Compiling Image Processing Applications to Reconfigurable Hardware *

Robert Rinker, Jeff Hammes, Walid A. Najjar, Wim Böhm, Bruce Draper
Computer Science Department
Colorado State University
Ft. Collins, CO 80523-1873
{rinkerr,hammes,najjar,bohm,draper}@cs.colostate.edu

Abstract

This paper describes the compilation of high-level language programs written in a single-assignment language called SA-C into the binary codes used for programming reconfigurable hardware. The primary application domain is image processing. The paper describes the SA-C language, the compiler and the optimizations it performs, the process of converting the intermediate form called dataflow graphs into VHDL, and the generation of hardware configuration codes. Performance data on a typical image processing program, written in SA-C and executed on a reconfigurable computing system, is presented and compared to a hand-written VHDL version and a C version running on conventional processors.

1: Introduction

This paper presents a complete compilation path from an algorithmic programming language, SA-C, to a reconfigurable computing system (RCS). SA-C has been designed to express Image Processing (IP) applications on a high level, while being amenable to efficient compilation to fine grain parallel hardware systems. Reconfigurable computing systems are typically based on FPGAs, which are large arrays of programmable logic cells and interconnections. Multiple FPGAs, local memories, and interface hardware are packaged as co-processor boards. For most reconfigurable systems today, 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 for each piece of code. FPGAs are programmed in hardware description languages (HDLs), such as VHDL or Verilog. While such languages are suitable for chip design, they are not well suited for the kind of algorithmic expression that takes place in applications programming.

Applications that may benefit from the use of reconfigurable systems are those that exhibit regular and stream-oriented high bandwidth behavior. IP applications fit this category, as they feature large, regular image data structures with regular access patterns and can benefit tremendously from parallel implementations. To bring reconfigurable computing to applications programmers, the Cameron Project [7] has created a high level *algorithmic programming language*, SA-C, which can be mapped automatically to reconfigurable

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

hardware. The SA-C language is hardware independent, but still is able to be mapped to reconfigurable hardware. It allows easy extraction of fine grain parallelism, and the results can be optimized to minimize hardware space and execution time.

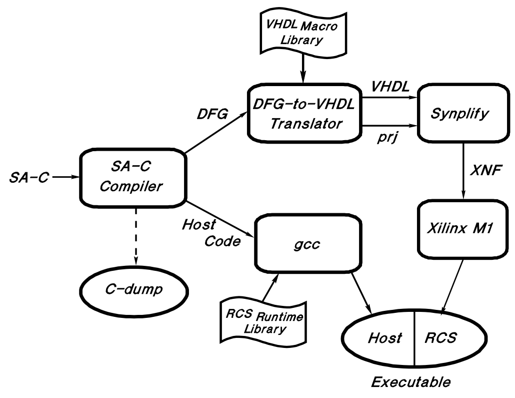

Figure 1. System overview

The compilation, simulation and execution path, shown in Figure 1, uses data dependence analysis for optimizations. Code partitioning between host and RCS is directed by the user. The RCS part is converted to dataflow graphs (DFGs) that are then translated via VHDL to FPGA codes. The system allows executable code to be generated at various stages of the compilation process for validation and simulation purposes. The end result of the compilation process consists of host code, host/RCS interface code, and RCS configuration codes.

The rest of this paper is structured as follows. Section two introduces the SA-C language, and section three describes dataflow graphs. Section four discusses an abstract target machine model and presents the translation of dataflow graphs to the VHDL code implementing this abstract machine. Section five discusses optimizations and pragmas to improve code size, to fit the computation on the FPGA hardware, and to improve execution time. Section six presents the compilation path and the behavior of an IP relevant example code. Section seven concludes.

2: The SA-C Language

The Cameron Project has created the high level language SA-C (derived from "single-assignment C," but pronounced "sassy") for compilation to reconfigurable hardware. The design goals are *single-assignment*, for easy compiler analysis and translation to DFGs; *no pointer arithmetic*, for easy compiler analysis; *high-level reduction operators* for IP applications; *variable bit-width data types* for efficient use of FPGA space; and *user control of optimizations*. SA-C draws ideas from a number of languages. The general syntax is derived, as much as possible, from C; powerful multi-dimensional array capability is drawn from Fortran 90; SA-C's loop generators are inspired by Sisal [9].

Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. For example, a **uint8** is an 8-bit unsigned type, whereas a **fix12.4** is a fixed point type with a sign bit, seven whole number bits and four fractional bits. SA-C also has **float** and **double** types that hold 32- and 64-bit values, respectively. There is also a **bool** type, a **bits** type that can hold non-numeric bit vectors, and complex types composed of the language's signed numeric types. Since SA-C is a single-assignment language, a variable is declared and given a value simultaneously. For example,

```
uint12 c = a + b;
```

gives `c` both its type and its value.

SA-C has multidimensional rectangular arrays whose extents are determined dynamically during program execution. (A user may specify any array's extents statically, however.)

int14 M[:,6] is a declaration of a matrix **M** of 14-bit signed integers. The left dimension is determined dynamically; the right dimension is specified by the user to be six. Arrays in SA-C can be sectioned or *sliced* using a syntax similar to Fortran 90: the expression **A[:,i]** for example returns the ith column of array A.

The most important part of SA-C is its treatment of **for** loops. A loop in SA-C returns one or more values (i.e., a loop is an expression), and has three parts: one or more generators, a loop body and one or more return values. The generators interact closely with arrays, providing array access expression that is concise for the user and easy for the compiler to analyze. Most interesting is the **window** generator, which extracts sub-arrays from a source array. Here is a median filter written in SA-C:

```
uint8 R[:,:] =
 for window W[3,3] in A {
  uint8 med = array_median (W);
} return (array (med));
```

The **for** loop is driven by the extraction of 3x3 sub-arrays from array A. All possible 3x3 arrays are taken, one per loop iteration. The loop body takes the median of the sub-array, using a built-in SA-C operator. The loop returns an array of the median values, whose shape is derived from the shape of A and the loop's generator. SA-C's generators can take windows, slices and scalar elements from source arrays, making it frequently unnecessary for source code to do any explicit array indexing whatsoever.

In the complete host scenario, the compiler produces host executable for the entire program. In the host/co-processor scenario, the compiler produces host code, host/co-processor interface code, and the configuration codes for the co-processor. In the second scenario the compiler transforms bottom-level loops into dataflow graphs (DFGs), suitable for mapping onto FPGAs. The host code includes interface code that downloads FPGA programs and source data, and uploads results on the host.

3: Dataflow Graphs

The SA-C compiler translates that part of the program targeted to the reconfigurable system to dataflow graphs: a low-level, non-hierarchical and asynchronous program representation. DFGs can be viewed as abstract hardware circuit diagrams without timing considerations taken into account. Nodes are operators and edges are data paths. DFGs are designed to allow token driven simulation, used by the compiler writer and applications programmer for validation and debugging.

The main node types are arithmetic, low level control (e.g. selective merge), and data extraction and routing nodes. These nodes reflect window generators driving loops, and array generators returning values out of loops.

As an example, consider the following SA-C fragment:

```
int16 H[3,3] = {  {-1, 0, 1},
                  {-1, 0, 1},
                  {-1, 0, 1}  } ;
int16 R[:,:] = for window W[3,3] in Image {
   int16 iph =  for h in H dot w in W
                return( sum(h*w) );
  } return( array(iph) );
```

This code performs the convolution of a 3×3 constant mask H over a larger input array Image, as one might see in an edge detection routine such as the Prewitt edge detection algorithm [12].

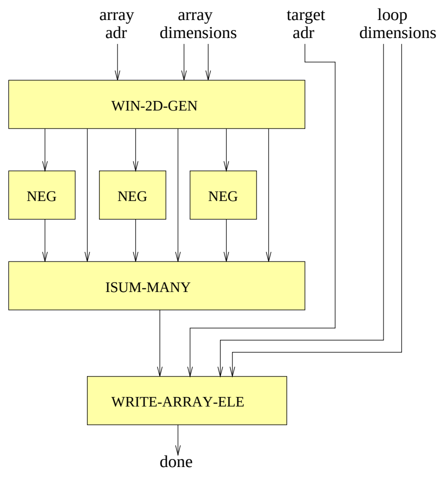

Figure 2. Dataflow Graph

The dataflow graph for this code is shown in Figure 2. Both loops of this nested structure are converted to a single dataflow graph by the SA-C compiler – i.e., the inner loop has been unrolled. The Window-Generator node near the top of this graph reads elements from a 3×3 window of the *Image* array at each iteration, and as this window of data flows through the graph, the required convolution with H is performed. Notice the multiplications explicit in the code have been removed by the compiler, replaced with either no-ops (for multiplication by 0 or 1) or negate operations (for multiplication by -1). Thus, the nine multiplications and eight additions explicit in the code at each iteration of the loop have been replaced with three negation operations and five additions at each iteration.

