Compiling a High–level Language to Reconfigurable Systems

Jeff Hammes, Bob Rinker, Wim Böhm, Walid Najjar
Computer Science Department, Colorado State University

July 8, 1999

1 Introduction

Image processing (IP) applications feature large, regular image data structures with regular access patterns and therefore can benefit from parallel implementations. The programmable nature of parallel reconfigurable computing systems (RCSs) allows great flexibility and promises massive fine-grain parallelism with high throughput. RCSs 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.

Reconfigurable computing systems are typically based on Field Programmable Gate Arrays (FPGAs), which are large arrays of programmable logic cells. These systems present new challenges to language designers and compiler writers. The task of programming and compiling applications consists of partitioning the algorithm between a host processor and reconfigurable modules, and devising ways of producing efficient FPGA configurations. Presently, FPGAs are programmed in hardware description languages, such as VHDL or Verilog. While such languages are suitable for chip design, they are poorly suited for the kind of algorithmic expression that takes place in IP programming.

The programming language SA-C, developed as part of the Cameron Project [1], is designed to provide high level, algorithmic language support for RCSs, allowing applications experts, rather than only hardware experts, to reap the benefits of these systems. This paper introduces SA-C, its optimizing compiler that generates dataflow graphs (DFGs), and the mapping of the DFGs to reconfigurable systems.

2 SA-C: Single Assignment C

SA-C [5] combines features of existing imperative and functional languages in order to create a language well suited to image processing and amenable to compiler analysis and optimization. It is based on C in order to be as intuitive as possible to image processing experts. It is an *expression-oriented*, single assignment language, and has scalar types that include signed and unsigned integers and fixed point numbers with specified bit widths. It has no explicit pointers, and is non-recursive. Its true multi-dimensional arrays are similar to those in Fortran 90.

The elimination of pointers and recursion, along with the single-assignment restriction, enable important compiler code optimizations. Loops and arrays are at the heart of the language, and are closely interrelated. Loops have special forms designed to work with arrays, and arrays are easily created as return values of loops. The parallel `for` loop is the source of coarse-grain parallelism and has three parts: one or more *generators*, a loop *body*, and one or more *return* values. There are three kinds of loop generators: *scalar*, *array-component* and *window*. The window generator allows a rectangular window to traverse the source array. `Dot` and `cross` products combine generators: the `dot` product runs the generators in lock step, whereas cross products produce all combinations of components from the generators. Every construct, and in particular every loop, returns one or more values. In addition to returning scalar values, a loop can return arrays and reductions built from values that are produced in the loop iterations, including `sum`, `product`, `min`, `max`, `mean`, `median`, and `histogram`.

Loop generators make compiler analysis of array access patterns easy. In C or Fortran, the compiler must analyze index expressions in loop nests and infer array access patterns from these expressions, but in SA-C, the index generators and the array references have been unified, so the compiler can reliably infer the patterns of array access.

The compiler uses an intermediate form called a *Data Dependence and Control Flow* (DDCF) graph, in which the high-level subgraphs have a direct correspondence to the constructs of the source language. This graph form exposes data dependencies, opening up a wide range of loop- and array-related optimization opportunities, including code motion, function inlining, (full) loop unrolling, loop fusion, common subexpression elimination, array and loop size propagation, constant folding, array value propagation, and algebraic simplification. The array and loop size propagation pass is vitally important in this compiler, since it exploits the close association between arrays and loops in the SA-C language. Source array size information can propagate through a loop and into its result arrays (downward flow), and result array size information can be used to infer the sizes of source arrays (upward flow). In addition, since the language requires that the individual generators enclosed in a dot product graph have identical shapes, size information from one generator can be used to infer sizes in the others (sideways flow).

```
int16[:,:] main (uint8 Image[:,:]) {
     int16 H[3,3] = {  {-1,-1,-1}, { 0, 0, 0}, { 1, 1, 1}} ;
     int16 V[3,3] = {  {-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}} ;
     int16 M[:,:]  =   for window W[3,3] in Image {
                         int16 dfdy, int16 dfdx=
                                 for h in H dot w in W dot v in V
                                 return( sum(h*w), sum(v*w) );
                         int16 magnitude = sqrt(dfdy*dfdy+dfdx*dfdx);
                       } return( array(magnitude) );
} return(M);
```

Figure 1: SA-C code for the Prewitt Edge detector.

