High-Level Language Abstraction for Reconfigurable Computing

Single-Assignment C is a C language variant designed to create an automated compilation path from an algorithmic programming language to an FPGA-based reconfigurable computing system.

Walid A. Najjar
University of California, Riverside

Wim Böhm
Bruce A. Draper
Jeff Hammes
Robert Rinker
J. Ross Beveridge
Monica Chawathe
Charles Ross
Colorado State University

One consequence of increasing VLSI system speed and circuit density is the growing viability of field programmable gate arrays as platforms for reconfigurable computing systems. RC systems typically consist of an array of configurable computing elements. The computational granularity of these elements ranges from simple gates—as abstracted by FPGA lookup tables—to complete arithmetic-logic units with or without registers. A rich programmable interconnect completes the array.

The pros and cons of fine- versus coarse-grained reconfiguration have been widely discussed.[1] Although several projects have investigated and successfully built systems in which the reconfiguration is done within a processor or among processors, we restrict our focus to the more common FPGA-based systems.

FPGA ADVANTAGES

Computing with FPGAs offers many advantages over traditional von Neumann processors. If the computation supports it, FPGA circuits can be massively parallel and customized to the task, whether this requires task, data, or pipeline parallelism, or a mixture thereof. FPGA circuits also can be specialized in terms of bit resolution: If a program needs to add a 9-bit value to a 12-bit value, it can construct a specialized adder for this task, rather than wasting circuitry with a standard 32-bit adder.

Implementing the program as a circuit with data streaming through it eliminates all the logic associated with instruction and address decoding. As a result, FPGAs can achieve speedups several orders of magnitude greater than that of traditional von Neumann processors,[1,2] as well as a significant reduction in energy consumed—an important consideration for mobile devices.[3] As VLSI technology increases in both density and speed, more interest in and opportunities for this technology will arise.

PROGRAMMING LIMITATIONS

Programmability provides the most daunting challenge developers face when using RC systems. In a typical design flow, the developer manually partitions an application into two segments: a hardware component that will execute as a circuit on the FPGA and a software component that will execute as a program on the host. The developer expresses the hardware component in a hardware description language such as VHDL or Verilog. The software component works as a driver, feeding input data to the FPGA and collecting output data.

This time-consuming approach to circuit design is a tedious process because complex algorithms must be expressed in terms of bits, registers, and clock signals. Moreover, because they typically lack circuit-design skills and knowledge of hardware description

0018-9162/03/$17.00 © 2003 IEEE

```
//convolution inner loop
result[:,:] =
    for window win[r,c] in Image
        {uint8 conv =
            for elem1 in win dot elem2 in Kernel
            return(sum(elem1*elem2));
        }
    return(array(conv));
```

Figure 1. Convolution example in SA-C. Irrespective of the kernel's and image arrays' element size, the result of each convolution will be cast to a `uint8` size.

languages, application programmers tend to find hardware design paradigms inaccessible.

SINGLE-ASSIGNMENT C

To overcome these problems, we developed a complete, automated compilation path from an algorithmic programming language to an FPGA-based reconfigurable computing system. The high-level language, a variant of C called *Single-Assignment C*, expresses image processing applications at a high level.

SA-C is amenable to efficient compilation into fine-grained parallel hardware systems because it hides the details and intricacies of low-level hardware design. At the same time, the SA-C compiler leverages extensive optimizations and code transformations to increase the speed and reduce the size of the resulting circuit.

Design challenges

Initially, we hesitated to design a new programming language. Most application developers, already trained to use C and C++, are understandably reluctant to learn a new language. Unfortunately, C is inextricably linked to the von Neumann processor model, in which variables correspond to memory locations and function invocations reside on stacks. For example, C lets users manipulate pointers to memory and to functions, which does not make sense in an FPGA circuit model. Thus any attempt to compile C to FPGA configurations would encounter problems that derive purely from the C language, not from the image-processing application.

At the same time, simply compiling a subset of C to FPGAs might not always be efficient because the disallowed operators would almost certainly be present in any existing C program.[4] This would then require rewriting existing programs for that C subset. For example, it would be difficult to write efficient code for processing images or other multidimensional arrays in C without pointer manipulation. Thus, developers would need to add new facilities to C to compensate for any disallowed operators. When developers work with both restricted and extended C, they essentially create a new programming language, or at least a significantly different dialect of C.

