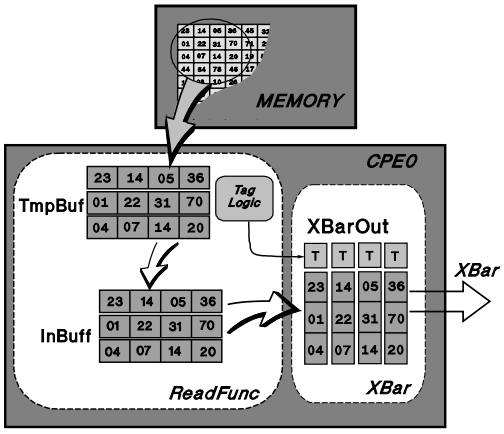


Fig. 7. Diagram of the read window generator function

of the system. When finished, the module sends a `Ready` signal to the rest of the system.

C.2 The Data Generator Components on CPE0

The data generator is responsible for retrieving data from memory in the proper order and buffering it, so it can serve as input to the inner loop body. At present, there are three types of data generators, selected because they seem to cover virtually all of the data access patterns required in IP applications. Scalar generators generate single values similar to the traditional `for` loops found in C and other languages. Element generators extract single values from an array. Window generators perform the more complex task of extracting small sub-arrays from a larger array. The discussion that follows specifically describes the window generator; the other generator types operate in a similar fashion (in fact, the element generator is currently implemented as a window generator with a 1×1 window).

The window generator function is distributed over the two PE's. The part that resides on CPE0 (called the *read* function) is the more complicated of the two parts; it is responsible for reading data from CPE0's memory and placing the data on the crossbar. It consists of two separate state machines – one which is responsible for retrieving data from memory (the *ReadData* state machine), and a second that sends the retrieved data out to the crossbar (the *XBar* state machine).

Initially, the `XBar` state machine waits for the `Ready` signal from the `ConstGrabber` module, then sends the retrieved run-time constants on the crossbar, so they are available to the components in PEx. Once this initial step occurs, the main operation of the read function starts. Figure 7 illustrates this operation for a 3×3 window example.

The `ReadData` state machine begins reading words of data from memory. The number of words read is equal to the number of columns in the window being used, and is called a *frame* of data. As the data is being read, it is stored in a buffer (`TmpBuf`); once all the words for a complete frame have been read, they are transferred to a second buffer called `InBuff`. This double buffering allows one frame to reside within CPE0 while the next frame is being retrieved from memory. Once a frame has been input, `ReadData` sends a signal to the `XBar` state machine and then begins reading the next frame.

Upon signal from `ReadData`, the `XBar` machine starts sending data along the crossbar. Data is stored in memory and therefore retrieved in row-major order; however, it is sent along the crossbar in columns. An interconnection network, generated using the static parameters passed to the component at compilation, performs the transposition from rows to columns. The resulting buffer (`XbarOut`) holds the data that is output to the crossbar.

Several signals are generated and sent as tag bits as `XBar` sends the data; these signals are used by PEx to help interpret the data:

- `ValidData` (crossbar bit 35) - Indicates that the value on the crossbar is a legal data item. When set to 0, the value currently on the crossbar is undefined (i.e. a `NULL`). `NULL`s are sent during cycles when no valid data is available to be sent, such as when `XBar` is waiting for data from `ReadData`.
- `Start/Stop` (bit 34) - Set to indicate the beginning and end of data.
- `DontStore` (bit 33) - Used to indicate that the calculations calculated by the ILB at this time step should not be stored. This situation occurs at the beginning of each row, and when the window generator is using a step size other than 1, as described below.
- `LastCol` (bit 32) - set to one when the values being sent are from the last column in a row.

In addition to sending window data in the proper sequence, the window generator code on CPE0 must handle several special situations. It is possible for windows to be created faster than the result data can be stored. This might occur when an ILB produces several output values for each input. In this case, `XBar` places `NULL` values on the crossbar to "slow down" the window generation rate. To implement this, the translator determines the number of cycles required by each component to process a frame of data, and then finds the maximum of these. This value is used to determine the number of `NULL`s (if any) that must be sent after each frame to keep the data generation rate from overrunning the output bandwidth. No `NULLS`s are sent if the writing of results can keep up with the generator.