As an example, Figure 1 shows SA-C code for the Prewitt edge detector [4]. Size propagation allows the compiler to determine that the inner loop will execute nine iterations, with a 3×3 shape. Since the number of loop iterations is known and is not large, the inner loop can be fully unrolled. Then array value propagation replaces array reference nodes of the H and V arrays with their values. Since all multiplications are by simple constants, the multiply nodes can be removed.

The optimized DDCF graph can now be transformed into a DFG (see figure 2). It is flat (not hierarchical), and it has operators suitable for generating VHDL code to represent FPGA configurations. The `SUM` nodes can be implemented in a variety of ways, including a tree of simple additions. The window generator allows a variety of implementations, based on the use of shift registers. The `CONSTRUCT_ARRAY` node simply streams its values out to a local memory or the host, as appropriate.

3 Translation of the DFG to VHDL

A major challenge in compiling a high level program to a RCS is due to the large amount of unstructured, low level logic hardware: it does not have the semantic equivalent of an *instruction set architecture* as in a traditional processor. Rather than target a particular reconfigurable architecture, an *abstract architecture* has been defined that provides a *structured semantic framework* that can be used by the compiler to convert DFGs to VHDL circuit descriptions.

After DDCF-level optimizations have been applied, a loop may bear little resemblance to its original form. It has one or more generators, expressed as data distribution operators; a loop body, built from simple and compound operators; and one or more reduction operators that produce the return values. Therefore, the abstract architecture consists of three main types of operators:

- *Arithmetic and logic operators* have well-defined semantic meanings that can be instantiated for a variety of logic data types (e.g signed or unsigned logic) and bit widths. These include several application-domain specific functions, targeting image processing applications.
- *Data distribution operators* define the data structures and operators necessary for supplying the RCS with the required data.

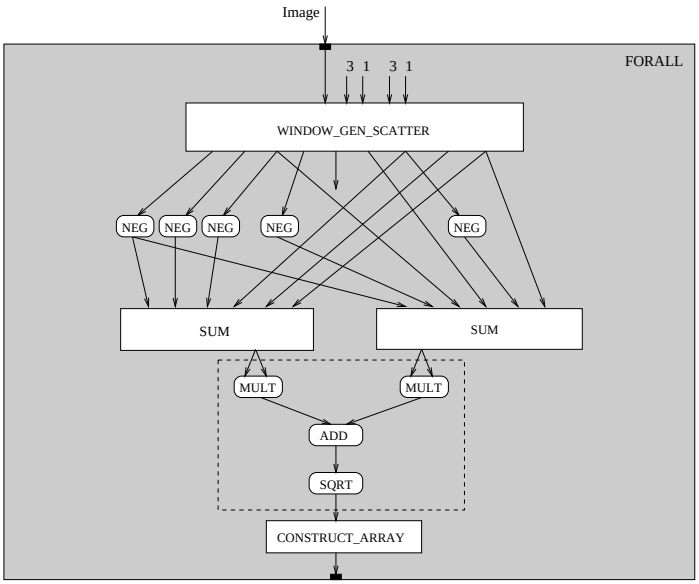

Figure 2: Data flow graph after full optimizations.

- *Data collection operators* reassemble the result data into a format accessible to the next stage of computation or for output.

The arithmetic and logic operators can be expressed in VHDL in a platform-independent way, and are used to form *combinational* circuits that perform the calculations contained in the loop bodies. Three categories are defined:

- *Simple operators* include addition, subtraction, AND, OR, logic and arithmetic shifts, compare, select, etc.
- *Compound operators* are complex operations that are costly to implement as calculations on the RCS but can be efficiently implemented as lookup tables by precomputing all possible values of the function based on the data type and size of the operands, and including the table values in the DFG. Examples of such operators include square, square root, vector magnitude and angle, polar to Cartesian coordinate conversion. The current SA-C compiler is able to compute these tables, and the latest generation of FPGAs, such as the Xilinx Virtex family, provides efficient support for them in hardware.
- *Reduction operators* are applied to a stream of data and return either a single value or an array of values. They are closely related to the reduction operators found in SA-C. Examples include sum, max, min, mean, median, histogram and population count.