SA-C restrictions

As the name suggests, SA-C's most important restriction in comparison to C is that the value of any variable can be set only once, when the variable is declared. Many functional programming languages use this single-assignment restriction, which breaks the von Neumann equivalence between variables and memory locations. Since variables can be set only once, they correspond to values, not addresses, and can be assigned directly to wires. SA-C also removes the C dereferencing and address operators, thus eliminating pointers, and forbids recursion.

To compensate for these restrictions, SA-C introduces true multidimensional arrays, including arrays whose size is unknown at compile time. More significantly, it also introduces new versions of the *for loop* that exploit multidimensional arrays, allowing users to apply a loop body for every element, window, or dimension of an array. A reduction clause at the bottom of the loop body either collects values into a new array or applies a reduction operator to values it produces. Most of the code translated into FPGA configurations takes the form of these extended loops.

To take advantage of an FPGA's ability to create arbitrarily sized circuits, SA-C adds variable-bit-precision integers and variable-bit-precision fixed-point numbers to C. It also allows any function, loop, or conditional expression to return multiple values—a significant feature given the single-assignment restriction.

Compiler optimizations

The SA-C compiler supports a wide range of optimizations aimed at producing a more efficient hardware execution. Most of these optimizations seek to reduce the circuit's size, propagation delay, or I/O requirements. SA-C optimizations[5-7] include traditional methods such as constant folding, operator-strength reduction, function in-lining, dead-code elimination, invariant-code motion, and common subexpression elimination.

Other optimizations are specific to SA-C or adapted from vector and parallel compilers or synthesis tools. These include bit-width narrowing, loop unrolling, loop fusion, strip mining, and temporal common subexpression elimination.

Figure 1 shows an example of the SA-C code for a convolution with a window of size [r,c] of a fixed-array kernel over the array image. Both the kernel and image arrays would have been declared earlier in the program. The value `conv` is declared to be an unsigned eight-bit integer, which means that, irrespective of the kernel and image arrays' element size, the result of each convolution will be cast to a `uint8` size.

