A High Level, Algorithmic Programming Language and Compiler for Reconfigurable Systems *

Jeffrey P Hammes, Robert Rinker, Wim Böhm, Walid A. Najjar, Bruce Draper
Computer Science Department
Colorado State University
Ft. Collins, CO, U.S.A.

Abstract ***This paper presents a high level, machine independent, algorithmic, single-assignment programming language SA-C and its optimizing compiler targeting reconfigurable systems, and intended for Image Processing applications. Language features are introduced and discussed. The intermediate forms DDCF and DFG, used in the optimization and code-generation phases are described. Conventional and reconfigurable system specific optimizations are briefly introduced. The code generation process, using an abstract target machine, is described. Finally the performance effects of combinations of various optimizations are compared to hand coded C, using an edge detection algorithm followed by a threshold operator. Timing results are encouraging. Improvements of the compilation and code generation route are discussed.***

Keywords: Reconfigurable Computing Systems, FPGA, Image Processing, High Level Languages, Optimizing Compilation

1 Introduction

This paper presents an algorithmic programming language, SA-C [6] (derived from "single-assignment C, and pronounced "sassy") and its optimizing compiler targeting reconfigurable systems. SA-C has been initially designed for Image Processing applications, while being amenable to efficient compilation to fine grain parallel hardware systems.

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

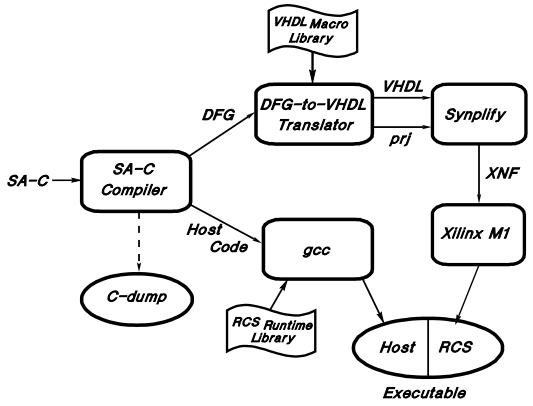

Figure 1. System overview

Currently, 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 applications programming. Applications programmers, who want to exploit the potential of these reconfigurable systems, are discouraged by the difficulty of implementing algorithms in circuit design languages, and by the amount of hardware specific knowledge needed to use these systems.

Mapping an *algorithmic programming language* to reconfigurable hardware brings new challenges to language designers and compiler writers. The language must allow easy extraction of fine grain parallelism as well as aggressive optimization, both in terms of code space and execution time. The Cameron Project [7, 10] provides such a high level, algo-

rithmic language, SA-C, and optimizing compiler for the development of image processing algorithms on reconfigurable computing systems, see Figure 1.

2 The SA-C Language

The design goals of the SA-C language are

- high-level, algorithmic language
- single-assignment, for better compiler analysis and translation to DFGs
- no pointers or side effects, for better compiler analysis
- emphasis on loops and arrays
- high-level operators for IP applications
- operator syntax and precedences as in C
- variable bit-width data types
- user control of optimizations

The emphasis on loops and arrays accommodates the efficient expression of low and medium level Image Processing algorithms, which form the FPGA target application domain of the Cameron Project. As SA-C does not allow dynamic data structures and recursion, more irregular computations, e.g. involving trees and graphs, are not easily expressed.

SA-C differs significantly from other efforts to map higher level languages to FPGAs. Handel-C [1] programs are closer to hardware than SA-C programs. For example, timing is explicit in Handel-C. The parallelism in Handel-C is also more explicit: the user must declare processes and interconnecting channels. Ocapi [9] and SystemC [12] are C++ extensions that allow the user, through the use of class libraries, to begin creating an application at a high level and gradually migrate certain parts of the code toward a more explicit hardware description. By the time the user is down at the hardware level, the languages are in effect hardware description languages, but with a more familiar look. There is no emphasis on aggressive automatic compiler optimizations as in SA-C. Another project using C++ is Streams-C [3]. The language model has processes and streams, and the compiler uses the SUIF infrastructure.

Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. Since SA-C is a single-assignment language, each variable's declaration occurs together with the expression giving it its value. (This approach prevents semantically unpleasant "dynamic single-assignment" situations such as declaring a variable in an outer code block and potentially assigning to it in only one part of a conditional.) SA-C has multidimensional rectangular arrays whose extents are determined dynamically or statically. The type declaration **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. 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 subarrays 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. In this example, if array A had a shape of [100,200], the result array R

would have a shape of [98,198]. 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.