The system must also handle window steps other than 1 – for example, a step of 2 specifies that only every other window is supposed to produce a value. In these situations, all of the data is still sent, and the ILB, being a combinational circuit, still calculates values at each time step; however, only some of the calculated results are stored. The `DontStore` signal determines when a value should not be stored. `DontStore` is also used to prevent the storing of results calculated at the beginning of each row, before the first complete window of data has been transmitted along the crossbar.

C.3 The Data Generator Components on PEx

That portion of the data generator that resides on PEx is called the *distribute* function. It retrieves the data from

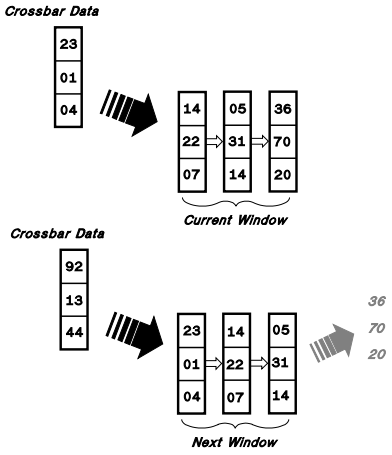


Fig. 8. Pictorial representation of a 3 x 3 sliding window generator for the current, and then next windows.

the crossbar and buffers it, so it can be presented to the inputs of the ILB. It also uses the tag bits sent on the crossbar to generate signals which control the other components on PEx. The distribute function consists of a single state machine.

In concert with operation of the code on CPE0, the `Distribute` state machine initially retrieves run-time data from the crossbar, then goes into its "normal" operation. The heart of the `Distribute` function is a shift register which buffers the window data. its operation is illustrated in figure 8. A "sliding window" effect is created by the shift register – when new data arrives from the crossbar, a new window is formed by taking the previous columns that had been sent, shifting them over, and then placing the new column of data into the space just vacated by the shift operation.

`Distribute` also generates a signal called `Storedata`, which is derived from the `ValidData and DontStore` tags bits from the crossbar. This signal inhibits the storing of data at the beginning of each row and during window stepping.

One special operator used with data generators is the `dot` (or dot product, although the operator does not imply that a product or any other arithmetic operation is to be performed). Consider the following SA-C statement:

```
for elem1 in image1 dot elem2 in image2
```

This syntax specifies that two (or more) data generators are to operate simultaneously, thereby providing two (or more) sets of data to the ILB at the same time. The latest version of the generator code has extended the window generator code to handle certain combinations of the `dot` operation. The operation of this generator is very similar to that of the single window. The primary difference is that a separate set of buffers on both CPE0 and PEx are instantiated for each generator. As the `XBar` machine sends data on the crossbar, first a column from one window is sent, then a column from the next window is sent, etc. The `Distribute` machine on PEx then places the data retrieved from the crossbar into the proper shifter register.

C.4 The Collector function

Once data has been presented to the input of the ILB, the resulting values must be stored in memory. The `Collector` component consists of a state machine which performs this function. The `StoreData` signal generated by `Distribute` causes the result value to be placed into a buffer; once an entire word of data has been collected, the buffer is sent to the `MemArb` component.

In addition to single values, the *collector* can also handle *tiles* of results. Tiles are small arrays of data, similar to windows, that can be produced by an ILB. Usually tiled output occurs as a result of the stripmining and tile optimizations applied by the compiler – an original (single) ILB is transformed into several instantiations, with the resulting output being a tile whose size and shape is determined by the order in which the compiler optimizations were applied. The tile shape is determined at compile time, so buffers are instantiated within `Collector` to hold the resulting tile values.

A different situation occurs for a single ILB which produces multiple (independent) values. In this case, a separate `Collector` component is instantiated for each output. This method is used so that the proper size and shape of buffers can be independently instantiated for each output. The PEx glue code is responsible for providing the interconnection for each instance of the `Collector` component.

With multiple and tiled outputs being produced by the ILB, it is possible that a single set of input values can produce multiple outputs, thus causing the output bandwidth to exceed the input. As discussed earlier, the `Collector` function does not need to worry about this, since the window generator adds wait cycles if necessary to guarantee that the input rate will not cause data overrun on output.

C.5 The MemArb function

The final component in the system is the `MemArb` component. This module receives words of data to be stored in memory, and performs the storing operation. In contrast to the `Collector`, which is instantiated once for each set of outputs, there is always only one `MemArb` component – it handles multiple store requests from possibly several `Collectors`. A simple priority encoder determines which output is stored first. However, since the window generators never produce data faster than the results can be stored, the exact order in which the stores occur is not important because even the lowest priority values will have time to store.