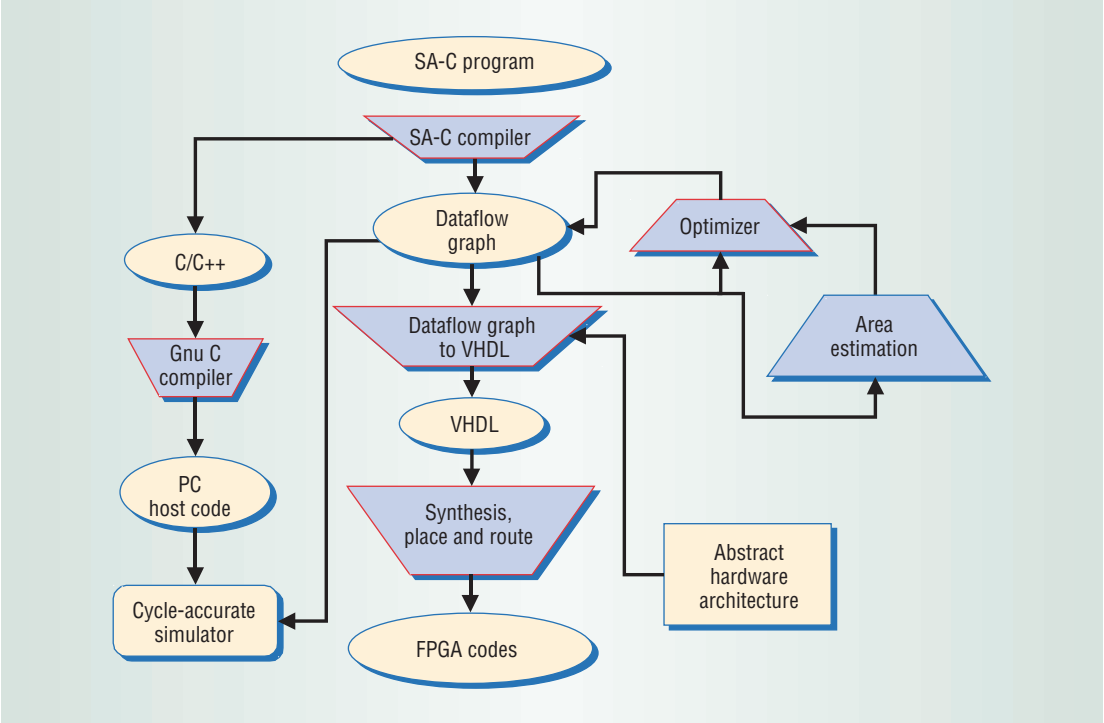

Figure 2. SA-C design flow. The SA-C compiler generates two executables: a host code that executes on the host PC and drives the execution on the FPGAs, and an FPGA configuration file.

Other languages

SA-C differs significantly from other efforts to map higher-level languages to hardware. Handel-C (www.celoxica.com) programs more closely resemble hardware than SA-C programs: The user must explicitly specify both timing and parallelism. The Ocapi (www.imec.be/ocapi) and SystemC (www.systemc.org) C++ extensions work with class libraries to create an application at a high level, then gradually migrate certain parts of the code toward a more explicit hardware description.

Perhaps the closest language to SA-C is Streams-C,[8] but while SA-C emphasizes loops and arrays, Streams-C emphasizes streams and processes. SA-C has also built on the experience of previous single-assignment languages such as Sisal (http://sisal.sourceforge.net).

SA-C advantages

SA-C was not conceived to be a stand-alone language. Rather, we assume that developers would rewrite selected loops and functions of existing C programs in SA-C and incorporate them in the original program. The SA-C compiler would then map these segments to hardware. Actually, SA-C does not support any file I/O operations because we assumed that such operations will be carried out in C.

SA-C also allows importing VHDL codes: The programmer can use a pragma to specify a function to be a VHDL code. The compiler will then insert this code and connect it properly to the rest of the circuit during the dataflow graph-to-VHDL translation phase. This feature lets the programmer use existing highly optimized VHDL codes with SA-C.

COMPILING SA-C TO HARDWARE

Figure 2 shows the overall SA-C design flow. SA-C programs compile to FPGA configurations and an accompanying C program that manages the FPGA's download of the configuration and data, triggering, and uploading of results. Thus, from an application developer's viewpoint, SA-C programs resemble any program running on a more traditional processor.

The compiler maps SA-C programs to executables, which the system invokes like any other program on the host. The program's execution speed provides the only clue that part of the program was actually mapped to a circuit and executed on a reconfigurable coprocessor.

As Figure 2 shows, the SA-C compiler generates two executables: a host code that executes on the host PC and drives the execution on the FPGAs, and an FPGA configuration file.

First, the compiler analyzes the SA-C source code to find parallel loops it can execute on an FPGA. It then translates all sequential code to C and includes it in the host program. Code targeted to the reconfigurable hardware then undergoes a series of representational transformations that bridge the gap between high-level source code and FPGA configurations.

Hierarchical dataflow graphs

The compiler then translates the SA-C loops into a *data-dependence and control-flow* graph. This hierarchical dataflow representation reflects the source program's structure in terms of function calls and nested loops. Once the compiler translates the program into a DDCF graph, it begins a series of