SA-C **for** loops may have "nextified" variables, a mechanism borrowed from other loop-oriented functional languages to allow loop-carried dependencies to be expressed. In the absence of nextified variables, a SA-C loop is fully parallel, and this loop-level parallelism can be exploited by the compiler in various ways. The presence of a nextified variable imposes an execution order on the loop. In SA-C this order is a row-major traversal of each array accessed by the loop's generators.

A SA-C program compiles to a host machine executable that has calls to a reconfigurable coprocessor board. The system can also compile the entire program to a host executable for efficient program debugging. When compiling calls to reconfigurable hardware, it transforms bottom-level loops into dataflow graphs (DFGs) [8], suitable for mapping onto FPGAs. The host code includes interface code that automatically downloads FPGA configurations and source data, and uploads the results for further computation on the host.

The SA-C compiler attempts to translate every bottom-level loop (i.e., a loop that contains no loop) to a dataflow graph (DFG), a low-level, non-hierarchical and asynchronous program representation that will be mapped for execution on reconfigurable hardware. (In the present system, not all loops can be translated to DFGs. The most important limitation is the requirement that the sizes of a loop's window generators be statically known.) DFGs can be viewed as abstract hardware circuit diagrams without timing considerations taken into account. Nodes are operators and edges are data paths. The dataflow graphs are designed to allow token driven simulation, used by the compiler writer and applications programmer for validation and debugging. There are four general classes of node types that can appear in a dataflow graph:

- arithmetic
- low level control (e.g. selective merge)
- data extraction and routing nodes that reflect the generators that drive a loop
- data collection nodes that accumulate a loop's return values

3 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 [5], which it uses to perform all optimizations [4]. The traditional optimizations include Common Subexpression Elimination, Constant Folding, Invariant Code Motion, and Dead Code Elimination. The compiler also does specialized variants of Loop Stripmining, Array Value Propagation, Loop Fusion, Loop Unrolling, Function Inlining, Lookup Tables and Array Blocking, along with 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 on parallel **for** loops, which when followed by full loop unrolling produces the effect of multi-dimensional 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 subarrays and computes the eight 3x3 medians in parallel. Since the new loop has non-unit strides, there are fewer loop iterations.

The SA-C compiler can fuse many loops that have a producer/consumer relationship. For example, a Prewitt edge detector [11] might be followed by a threshold operator, as shown here

```
int3 vert[3,3] =
 {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
int3 horz[3,3] =
 {{-1,-1,-1}, { 0, 0, 0}, { 1, 1, 1}};

uint8 R0[:,:] =
 for window win[3,3] in image {
   int11 v =
     for elem1 in win dot elem2 in vert
     return(sum((int11)elem1*elem2));
   int11 h =
     for elem3 in win dot elem4 in horz
     return(sum((int11)elem3*elem4));
   int22 sqv = (int22)v*v;
   int22 sqh = (int22)h*h;
   uint8 mag = sqrt((int23)sqv+sqh);
 } return (array (mag));

uint8 R1[:,:] =
  for pix in R0 {
    uint8 v = pix>127 ? 255 : 0;
  } return (array (v));
```

where the bit-widths and casts, which behave the same as in C, cause the operations to be done correctly using the least number of bits. The compiler will fuse the loops into one new loop, eliminating the intermediate array R0. The SA-C compiler also is able to fuse a loop pair in which both loops have window generators, and is able to fuse a pipeline of such loops. The goal of fusion is primarily to reduce both host/coprocessor-board and local-memory/FPGA data communication. The user can prevent fusion by placing a **no_fuse** pragma on the producer loop.

Lookup tables are often an attractive alternative to complex computations. 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. This is feasible in SA-C, as the language allows small bit-width data types.

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.

4 Low-level implementation

Unlike standard processors, which provide a relatively small set of well-defined instructions to the user, reconfigurable computing systems are composed of an amorphous mass of logic cells, which can be interconnected in a countless number of ways. To impose structure on the compilation process, an *abstract machine* has been defined, shown in Figure 2. The DFG for a SA-C program consists of a loop generator, an inner loop body (ILB), and a data collector. The loop generator reads data from local memory and presents it to the ILB. Values that are calculated by the ILB are then collected before being written to memory. The ILB is combinational; all timing and control of the computation process is handled by the loop generator and data collector.