VI. An Example: the Prewitt Algorithm

The Prewitt algorithm is a well known edge-detection algorithm used in many IP applications; its development and operation is described in [28], and in most IP texts, such as [29]. The algorithm involves the convolution of an image with two constant 3×3 masks – one which forms the X gradient, and one which forms the Y gradient; the two masks are shown in figure 9. The results of these convolutions form two vector components; the magnitude of the

$$\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{pmatrix} \quad (a) \qquad \begin{pmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{pmatrix} \quad (b)$$

Fig. 9. Constant convolution masks for the Prewitt algorithm. (a) the X gradient mask, and (b) the Y gradient mask.

```
uint8[:,:] prewitt(uint8 Image[:,:]) {
           int2 H[3,3] = {{-1,-1,-1},
                          { 0, 0, 0},
                          { 1, 1, 1}};
           int2 V[3,3] = {{-1, 0, 1},
                          {-1, 0, 1},
                          {-1, 0, 1}};
 uint8 res[:,:] =
   for window W[3,3] in Image {
     int11 sh, int11 sv =
      for h in H dot w in W dot v in V
        return (sum( (int11)w*h ),
                sum( (int11)w*v ));
    } return (array (magnitude(sh,sv)/8));
  } return (res);
uint11 magnitude(int11 a, int11 b)
  return (sqrt( (int22)a*a + (int22)b*b ));
```

Fig. 10. The Prewitt edge-detection algorithm, written in SA-C

resulting vector forms the desired result. This fundamental operation is performed over the entire image, with the result being another $2D$ array called the *gradient array*.

While this algorithm is important in its own right as a fundamental IP operation, it is an interesting example for other reasons:

- It is inherently a parallel operation, with each 3×3 convolution being entirely independent of the others,
- It involves constant masks, which allows considerable optimization before being implemented in hardware,
- It presents some computational challenges, requiring a squaring (multiplication) and square root, which are difficult on FPGAs,
- It involves the use of a streaming data model from host to RCS back to host, which is common in IP applications,
- It is representative of a large number (perhaps even the majority) of other common IP applications. In fact, the SA-C language contains *window generators* to provide a simple means for expressing the extraction of a small subarray from a larger one.

A SA-C program which performs the Prewitt calculation is shown in figure 10.

The SA-C compiler performs several of the optimizations described earlier on this program before producing the DFG. Since the convolutions involve multiplications with 3×3 masks that are composed of the constants 1, 0, and -1, the compiler optimizes the calculation to a series of additions and subtractions. Multiplications with zero are eliminated completely. These optimizations eliminate all multiplies, and reduce the number of addition/subtractions from 16 down to 10.

The `magnitude` function is the most expensive part of the ILB, since it involves the squaring of the two results (requiring a multiply), then finding the square root. An efficient square root routine is used which uses only shifts, adds, and bit operations. Nonetheless, the multiply/square-root operation consumes over 50% of the space, and requires more than 80% of the time required by the entire ILB.

TABLE I

Statistics for the Wildforce implementation of the Prewitt algorithm using the SA-C compiler/translator.

(a) Lines of Code

SA-C	19
VHDL	
`WINDOW-GEN` - PE0	572
`WINDOW-GEN` - PE1	194
Generated Inner Loop Body	3744
`WRITE-VAL`	406
Glue Code and Misc	199
Total VHDL	5115

(b) FPGA Logic cell (CLB) Usage

`WINDOW-GEN` - PE0	251	(19.4%)
`WINDOW-GEN/WRITE-VAL` - PE1	236	(18.2%)
Inner Loop Body	281	(21.7%)
Glue Code and Misc	19	(1.5%)
Total - PE1	536	(41.3%)
Total CLBs	787	(30.4%)

(c) Propagation Delay - Inner Loop Body

`ILB - Convolution`	58.1 ns	
`ILB - Magnitude`	295.9 ns	
Total Propagation Delay	354.0 ns	(2.8 MHz)

(d) Execution Times (msec)

	No stripmine	4 × 3 stripmine	Manual VHDL	Pentium
Data Download	0.54	0.54	?	—–
Computation	59.14	30.10	4.66	28.4
Result Upload	0.93	0.93	?	—–
Total time	60.61	31.57	4.66+	28.4
Freq(MHz)	2.82	2.78	16.9	450

The resulting DFG is processed by the DFG-to-VHDL translator, which extracts the ILB and translates it directly to VHDL, selects the appropriate generator and collector components from the VHDL library, and creates the top-level VHDL program which "glues" the entire system together. The translator also creates the script files needed by commercial design tools to compile and place-and-route the VHDL into FPGA configuration codes. These files, along with a compilation script that controls the numerous steps in the compilation process, allows the entire process, from high level language compilation down to the production of FPGA configuration codes and the host-based control program, to be fully automated. The user can execute the entire algorithm on the hardware like any other application (by typing `a.out` or something similar), without needing to worry about any of the operational details of the hardware.