Data distribution operators convert the data access pattern specified by a generator node in the DFG into a properly ordered stream of data that is presented to the loop body circuits. The distribution process involves accessing a local memory or a port to host memory, retrieving the image data and buffering it. Buffer operations play an important role in the process: since the bandwidth to the memory is often a bottleneck, properly constructed buffers minimize memory accesses by allowing the reuse of previously retrieved data. For example, the very common operation of accessing an image using a sliding "window" can be implemented using a set of buffers connected as shift registers. The "sliding" effect is implemented by shifting the existing rows (or columns) and bringing a new row (or column) into the first buffer; only the new data that is not in the previous window need to be retrieved for the new operation. The design of the buffers also influences the ability to replicate circuits to increase parallelism, as described below. Rather than being implemented using combinational logic, the data distribution operators are implemented as state machines that control the buffer operations. Some of the details involving the access of input ports or local memory are system dependent, but most of the buffer code can be expressed in general purpose VHDL.

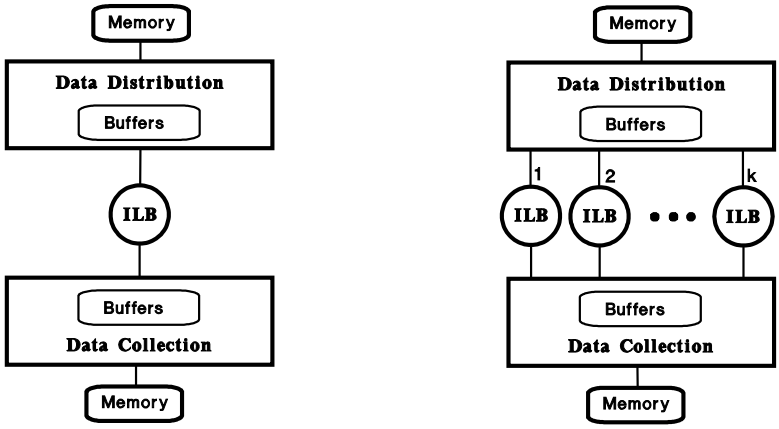

Figure 3: Implementation using single (left) and multiple (right) inner loop body (ILB) instantiations.

The data collection operators perform the complement of the data distribution operators – they collect the results from the loop body circuits, and store them in local memory or output ports to host memory. Their implementation is similar to the data distribution operators, and consist of data buffers and state machines that control the buffer operations. In fact, in a pipelined operation the data collection operation of one stage becomes the data distribution operator for the next. Since the reduction operations often occur during the data collection process, certain reduction operations are incorporated directly into the data collection process.

The overall view of an application implemented using this abstract architecture is shown at left in figure 3. The data distribution operators are state machines that retrieve data from memory and place it into buffers. The loop bodies are combinational circuits that are connected to the input buffers; the calculation proceeds through these circuits at a rate determined by the propagation delay of the circuit technology. Finally, the data collection operators take the results from the loop body circuits and store them into memory. A clock on the RCS controls the speed at which the data distribution/collection state machines operate, and the clock frequency is determined by the maximum propagation delay through the loop body circuits.

One important RCS performance enhancement can be achieved through the instantiation of multiple copies of the calculation. In the abstract architecture model, this involves replication of the loop body circuits and modification of the data distribution/collection operators to connect to the additional replications. The resulting configuration is shown at right in figure 3. The entire system now takes the form of a set of k-bounded loops[2], with the value of k determined by the number of loop bodies that can fit on the RCS, as well as the bandwidth necessary to support the multiple instances.

4 An Example: the Prewitt Algorithm

The Prewitt edge detector function, whose SA-C code and optimized dataflow graph were shown in figures 1 and 2, helps to demonstrate many of the concepts discussed above, including the mapping of the abstract architecture to real hardware. The Prewitt algorithm represents a typical image processing task, but it is also interesting from a compilation and mapping point of view. It accesses the image using a sliding window, and it has simple (e.g., additions) as well as compound (multiplications and square root) operators in its loop body. Moreover, when loop bodies are *replicated*, i.e., more than one loop body is allocated per FPGA, the data collection operators write *tiles* rather than individual elements to the local memory.