4: Abstract Machine

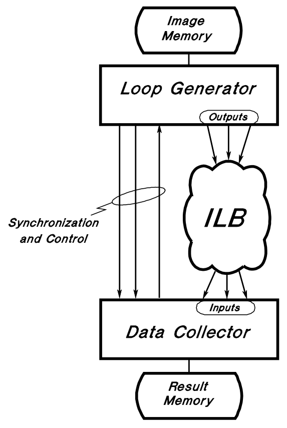

Figure 3. Abstract machine structure

Unlike standard processors, 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 many ways. To limit the number of possibilities available to the compiler, an *abstract machine* has been defined - this abstract machine provides a hardware independent model which serves as a reasonable compiler target machine.

The abstract machine model is shown pictorially in Figure 3. The DFG for a SA-C program consists of one or more data generators, which read data from RCS local memory and present it in the proper sequence to the main computational section of the program, called the inner loop body (ILB). Values that are calculated by the ILB are then "collected" together before being written to memory. The ILB is, at least at the present time, entirely combinational; all timing and control of the computation process is handled by the data generator, with input from the collector.

The DFG to VHDL translation process, therefore, is divided into two 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. Then, the loop generator and collection nodes are implemented by selecting the proper VHDL components from a library, and are parameterized with values extracted from the appropriate DFG nodes. The interconnections between the ILB and generator/collection components are made by a top–level VHDL module, which is also generated by the translator.

The translation of the ILB involves a traversal of the dataflow graph. A VHDL *component* is created whose inputs are the outputs of the loop generator node, and whose outputs are the inputs to the data collection node. 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 more complicated operations, such as the SA-C reduction operations like *array sum*, the translator generates a connection to a VHDL component. A library of such components has been written directly in VHDL; this allows a SA-C program access to operators 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.

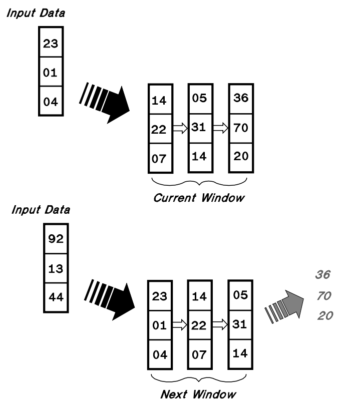

Figure 4. Window generator

The loop generator is responsible for presenting the proper data to the input of the ILB and providing the signals necessary to control the operation of the result buffers in the collector component. Figure 4 illustrates the operation of a 3×3 (2-D) window generator. Data read from memory is placed in a shift register - at each subsequent computation cycle, the oldest column of data is shifted out, the other rows are shifted “to the right,” and a new column is shifted in. This shift register provides a “sliding window” effect - at each cycle, the values in the shift register are presented to the inputs of the ILB.

The collector accepts the ILB outputs, buffers them into words, and then writes them into the result memory. These steps are controlled by the window generator - if more than one value is produced by the ILB, timing signals within the window generator insure that the collector has enough time to write the data before the next window of data is produced.

5: Optimizations and Pragmas

The SA-C compiler does a variety of optimizations, some traditional and some specifically designed to suit the language and its reconfigurable hardware targets. The compiler converts the entire SA-C program to an internal dataflow form called “Data Dependence and Control Flow” (DDCF) graphs, which it uses to perform optimizations. The traditional optimizations include Common Subexpression Elimination, Constant Folding, Invariant Code Motion, and Dead Code Elimination. It also does specialized variants of Loop Stripmining, Array Value Propagation, Loop Fusion, Loop Unrolling, Function Inlining, Lookup Tables, Array Blocking and Loop Nextification, as well as a loop and array Size Propagation Analysis. Some of these interact closely and are now described briefly.