A. Preliminary Performance Results

Table I shows some of the statistics of the entire compilation/translation process. The 19 line SA-C program for the Prewitt algorithm eventually requires over 5000 lines of VHDL, and occupies approximately 30% of the CLBs in the

two FPGAs used in the implementation. Table I(c) shows the long delay of the magnitude function discussed earlier. Finally, Table I(d) shows the execution performance of the algorithm, first with no stripmining (i.e., a single inner loop body), and then with 4×3 stripmining (which results in two inner loop bodies). As expected, increasing the parallelism from one to two essentially doubles the processing rate.

Two comparisons of the execution results obtained by the SA-C system are appropriate. First, the Prewitt algorithm was coded manually in VHDL, with two inner loop instantiations (equivalent to a 4×3 stripmine in the SA-C version), and with the magnitude computation replaced with a lookup table. The design was further optimized by pipelining the resulting inner computation. This design was meant to represent optimal performance of the algorithm on the RCS. The resulting design runs at 17 MHz, shown in the third column of Table I(d), compared to only 2.82 MHz achieved by the automated design. Clearly the manual version enjoys a major advantage with its lookup table version of magnitude – it executes more than 6 times faster than the equivalent (stripmined) SA-C version. We believe that it is reasonable to expect that the performance of the SA-C version can be improved several fold, to within a few percentage points of the manual version, by using the same optimizations used in the manual method.

The second comparison is with an equivalent algorithm running on a Pentium. A C version of prewitt was coded in C and compiled with the `gcc -O6` option; the results of execution on a 450MHz Pentium is shown in the last column of Table I(d). The results are compared to the stripmined version of the SA-C version. We are encouraged by these results, since we believe we can improve the execution time several fold. Porting the SA-C system to more modern hardware will improve performance even more.

Perhaps more important than performance is the saving gained in development time by using the automated approach. The entire Prewitt algorithm was coded in SA-C and converted to hardware in a matter of a few hours, compared with several weeks required by the manual method. Equally important, the development process can be carried out by an application programmer, with little or no knowledge of hardware design or VHDL. Control of compiler operation via pragmas allows the programmer to optimize the design by changing the parallelization included in the application.

VII. Future Work and Conclusion

Up to now, most of the project effort has focused on functionality, rather than optimal performance, as evidenced by the performance numbers. Work on the compiler always stays several steps ahead of the DFG-to-VHDL translation effort. Nonetheless, a large portion of the SA-C language can now be translated to hardware; unimplemented language features are added on an as-needed basis. Numerous compiler optimizations which control how an application is mapped to hardware, such as stripmining and loop fusion, have been completed.

Now attention is turning toward optimizing the translation from DFG to hardware. In particular, two areas of optimization hold considerable promise:

- Lookup tables - many common operations are inefficient on reconfigurable hardware. We are implementing a scheme which allows a SA-C function to be converted to a lookup table, via a pragma. The original function is then replaced with table lookup code, which is much faster, although it often requires more space.
- ILB Pipelining - calculations in the ILB's tend to create circuits with long propagation delays, which require slow clock speeds. The propagation delays can be reduced by strategically placing pipeline registers in the ILB.

New technology promises to provide impressive performance gains, both in speed and in the amount of space available for programming. New generations of FPGAs provide resources, such as on-chip memory, which can easily be utilized to enhance the current system performance.

We are currently porting the system to a Virtex-based AMS Starfire board. Preliminary results indicate that the new board can accommodate applications up to 30 times larger than those on the current board. With no board-specific optimizations, applications execute 3-5 times faster than the same application on the older board.

Reconfigurable computing holds the promise of significant performance gains over conventional computing for certain types of problems. The automated process for application development described here promises to greatly reduce software development times for such systems, and to bring the realm of hardware design to the application programmer.

References

[1] "The Cameron Project," Information about the Cameron Project, including several publications, is available at the project's web site, `www.cs.colostate.edu/cameron`.