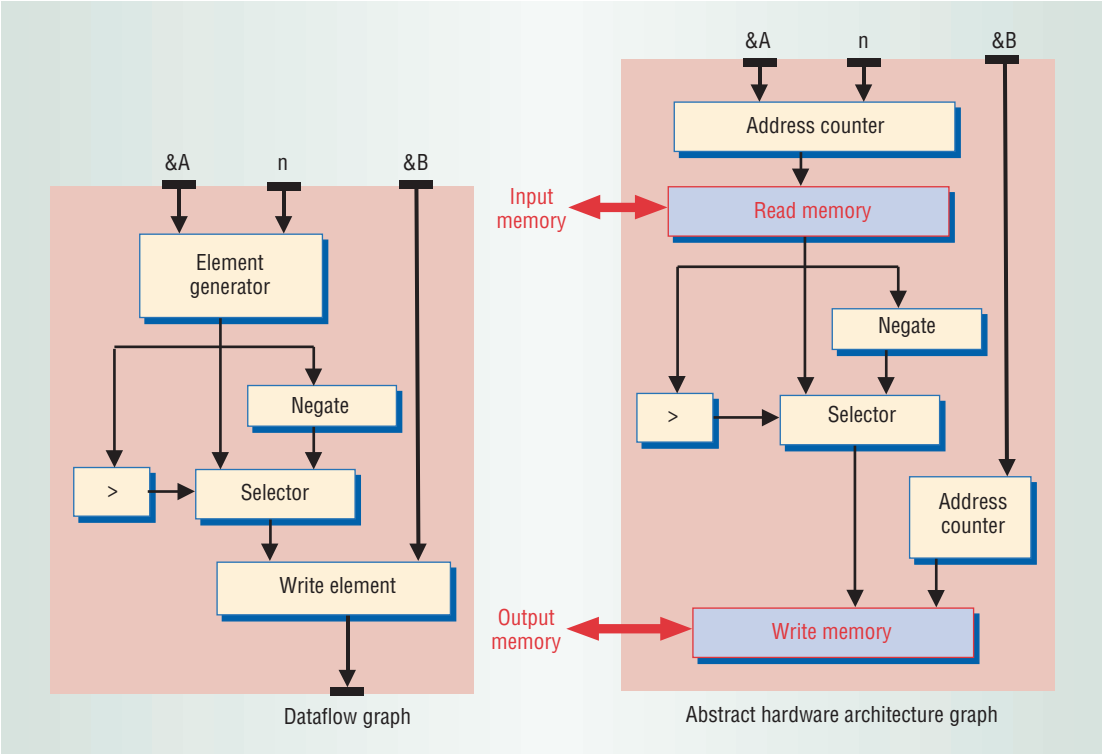

Figure 3. Dataflow and abstract hardware architecture graphs. &A and &B are the A and B arrays' starting addresses, n is the array size.

```
uint1[:,:] main(uint8 Image[:,:])
{
  // the horizontal and vertical masks
  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}};

  //Prewitt edge detection code
  int16 R[:,:] = for window W[3,3] in Image {
    int16 iph, int16 ipv = for h in H  dot  w in W  dot v in V
                           return(sum(h*w), sum(v*w));
    int16 SqrtSumSquare = sqrt(iph*iph + ipv*ipv);
  } return( array(SqrtSumSquare) );

  //Threshold code
  uint1 T[:,:] = for pix in R{ uint8 t = pix>127 ? 1 : 0;}
                 return(array(t));
} return(T);
```

(a)

```
uint8 T[:,:] = for window W[4,3] in Image step(2,1) {
//compute the left [3,3] window
int8 iph1 =  (W[0,2]+W[1,2]+W[2,2]) - (W[0,0]+W[1,0]+W[2,0]);
int8 ipv1 =  (W[2,0]+W[2,1]+W[2,2]) - (W[0,0]+W[0,1]+W[0,2])
uint8 mag1 = sqrt(iph1*iph1 + ipv1*ipv1);
uint8 t1 = mag1 >127 ? 255 : 0;

//compute the right [3,3] window
int8 iph2 =  (W[1,2]+W[2,2]+W[3,2]) - (W[1,0]+W[2,0]+W[3,0]);
int8 ipv2 =  (W[3,0]+W[3,1]+W[3,2]) - (W[1,0]+W[1,1]+W[1,2]);
uint8 mag2 = sqrt(iph2*iph2 + ipv2*ipv2);
uint8 t2 = mag2 >127 ? 255 : 0;
uint8 t[2,1] = {{t1},{t2}};

//return the result of both window computations as a [2,1] tile
                } return(tile(t) );
```

(b)

Figure 4. Prewitt edge detection followed by a threshold operator. (a) The original code, before optimization; (b) the same code, expressed as a fragment of SA-C code, after loop-unrolling, index propagation, loop fusion, and strip mining optimizations.

transformations aimed at optimizing these executions by increasing parallelism as well as decreasing the circuit size.

This process flattens the DDCF graph structure, eventually producing a nonhierarchical *dataflow graph*. DFGs are token-driven structures whose nodes correspond to operations and whose edges correspond to wires. As such, DFGs provide abstract circuit descriptions, without timing information or resource contention.