The implementation of a window generator uses shift registers. In each loop iteration the oldest column of data is shifted out and a new

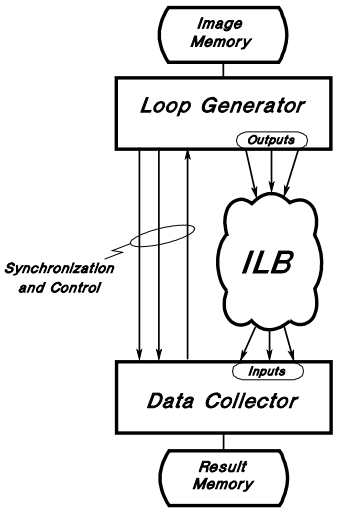

Figure 2. Abstract machine structure

	unfused		fused	fused,
	prew	thresh		strip
data download (msec)	0.54	0.51	0.54	0.54
execute (msec)	59.14	11.34	26.95	17.86
data upload (msec)	0.93	0.91	0.93	0.94
total time (msec)	60.61	12.76	28.42	19.34
freq (Mhz)	2.48	10.35	5.45	4.53
execute (K cycles)	146.9	117.8	147.0	80.9

Table 1. Performance of loop fusion and stripmining.

column is shifted in. Words are read from the FPGA's local memory as needed. The collector accepts the ILB outputs, buffers them into words, and 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 Performance

This section discusses the effects of some of the SA-C compiler's optimizations, using the Prewitt/threshold loop pair shown in Section 3, run on a Wildforce$^{(TM)}$ board from Annapolis Microsystems[2]. The code is compiled in three ways: unfused (two separate loops, with two separate data round trips), fused, and fused with 4x3 stripmining. Each loop execution on the reconfigurable system requires the downloading of data, running of the loop, and uploading of the result. The downloads and uploads are done using DMA on the PCI bus. The loop execution takes one board cycle per iteration, regardless of the loop body size. However, the clock frequency varies from loop to loop, and is determined by the critical path length, i.e. propagation delay, of the circuit that is derived from the place-and-route step that creates the configuration file for the FPGA. Clock frequency on the Wildforce board also affects the speed of memory transactions between the FPGA and its local memory, since there is one memory access per clock cycle.

Table 1 shows the resulting times, as well as the board frequencies and number of execution cycles, run on an image of 198 by 300 pixels. The unfused loop pair takes two data round trips, and two loop executions. The Prewitt loop runs at only 2.5 MHz due to the long path length of the square root computation. The loop pair takes 73.37 msec (the sum of the two individual loop executions). When the loops are fused, the total execution time drops to 28.42 msec. The performance increase is due to a number of factors: First, the total number of iterations is cut in half. Second, the clock frequency is higher than that of the Prewitt alone[1]. Third, there is one data round trip rather than two, though the DMA transfer

[1]The rise in frequency is probably due to the fact that the threshold is comparing with 127. This requires looking at only the high-order bit, and allows the low-level FPGA mapping software to eliminate parts of the square root computation, thus reducing the critical path length.

times are so fast that this makes a very small difference.

Stripmining improves the time further, to 19.34 msec. This gain is primarily due to a reduction of FPGA reads from its local memory: when stripmined to a 4x3 window, the window has a vertical stride of two, so each data row is read twice instead of three times. There is also some improvement due to a further reduction in number of iterations, which saves some loop overhead. Studies in the Cameron group have shown that deep stripmining is able to give significant improvements in performance. Unfortunately the FPGAs on the WildForce board are small, and deeper stripmining of this example was not possible because of space constraints.

5.1 SA-C vs. a Hand-Coded VHDL Example

During development of the abstract model, several algorithms were manually-coded in VHDL. These hand-coded examples provided considerable insight into the issues that the compiler system must address in producing efficient hardware codes, and motivated many of the optimizations incorporated into the compiler. The performance of these manual implementations, which are optimized to use specific hardware resources, serve as a benchmark with which to judge the performance of the automated system.