Since SA-C targets FPGAs, the compiler does aggressive Full Loop Unrolling, which converts a loop to a non-iterative block of code more suitable for translating to a DFG. To help identify opportunities for unrolling, the compiler propagates array sizes through the DDCF graph, inferring sizes wherever possible. SA-C’s close association of arrays and loops makes this possible. Since the compiler converts only bottom-level loops to dataflow graphs, full loop unrolling can allow a higher-level loop to become a bottom-level loop, allowing it then to be converted to a DFG.

The SA-C compiler can do Loop Stripmining, which when followed by full loop unrolling produces the effect of multidimensional partial loop unrolling. For example, a stripmine pragma can be added to the median filter:

```
uint8 R[:,:] =
  // PRAGMA (stripmine (6,4))
  for window W[3,3] in A {
    uint8 med = array_median (W);
  } return (array (med));
```

This wraps the existing loop in a new loop with a 6x4 window generator. Loop Unrolling then replaces the inner loop with eight median code bodies. The resulting loop takes 6x4 sub-arrays and computes the eight 3x3 medians in parallel.

The SA-C compiler can fuse many loops that have a producer/consumer relationship. For example, the median filter might be followed by an edge detector, as shown here

```
uint8 R[:,:] = for window W[3,3] in A {
    uint8 med = array_median (W);
   } return (array (med));
uint8 S[:,:] = for window W[3,3] in R {
    uint8 pix = prewitt (W);
   } return (array (pix));
```

where prewitt is defined by the user as a separate function. The compiler will inline the function call, and fuse the loops into one new loop that runs a 5x5 window across A. (Note that a 5x5 is the size required to provide the needed elements to compute one value of S.) The new loop body will have nine median code bodies, and one prewitt code body. The goal of fusion is primarily to reduce data communication, both host/co-processor board and local memory/FPGA.

Loop Fusion as described above does redundant computation of medians. If FPGA space is plentiful, this is not a problem since the medians are computed in parallel, but if space is tight, Loop Nextification will remove the redundancies in the horizontal dimension by passing the computed medians from one iteration to the next. In the median filter-edge detector example, this reduces the loop body to three medians and one prewitt. If, after nextification, sufficient FPGA space is available, the fused loop can then be stripmined to reduce the number of iterations.

Lookup tables are often an attractive alternative to repeated computations, when the table size is feasible. SA-C allows a function to be given a pragma that tells the compiler to convert the specified function to a lookup table. The compiler computes all possible values of the function, building them into an array, and it converts all calls to the function to array lookups.

Though SA-C is a high-level language, it gives users control over the compilation process through the use of pragmas. The user can control Function Inlining, Loop Fusion, Loop Unrolling, Array Blocking, Stripmining and Lookup Table Conversion through the use of pragmas. In addition, the user can create a function prototype that is designated as an external VHDL plug-in; the SA-C compiler will pass calls to the designated function down through the DDCF and dataflow graphs, leaving "holes" that can be filled in at low level with a user's own VHDL routine.

5.1: Example: Probing

A common approach to automatic target recognition (ATR) is known as *probing*; it is a template-based technique for locating and distinguishing specific targets in an image. A *probe* is a pair of pixel locations spanning the edge of an object; the probe is "true" if the difference in pixel values exceeds a threshold. Typically, probes are arranged along the boundary of a target, and a set of probes defined for a single template is called a *probe set*. A target is detected by a template if the number of true probes exceeds a specific threshold.

A single probe set is useful for detecting a target only if the target appears at a fixed orientation and scale. To compensate for this, ATR probing systems generate a hemisphere of viewpoints around the target, and define a probe set for each view. A naive implementation of probing scans the image hundreds of times, once for each probe set.

In one such problem there are 365 probe sets. These sets are fixed; therefore they can be compiled as constants, thereby enabling several compiler optimizations. The code is rearranged such that a window traverses the image once. For each window location, the probes in *all* probe sets are computed, resulting in over 19,500 separate probes, each requiring a subtraction/threshold operation. However, these probe sets have many probes in common. The compiler's common subexpression elimination phase finds and removes the redundant operations, leaving only 5,500 operations. A new optimization further reduces the number of operations to only 379: it is a kind of common subexpression elimination that works across loop iterations, detecting situations where a value computed in one iteration is guaranteed to be the same as a value computed in a later iteration [5]. Special variables designed to transmit loop-carried dependencies can be used to pass the value across iterations, thereby eliminating the recomputation.