[2] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE CS Press, 1996.

[3] Xilinx, Inc., San Jose, CA., *The Programmable Logic Databook*, 1998, `www.xilinx.com`.

[4] D.T. Hoang, "Searching genetic databases on Splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines.* 1993, pp. 185–192, CS Press, Loa Alamitos, CA.

[5] D. V. Pryor, M. R. Thistle, and N. Shirazi, "Text searching on Splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines.* 1993, pp. 172–178, CS Press, Loa Alamitos, CA.

[6] N. K. Ratha, D. T. Jain, and D. T. Rover, "Fingerprint matching on Splash 2," in *Splash 2: FPGAs in a Custom Computing Machine*, pp. 117–140. IEEE CS Press, 1996.

[7] N. K. Ratha, D. T. Jain, and D. T. Rover, "Convolution on Splash 2," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines.* 1995, pp. 204–213, CS Press, Loa Alamitos, CA.

[8] P. M. Athanas and A. L. Abbott, "Processing images in real time on a custom computing platform," in *Field-Programmable Logic Architectures, Synthesis, and Applications*, R. W. Hartenstein and M. Z. Servit, Eds., pp. 156–167. Springer-Verlag, Berlin, 1994.

[9] Annapolis Micro Systems, Inc., Annapolis, MD, *WILDFORCE Reference Manual*, 1997, `www.annapmicro.com`.

[10] Annapolis Micro Systems, Inc., Annapolis, MD, *STARFIRE Reference Manual*, 1999, `www.annapmicro.com`.

[11] B. Schott, S. Crago, Chen C., J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, "Reconfigurable architectures for systems level applications of adaptive computing," Available from `http://www.east.isi.edu/SLAAC/`.

[12] Xilinx, Inc., *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999, `www.xilinx.com`.
[13] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi, "The Morphosis parallel reconfigurable system," in *Proc. of EuroPar 99*, Sept. 1999.
[14] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: RAW machines," *Computer*, September 1997.
[15] W.H. Mangione-Smith, "Seeking solutions in configurable computing," *IEEE Computer*, vol. 30, pp. 38–43, Dec. 1997.
[16] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: A coprocessor for streaming multimedia acceleration," in *Proc. Intl. Symp. on Computer Architecture (ISCA '99)*, 1999, `www.cs.cmu.edu/~mihaib/research/isca99.ps.gz`.
[17] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, and M. Srikrishna, D.and Taylor, "The RAW compiler project," in *Proc. Second SUIF Compiler Workshop*, August 1997.
[18] S. C. Goldstein and M. Budiu, *The DIL Language and Compiler Manual*, Carnegie Mellon University, 1999, `www.ece.cmu.edu/research/piperench/dil.ps`.
[19] OXFORD Hardware Compiler Group, "The Handel language," Tech. Rep., Oxford University, 1997.
[20] M. Gokhale, "The Streams-C Language," 1999, `www.darpa.mil/ito/psum19999/F282-0.html`.
[21] "SystemC. SystemC homepage," `www.systemc.org/`.
[22] "IMEC. Ocapi overview.," `www.imec.be/ocapi/`.
[23] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden, "MATCH: a MATLAB compiler for configurable computing systems," Tech. Rep. CPDC-TR-9908-013, Center for Parallel and distributed Computing, Northwestern University, August 1999.
[24] S. Periyayacheri, A. Nayak, A. Jones, N. Shenoy, A. Choudhary, and P. Banerjee, "Library functions in reconfigurable hardware for matrix and signal processing operations in MATLAB," in *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conf. (PDCS'99)*, November 1999.
[25] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin, "Automatic mapping of Khoros-based applications to adaptive computing systems," Tech. Rep., University of Tennessee, 1999, Available from `http://microsys6.engr.utk.edu:80/~bouldin/darpa/mapld2/mapld_paper.pdf`.
[26] K. Konstantinides and J. Rasure, "The Khoros software development environment for image and signal processing," in *IEEE Transactions on Image Processing*, May 1994, vol. 3, pp. 243–252.
[27] J. Hammes and W. Böhm, *The SA-C Language - Version 1.0*, 1999, Document available from `www.cs.colostate.edu/cameron`.
[28] J. M. S. Prewitt, "Object enhancement and extraction," in *Picture Processing and Psychopictorics*, B. S. Lipkin and A. Rosenfeld, Eds. Academic Press, New York, 1970.
[29] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Addison-Wesley, Reading, MA, 1992.