Programmability and Efficiency in Reconfigurable Computer Systems*

Zhi Guo
Electrical Engineering

Dinesh Chander Suresh
Computer Science & Engineering

Walid A. Najjar
Computer Science & Engineering

University of California, Riverside
Riverside, California, 92521
{zguo, dinesh, najjar}@cs.ucr.edu

ABSTRACT
It has become commonly accepted that higher abstraction programming languages are necessary for a wider acceptance of reconfigurable computing technology by application developers. Anytime the abstraction level is increased, a tradeoff must be made between programmability and efficiency. This paper reports on the quantitative evaluation of this tradeoff using the SA-C language and VHDL. It relies on four benchmark codes ranging from the trivially simple to the complex. The results show a slowdown in execution by a factor less than two an increase in programmability by a factor as large as 10.

Keywords
Adaptive computing, configurable, reconfigurable components, reconfigurable computing, reconfigurable systems. FPGA, High level languages.

1. Introduction

Today, the high density and speed of field programmable gate arrays (FPGAs) make it possible to achieve high-speed, massively parallel, reconfigurable computation, typically making use of low level hardware description languages (HDLs) such as VHDL or Verilog. However, application programmers, such as computer vision or image processing researchers, are usually not familiar with HDLs and circuit design. It is imperative therefore that higher programming abstractions be developed that would allow application programmers to take advantage of the increased densities and speed of FPGAs. A number of new tools have been created to generate synthesizable VHDL or Verilog code from high level languages such as C: Streams-C compiler [5] generates RTL VHDL for a target FPGA board from parallel C programs. Other techniques [7] use MATLAB code as input notation and map the application to a distributed computing environment. The Cameron Project [2] has created such a tool named SA-C, a high level, single-assignment language with C-like syntax. The SA-C compiler targets FPGAs and allows programmers to write algorithms for them in a high level language.

One issue that often comes to the forefront in discussing these higher abstraction levels is that of efficiency: How efficient is the code being generated compared to hand-crafted VHDL or Verilog codes? This topic was addressed in [6] in the context of StreamsC. This paper describes a very similar evaluation using SA-C. The main justification in repeating this evaluation is that SA-C is based on more abstract programming paradigm than StreamsC. It is functional in nature and could therefore suffer from more inefficiencies in its implementation. As it turns out, the results are very similar to those observed for StreamsC.

In this paper we briefly introduce SA-C language and the corresponding hardware system (Section 2). Four programs, of increasing complexity, are implemented in SA-C and in VHDL and their performance and mapping parameters compared (Section 3). Based on these comparisons, we draw conclusions about the relative benefits and penalties to be derived from using SA-C.

2. SA-C: A Language for Reconfigurable Computing

The high-level language SA-C is a variant of C, and has been designed to express Image Processing (IP) applications at a high level, while being amenable to efficient compilation to fine grained parallel hardware systems. One of the main advantages of SA-C is that

* This work was supported in part by NSF Award ITR 0083080.

it hides the details and intricacies of low-level hardware design from the application programmer. 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.

The overall SA-C design flow is shown in Figure 1. SA-C programs are compiled to FPGA configurations (via Data Flow Graphs), plus a C program that manages the FPGA in terms of downloading the configuration and data, triggering the FPGA, and uploading the results. Thus, from the point of view of an application developer, SA-C programs are like any program running on a more traditional processor. The compiler maps SA-C programs to executables, which are invoked like any other program on the host. The only indication that part of the program was actually mapped to a circuit and executed on a reconfigurable co-processor is its speed of execution.

SA-C was not designed to be a stand-alone language. It does not support file I/O or any form of OS services invocation. Instead, it is intended to be included within a C/C++ program where only those functions or loops that are potential candidates for being mapped to hardware are expressed in SA-C.

As the name suggests, the most important restriction of SA-C (single assignment C) in comparison to C is that the value of any variable can be set only once, when the variable is declared. This single assignment restriction is found in many functional programming languages, and has the property that it 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 does away with the C de-referencing and address operators (* and &), thus eliminating pointers, and also forbids recursion. The SA-C language, compiler and compiler optimizations are described in [3] and [4].

A reconfigurable computing system usually combines one or more FPGA chips with local memory chips and a bus to the host. One such is the Annapolis Microsystems WILDSTAR/PCI/VME board [8], which we use as our platform. The board has three Xilinx Virtex XCV2000E processing elements (PEs), synchronous SRAM as local memory, and connects with the host by PCI bus. Standard VHDL modules can be used to design the interfaces to access and control the on-board components: For instance, the Clock Standard Interface provides the functionality to configure a delay locked loop (DLL), while the on-board memory is accessed through the Memory Standard Interface.