6: Compilation Path and Applications

Current day reconfigurable computing systems are typically based on FPGAs, which are large arrays of programmable logic cells, organized into one or more arrays of Configurable Logic Blocks (CLBs).

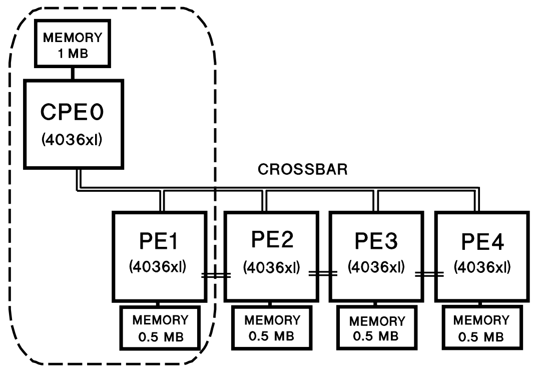

Figure 5. Architecture of the Wildforce-XL Reconfigurable Computing Board

One such system is the Wildforce-XL$^{(TM)}$, built by Annapolis Microsystems[2]. A simplified diagram of the board used as a target for the compilation system is shown in Figure 5 – it consists of five Xilinx 4036XL FPGAs; the PE's communicate via a 36 bit crossbar. Each PE has its own local memory. The board is connected to a host PC via the PCI bus. Our first implementation uses only that portion of the board inside the dotted lines – CPE0 retrieves image data from its local memory and sends it in the proper order over the crossbar to PE1, which buffers it (using the shift register scheme described earlier), presents it to the top of the ILB, and stores the results in its local memory.

The design is simplified by using two of the PE memories, thereby eliminating contention between reading and writing.

The Prewitt edge detection algorithm [12] is a very common operation in IP – it is used to identify edges in an image. While it is an important algorithm in its own right, it serves as an example here because it is typical of many common image processing operations. The algorithm creates a new array of values by performing the convolutions of a pair of fixed masks with every 3×3 sub-array within an image, then finding the magnitude of the vector formed by the two results. The SA-C program which performs the Prewitt calculation is shown in Figure 6.

```
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 ));
```

Figure 6. The Prewitt edge-detection algorithm, written in SA-C

The SA-C compiler performs several of the optimizations described earlier. 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 multiplications, 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 multiplication), 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 around 75% of the time required by the entire ILB.

The resulting DFG is processed by the DFG-to-VHDL translator, which extracts the ILB and translates it directly to VHDL, then 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, fully automate the entire process, from high level language compilation down to the production of FPGA configuration codes and the host-based control program. The user can execute the entire algorithm on the hardware like any other appli-

Code Entity	*Lines of Code*	*FPGA Usage CLBs*
SA-C Source Code	19	
VHDL Code - PE0		
`WINDOW-GEN`	572	251
VHDL Code - PE1		
`WIN-GEN/WRITE-VAL`	600	236
Inner Loop Body	3744	281
Glue Code & Misc	199	19
Total VHDL	5115	787

Execution Time (mSec)		
Image Size	300×198	665×699
Data Download	0.50	3.65
Computation	25.02	200.36
Result Upload	0.94	6.12
Total time	26.46	210.13

Computation Rate (MPixels/s)		
Computation	2.37	2.32
Comp + I/O	2.25	2.21

Table 1. Statistics for the Wildforce implementation of the Prewitt algorithm using the SA-C compiler/translator.

cation (by typing `a.out` or something similar), without needing to worry about any of the operational details of the hardware.

Table 1 shows some of the statistics of the entire compilation/translation process. A 19 line SA-C program eventually requires over 5000 lines of VHDL, and occupies approximately 30% of the CLBs in the two FPGAs used in the implementation.