Table 2 shows the execution times for a manually coded Prewitt design. It contains 2 ILB's, comparable to the 4×3 stripmined version described above. It differs in that it uses a lookup table for the magnitude computation (two multiplies and a square root) that accounts for most of the propagation delay of the loop body. As a result, the manual version is able to run more than two times faster than the automated one (10.1MHz vs. 4.53MHz). A second version of the manual design, which adds one stage of pipelining to the ILB, reduces the propagation delay by another 70%, allowing execution at nearly four times the speed of the automated version. Both lookup tables and pipelining are being added to the compilation system, and should yield results that are similar to the manual version.

	Hand-Coded Prewitt	
	Non-pipelined	Pipelined
execute (msec)	7.15	4.66
freq (Mhz)	10.1	17.0

Table 2. Performance of hand-coded Prewitt

5.2 SA-C Performance vs. C

Performance comparisons across platforms and languages are always difficult and sometimes contentious, but it is nevertheless useful to put these execution times into some kind of perspective. The inner loop of a separately developed C program for the Prewitt-threshold example, hand-fused, is shown here.

```
for (i=0; i<ysz-2; i++)
 for (j=0; j<xsz-2; j++){
  pt = image + i*xsz + j;
  r0 = (int)pt[0] + pt[1] + pt[2];
  r2 = (int)pt[0+2*xsz] + pt[1+2*xsz]
        + pt[2+2*xsz];

  c0 = (int)pt[0] + pt[xsz]
        + pt[2*xsz];
  c2 = (int)pt[2] + pt[2+xsz]
        + pt[2+2*xsz];

  rsm = r0-r2;
  csm = c2-c0;
  mag = sqrt ((double)rsm*rsm + csm*csm);

  res = mag>127 ? 1 : 0;

  *(rpt++) = res;
 }
```

This code was compiled using *gcc -O6* and executed on both 266 MHz and 800 MHz Pentium based machines, yielding execution times of 42.6 msec and 14.0 msec, respectively. We are encouraged by the comparison of these results with the SA-C code's performance, especially since the FPGA technology being used in this comparison is a number of years old.

6 Conclusions and Future Work

The main thrust of the Cameron research project is to provide to applications programmers the ease of programming for reconfigurable systems that they have enjoyed for conventional architectures. This has already been achieved for that subset of programs that the compiler can currently map to FPGAs. The three versions of the Prewitt/threshold code differ only in one or two pragma lines, and were written, compiled and executed in a matter of hours. This contrasts to the days, sometimes weeks, of development time required for VHDL programs.

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. In the above example, the original inner loop body was fully unrolled and array constants were propagated and folded. This gave rise to the unfused version. Loop fusion enlarged the loop body run on the FPGA, and reduced costly FPGA to local memory access, bringing the execution time down from 73.37 msec to 28.42 msec for an image of 198 by 300 pixels. This was further reduced to 19.34 msec by loop stripmining.

Work is currently underway to port the system to target boards containing Virtex [13] FPGAs. The order-of-magnitude space increase of this chip over the current FPGAs will allow deeper stripmining as well as fusion of longer loop pipelines. The chip's RAM Block memory will be used for lookup tables. Its higher clock frequencies, along with pipelining of the inner loop body, will allow significantly faster execution speeds.

In the compiler, work is currently underway to implement some novel optimizations designed to save space in the FPGAs. One program transformation will perform a kind of common subexpression elimination across loop iterations. Another will reduce window sizes by moving subexpressions across iterations. The saved space will allow still more aggressive application of the optimizations that have already been developed.

References

[1] OXFORD hardware compiler group, the Handel Language. Technical report, Oxford University, 1997.

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

[3] M. Gokhale. The Streams-C language. www.darpa.mil/ito/psum1999/F282-0.html.

[4] J. Hammes. *Compiling SA-C to Reconfigurable Computing Systems.* PhD thesis, Colorado State University, 2000.

[5] J. Hammes and W. Böhm. *The SA-C Compiler DDCF Graph Description*, 1999. Document available from www.cs.colostate.edu/cameron.

[6] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*, 1999. Document available from www.cs.colostate.edu/cameron.

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

[8] J. Hammes, R. Rinker, D. McClure, W. Böhm, and W. Najjar. *The SA-C Compiler Dataflow Description*, 1999. Document available from www.cs.colostate.edu/cameron.

[9] IMEC. Ocapi overview. www.imec.be/ocapi/.

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

[11] 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.

[12] SystemC. SystemC homepage. www.systemc.org/.

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