Loop fusion, one possible optimization in the DDCF-to-DFG translation, fuses two or more loops into a single loop body. When loops cannot be fused, a single DDCF graph will produce multiple DFGs, each of which leads to an independent FPGA configuration.

Architecture graphs

Once it has optimized the DFGs, the compiler translates them into *abstract hardware architecture* graphs, which differ from DFGs in that they include timing information. An AHA graph consists of several sections that resemble pipeline stages. Arbitrators in the AHA execution model control access to shared resources such as local memories.

After a final round of low-level machine-dependent optimizations, the compiler translates AHA graphs into VHDL. We use commercial tools—such as the Synplify synthesis tool from Synplicity (www.synplicity.com) and the Xilinx M1 place and route tool (www.xilinx.com)—to synthesize, place, and route the VHDL program.

A simple program that produces the absolute

value of every element in an array illustrates these transformations:

```
B[:,:] = for a in A
return(array(abs(a))
```

The program uses SA-C specialized *for loop* to iterate over all the elements in the array and collect absolute values in a new array. Figure 3 shows the DFG graph of the loop with the inlined abs function and the AHA graph of the same code. The DFG inputs represent the starting address of the input array A, the input array size, and the start address of the result array, respectively. Inputs to the AHA graph are the same as for the DFG.

COMPILER OPTIMIZATION EFFECTS

Figure 4 shows the effects of compiler optimizations expressed at the source-code level on a simple program with the Prewitt edge-detection operator, followed by a threshold. Figure 4a shows the original code, while Figure 4b shows the same code after optimization.

The Prewitt edge detector computes the convolution of every 3×3 window in an image with vertical and horizontal masks. Next, it computes the square root of the sum of the two results, squared. A threshold operator then creates a binary image, depending on whether the edge magnitude at every pixel is above or below a given value—in this case, 127.

The optimizing compiler unrolls the convolution's inner loop with the vertical and horizontal masks, folds the masks' constant values into the unrolled inner loop, propagates the constant indices, and fuses the Prewitt and threshold loops into a single loop. That loop is then strip-mined into two concurrent loop bodies.

Strip mining is a pragma-directed optimization in SA-C that reduces I/O. Without strip mining, the system reads every row of the image three times, except fringe rows, whereas in the strip-mined case the system reads every row only twice. Depending on the FPGA's capacity, strip mining can be increased, thereby further reducing I/O traffic. Strip mining also increases parallelism, because it computes two or more output values at the same time, thus creating additional opportunities for common subexpression elimination.

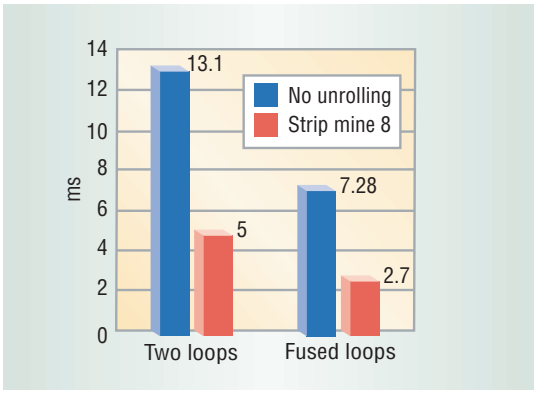

Figure 5. Execution time and speedup due to loop optimizations and their effects on circuit area and clock speed.

Figure 5 shows the results of these execution-time optimizations on a Xilinx Virtex 1000 FPGA. Table 1 shows the effect on chip area utilized and circuit frequency. The fused loops occupy a smaller area on the FPGA than the two separate loops. The clock speed is within the same range.

Most compiler optimizations have an impact on the area the resulting FPGA circuit uses. Because the FPGA does not have infinite resources, the compiler must be aware of these optimizations' area costs.

We have developed a compile-time area-estimation tool that generates an estimate of the circuit area in less than one millisecond.[9] The tool's average accuracy on a large number of codes is plus or minus 5 percent.

PERFORMANCE EXAMPLES

The implementation and performance of three code examples from linear algebra, image processing, and image compression provide examples of how SA-C works.