While the main goal of the Cameron Project is to bring the ability to program reconfigurable hardware to the application programmer, a natural question concerns performance, either compared to manual (VHDL) implementation, or to a conventional processor, such as a Pentium. A single ILB Prewitt design results in an effective processing rate of around 1 MegaPixels/sec. Stripmining the design to 4×3, which effectively replicates the ILB twice, nearly doubles processing rate. A manual implementation on the Wildforce board, which includes a lookup table for the magnitude calculation, achieves a rate of 4.6 MP/sec. Adding a single pipeline stage in the ILBs increases this to 5.7. Current compilation research is studying ways to include lookup tables and pipelining in the ILB. We are optimistic that the results we can obtain with these efforts will be comparable with those we can achieve manually.

An implementation of Prewitt, written in C and compiled with optimizations, achieves a computation rate of 2.19 MP/sec on an 450MHz Pentium. We are encouraged by these results - implementing the ILB optimizations described above, coupled with execution on more modern FPGA technology (i.e., Virtex), should allow us to achieve around an order of magnitude of speedup over conventional processors.

7: Conclusions, Future and Related Work

The main thrust of the Cameron research project is to provide the ease of programming to reconfigurable systems that applications programmers are used to for conventional architectures. This has been achieved for typical IP applications. For example, the Prewitt algorithm was written, compiled and executed in a matter of hours. This is in contrast to the days to weeks of development time for VHDL programs. The usefulness and versatility of SA-C and its optimizing compiler has also been demonstrated by implementing a library of image processing routines, selected to be representative of the routines in the Vector, Signal and Image Processing Library[4] and the Intel Image Processing Library [8].

The optimizations currently available in the SA-C compiler have been shown to be highly effective for the kind of IP codes we have tested.

Future work includes the implementation of lookup tables and ILB pipelining, described earlier; this will reduce the propagation delay of the overall system, allowing for higher operating frequencies. Also, we are moving the system onto bigger and more complex (Xilinx Virtex [13]) FPGA boards, which allows higher clock frequencies and provides on-chip (RAM Block) memory. This memory can be used for buffering purposes and for implementation of lookup tables.

In the compiler area, the portion of the language that can be compiled to FPGAs is being extended. This entails loops with loop carried dependencies, and loop bodies that generate dynamically sized (e.g. strings) results.

Several other projects are researching reconfigurable computing. Several projects, including the Berkeley BRASS Project [3] and the MIT Raw Project [1], are working on systems that use C to produce codes for custom reconfigurable targets. Other projects are using common application oriented projects for their source program, including MATCH [11] (Matlab) and Champion [10] (Khoros). Another project, Defacto [6], uses the SUIF intermediate form as its input, allowing several high-level languages to be used to produce reconfigurable code.

References

[1] 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.

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

[3] T. J. Callahan and J Wawrzynek. Instruction level parallelism for reconfigurable coprocessor. In Hartenstein and Keevallik, editors, *Springer Lecture Notes in Computer Science Series, 1482*. Springer-Verlag, 1997.

[4] VSIPL Consortium. Vector signal image processing library forum, October 1997. www.vsipl.org.

[5] B. Draper, W. Najjar, W. Böhm, J. Hammes, R. Rinker, C. Ross, M. Chawathe, and J. Bins. Compiling and optimizing image processing algorithms for FPGA's. In *IEEE Workshop on Computer Architectures for Machine Perception (CAMP)*, Oct. 2000.

[6] M. Hall, P. Diniz, K. Bondalapati, H. Zeigler, P. Duncan, R. Jain, and J. Granacki. DEFACTO: A design environment for adaptive computing technology. In *Proc. 6th Reconfigurable Architectures Workshop (RAW'99)*, 1999.

[7] J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT'99*, Oct. 1999.

[8] Intel image processing library, v2.1. support.intel.com/support/performancetools/libraries.

[9] J. McGraw and et al. *SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2.* Lawrence Livermore National Laboratory, memo m-146 rev. 1 edition, 1985.

[10] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin. Automatic mapping of Khoros-based applications to adaptive computing systems. Technical report, University of Tennessee, 1999. Available from http://microsys6.engr.utk.edu:80/~bouldin/darpa/mapld2/mapld_paper.pdf.

[11] 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.

[12] J. M. S. Prewitt. Object enhancement and extraction. In B. S. Lipkin and A. Rosenfeld, editors, *Picture Processing and Psychopictorics.* Academic Press, New York, 1970.

[13] Xilinx, Inc. *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999. www.xilinx.com.