The Prewitt algorithm has been implemented on the Annapolis Micro Systems Wildforce$^{(TM)}$ reconfigurable computing board[6], mapping the dataflow graph of figure 2 via VHDL into a hardware configuration bit stream. The loop body consists of two parts: a set of simple operators (additions and negations), computing a horizontal and a vertical differential, followed by a set of more complex operators (multiplications, addition, and square root, see the dashed subgraph in figure 2) computing a gradient *magnitude*. The nodes in the dashed subgraph are better implemented using a single lookup table that resides in the local memory of the Wildforce board. Using this lookup

table technique, one inner loop body takes three cycles to compute: one cycle for the combinational part of the computation, and two cycles for the table lookup.

The Wildforce board used here consists of five user-programmable FPGAs (Xilinx XC4036 chips [7]) labeled PE0 through PE4, and connected such that PE0, the input processor, is connected to the other four PEs via a 32-bit (i.e. four-pixel) wide crossbar. The data distribution operation (`WINDOW_GEN_SCATTER` in figure 2) is split between PE0, which retrieves data from its local memory and sends it along the crossbar, and the other PEs that take the data from the crossbar and buffer it. The inner loop body is replicated twice on each PE, resulting in eight concurrent loop bodies. The data collection operators write 2×1 sized data *tiles* to the local memory. This design uses the full bandwidth available on PE0 and the crossbar. Transfers between host and PE memories occur using DMA across a PCI bus connection.

The measured performance of this implementation is encouraging. The raw aggregate computation rate of the algorithm exceeds 10 Megapixels per second (MP/s) for a 512×512 image. Even when the host–RCS transfer time is included, the effective computation rate is better than 7.5 MP/s. When the propagation delay through the inner loop body is reduced by pipelining (inserting strategically-placed registers in the combinational circuits), these rates can increase to a raw rate of 20 MP/s, and an effective rate of 10.5 MP/s. This compares with a rate of just over 2 MP/s for a 450 MHz Pentium when executing the algorithm written in C. More specific details of the Prewitt implementation and its performance are available in [3]. Note that in the Prewitt example the host–RCS input and output transfer limits the effective computation rate. It is therefore important to minimize the host–RCS traffic, e.g., by avoiding intermediate data to be communicated with the host.

5 Conclusion and Future Research

This paper presents the compilation of a high-level, algorithmic programming language to FPGA based Reconfigurable Computing Systems. The translation of a typical Image Processing algorithm, the Prewitt edge detector, has been implemented and studied.

At present, not all the steps in the compilation process described here are fully automated or integrated with each other. In the near future the front-end and host code generator, the optimizer and dataflow code generator, and the dataflow graph to VHDL translator, will be connected together. In the longer term we will experiment with more optimizations, such as loop fusion and loop tiling to avoid intermediate array construction, and with various dataflow graph to VHDL code generation methodologies.

References

[1] The Cameron Project: http://www.cs.colostate.edu/cameron/

[2] David E. Culler. "Managing Parallelism and Resources in Scientific Dataflow Programs", MIT/LCS/TR-446, 1990.

[3] R. Rinker, A. Patel, G. Yarbrough, W. Najjar. "Implementation of the Prewitt Algorithm on the Wildforce-XL(tm) Reconfigurable Computing Board" CSU CS Technical Report, 1999.

[4] J. M. S. Prewitt. "Object Enhancement and Extraction", in Picture Processing and Psychopictorics, Lipkin and Rosenfeld (eds), Academic Press, New York, 1970.

[5] J.H. Hammes and A.P.W. Böhm. "The SA-C language, version 1.0". CSU CS Technical Report, 1999.

[6] WILDFORCE^{TM} Reference Manual. Annapolis Micro Systems, Inc., 1999

[7] Xilinx. *The Programmable Logic Data Book.* Xilinx, Inc., San Jose, California, 1998.