We ran the SA-C versions of these codes on the Annapolis Micro Systems' WildStar board (www.annapmicro.com). This board consists of three Virtex E 2000 FPGAs and 24 Mbytes of onboard SRAM memory. All the codes ran on a single FPGA. The C versions of these codes ran on an 800-MHz

Table 1. Experimental results with image-processing programs.

Code	FPGA clock (MHz)	SA-C execution time (ms)	PC execution time (ms)	Speedup	Comments
Wavelet	35.1	2.1	77.0	36.6	C++ code on PC
Canny	32.2	6.0	850.0	142.0	C code on PC
Canny MMX	32.2	6.0	135.0	22.5	MMX code on PC
Prewitt	42.1	1.9	158.0	83.0	MMX code on PC
AddS	51.7	0.67	5.95	8.8	MMX library call
Probing	41.1	80.00	65,000.00	800.0	C code on PC

Pentium III. Whenever feasible, the C implementation used the MMX extensions, in particular the Intel Image Processing Library (www.intel.com).

Wavelet image encoding

This program provides an implementation of the Cohen-Daubechies-Feauveau wavelet image-compression code.[10] It generates four images—each a quarter the size of the original—consisting of the original image as well as the image intensity differentials:

$$\frac{d^2I}{dx^2}, \frac{d^2I}{dy^2}, \text{ and } \frac{d^4I}{dx^2y^2}$$

We implemented the wavelet image-compression code in C on an Intel Pentium.

Canny edge detection

A commonly used edge-detection code, the Canny benchmarks is based on an algorithm[11] that consists of four steps:

1. *Image smoothing*. Convolution with a Gaussian mask smoothes the image.
2. *Edge magnitude and direction computation*. To compute edge magnitudes and edge directions, the algorithm uses two 5×1 edge masks. The masks give estimates of intensity surface *dx* and *dy*. The edge magnitude is the square root of (*dx*dx* + *dy*dy*). The edge direction $\pm p$ is coarsely quantized into four buckets of width $p/4$, based on the ratio of *dx* to *dy*.
3. *Directional edge suppression*. To avoid thick edges from long, slow gradients, the algorithm does nonmaximal suppression in the gradient's direction only. After suppression, the algorithm compares edges against two thresholds and classifies them as nonedge, low-edge, or high-edge.
4. *Connected-components algorithm*. A pixel is said to be an edge if, after nonmaximal suppression, it lies above the high threshold or above the low threshold and next to another edge pixel.

We implement the algorithm's first three steps, but not the fourth. We then compare the Canny code's runtime performance to two implementations on an Intel Pentium:

- a C++ version that, when feasible, uses the Intel Image Processing Library; and
- another Intel implementation in x86 assembly code.

Prewitt edge detection

The Prewitt edge-detection algorithm convolves every 3×3 window in an image with a horizontal and a vertical mask, as shown in Figure 4. The algorithm computes the edge magnitude by obtaining the square root of the sum of the squares of the convolution response. The SA-C implementation is compared to a C code that uses the Intel MMX library for convolution.

AddS MMX library

AddS is an MMX library that adds a scalar value *S* to each pixel in an image. It is a very small piece of code, hand-optimized for MMX execution.

Probing ATR algorithm

Probing, an *automatic target-recognition* algorithm, matches 243 templates representing three vehicles onto a $512 \times 1{,}024$ image of 12-bit pixels.[12] The algorithm applies each template—called a *probe set*—to every pixel in the image. On a PC, the program executes more than 60 billion instructions. The SA-C implementation executes 400 concurrent loop iterations on three Xilinx Virtex E 2000 FPGAs.

Performance evaluation of the high-level, algorithmic language SA-C for one-step compilation to host code and FPGA configuration codes has just begun. We are porting the system to a more complex board that contains three FPGAs. As performance issues become clearer, the system will be given greater ability to evaluate various metrics, including code space, memory use, and time performance, and to evaluate tradeoffs between conventional functional code and lookup tables. ■

Acknowledgment

This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319 and by National Science Foundation grant ITR 0083080.

References

1. W.H. Mangione-Smith, "Seeking Solutions in Configurable Computing," *Computer*, Dec. 1997, pp. 38-43.
2. A. DeHon, "The Density Advantage of Configurable Computing," *Computer*, Apr. 2000, pp. 41-49.
3. J. Villarreal et al., "Improving Software Performance with Configurable Logic," *J. Design Automation of Embedded Systems*, Nov. 2002, pp. 325-339.

4. Y. Li et al., "Hardware-Software Co-Design of Embedded Reconfigurable Architectures," *Proc. Design Automation Conf.* (DAC 00), ACM Press, 2000, pp. 507-512.
5. A.P.W. Böhm et al., "One-Step Compilation of Image Processing Algorithms to FPGAs," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 2001), IEEE CS Press, in press.
6. W. Böhm et al., "Mapping a Single Assignment Programming Language to Reconfigurable Systems," *J. Supercomputing*, vol. 21, no. 2, 2002, pp. 117-130.
7. R. Rinker et al., "An Automated Process for Compiling Dataflow Graphs into Hardware," *IEEE Trans. VLSI*, Feb. 2001, pp. 130-139.
8. M.B. Gokhale et al., "Stream-Oriented FPGA Computing in the Streams-C High Level Language," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 2000), IEEE CS Press, 2000, pp. 49-56.
9. D. Kulkarni et al., "Fast Area Estimation to Support Compiler Optimizations in FPGA-Based Reconfigurable Systems," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 2002), IEEE CS Press, 2002, pp. 239-247.
10. A. Cohen, I. Daubechies, and J.C. Feauveau, "Bi-Orthogonal Bases of Compactly Supported Wavelets," *Comm. Pure and Applied Mathematics*, vol. 45, 1992, pp. 485-560.
11. E. Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision*, Prentice-Hall, 1998.
12. W. Böhm et al., "Compiling ATR Probing Codes for Execution on FPGA Hardware," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 2002), IEEE CS Press, 2002, pp. 301-302.

Walid A. Najjar *is a professor in the Department of Computer Science and Engineering at the University of California, Riverside. His research interests include computer architectures, reconfigurable and embedded systems, parallel computing systems, and interconnection networks. Najjar received a PhD in computer engineering from the University of Southern California. Contact him at najjar@cs.ucr.edu.*

Wim Böhm *is a professor in the Department of Computer Science at Colorado State University. His research interests focus on the design and implementation of high-level languages for specific application domains such as image processing to execute on special-purpose hardware or reconfigurable computing systems. Böhm received a PhD in computer science from the University of Utrecht, Holland. Contact him at bohm@cs.colostate.edu.*

Bruce A. Draper *is an associate professor in the Department of Computer Science at Colorado State University. His research interests focus on object recognition, including biological vision, color vision, and the control of computer vision. Draper received a PhD in computer science from the University of Massachusetts, Amherst. He is a member of the IEEE. Contact him at draper@cs.colostate.edu.*

Jeff Hammes *is a compiler developer at SRC Computers in Colorado Springs, Colorado. His research interests include data flow, discrete event simulation, and efficient techniques for gather-scatter. Hammes received a PhD in computer science from Colorado State University. Contact him at hammes@cs.colostate.edu.*

Robert Rinker *is an associate professor in the Department of Computer Science at the University of Idaho and a PhD candidate at Colorado State University. His research interests include computer architecture and reconfigurable computing. Rinker is a member of the IEEE Computer Society. Contact him at rinker@cs.uidaho.edu.*

J. Ross Beveridge *is an associate professor in the Department of Computer Science at Colorado State University. His research interests include the evaluation of human identification algorithms, object recognition through iterative scene rendering, geometric matching, combinatorial optimization, and sensor fusion. Beveridge received a PhD in computer science from the University of Massachusetts. He is a member of the IEEE Computer Society. Contact him at ross@cs.colostate.edu.*

Monica Chawathe *is a doctoral candidate in the Department of Computer Science at Colorado State University. Her research interests include stream-oriented programming languages, compilation, and parallel algorithms. She received an MS in computer science from Colorado State University. Contact her at monica@cs.colostate.edu.*

Charles Ross *is a doctoral candidate in the Department of Computer Science at Colorado State University. His research interests include reconfigurable computing, hardware synthesis, AI, computer vision, and graphics. He received an MS in computer science from Colorado State University. Contact him at rossc@cs.colostate.edu.*