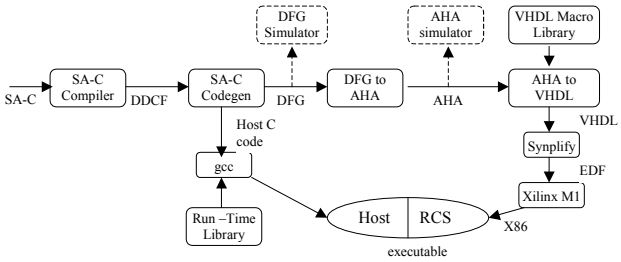

Figure 1 SA-C compilation system

Any SA-C produced or handwritten VHDL process has the same execution procedure: First of all, the host executive stores input data into the on-board memory, and then writes values to FPGA registers indicating the addresses and sizes of input and output data, triggering the calculation. After the computation, the host retrieves the output data from another part of the on-board memory.

3. Performance Comparisons

Four applications of increasing complexity were developed in both SA-C and handwritten VHDL, and their performance compared in terms of device area utilized, clock frequency achieved and execution time. Since one of the most significant aspects of SA-C is its treatment of FOR loops, our test applications focus on loops: In every case, we use the same algorithm and do the same loop optimizations to make sure the comparison is reasonable.

Finally, we compare the productivity and some other aspects of these two approaches.

3.1 A Simple Algorithm

First, we take as test cases two very simple operators, addition and multiplication. We take every pixel in an image and multiply it by, or add to it, a constant. Both SA-C and the handwritten version produce only several lines of VHDL for these processes, which are synthesized to an adder or multiplier. For SA-C code, the device utilization percentage (i.e. the fraction of the device area actually utilized in the implementation) is 15%, while for handwritten VHDL it is 4%. This difference is largely due to the fact that SA-C uses a standard pattern of hardware modules in its AHA (Abstract Hardware Architecture) VHDL

module library, even though in this case the operator itself is very simple. For example, the READ unit has its own circular buffer, which is unnecessary for a one clock-cycle operation. The main limitation on speed comes from the standard interfaces, which are the same for both SA-C and handwritten VHDL. The maximum clock frequency is about 66MHz. In terms of throughput, both designs deliver a result every clock cycle.

We wrote SA-C code to do the following computation:

$$[(Array\_a + Const\_a) * (Array\_b + Const\_b)] \quad (1)$$

The compiler parallelizes the two addition operators in two parentheses and pipelines the multiplication and addition. Again, the standard interfaces account for the performance bottleneck.

3.2 A Medium Complexity Algorithm

Figure 2 shows the SA-C code for finding the maximum of a 3×3 window sliding through an image array.

```
uint8 [:,:] main(uint8  indata[:,:])
{
 uint8 res[:,:] =
          //PRAGMA (stripmine(6,4))
          for window W[3,3] in indata {
          uint8 m =
                    array_max(W);
          }
          return(array(m));
}return(res);
```

Figure 2 SA-C code for finding max

The Stripmine PRAGMA causes a stripmining optimization, and encloses another loop that reads in chunks of data. This optimization has the effect of partial unrolling of the loop in multiple dimensions. The parameters indicate the size of the window in the new outer loop — in this example; the resized new window has 6 rows. Therefore, four array_max array operations are carried out simultaneously in the vertical direction.

Handwritten VHDL also calculates the maxima of four at the same time. The comparison results on a 256×256 8-bit image are shown in Table 1.

Table 1 Comparison of max filter

	Area	Clock Cycles	Clock Frequency (MHz)	Execution time (us)
VHDL	14%	16384	41	405
SA-C	16%	24471	35	709

The SA-C device utilization does not increase very much with respect to the previous example, while that of the handwritten code does: The handwritten code has to employ more modules to control the window sliding, while the SA-C code may use the same modules as in the previous simple case. Another reason for the improved relative utilization is that array_max is one of SA-C's built-in array operators. The compiler extracts window size information and generates the corresponding efficient VHDL.

However, the hand written version uses considerably fewer clock cycles: The 3×3 window slides through the image array, so a single pixel value is referenced in 9 windows. Though the SA-C compiler does parallelism and stripmine optimizations, the two bottom rows of data in each resized window still have to be fetched in the subsequent iteration. The handwritten design stores these values into the on-chip dual-port RAMS. In each iteration, the circuit reads new data from off-chip memory, and reads previous rows of data from on-chip dual-port memory simultaneously, while new data are fed into computing units and written into the dual-port memory.

A second reason for the difference is the form of loop control: Handwritten code employs state machines to manipulate the cooperation between I/O units and computing units. The circuit delivers a result every clock cycle until the reading unit reaches the end of the data in off-chip memory. Instead, the SA-C generated VHDL modules communicate with one another using a number of control signals. Every time the reading unit reaches the last data of a column, the circuit needs extra clock cycles to return to the head of next column.

We will analyze these two reasons in detail in the following subsection.

3.3 Two Complex Algorithms

Figure 3 shows SA-C code for the Prewitt edge detector [Prewitt, 1970]. It is one of the very common edge detection operators used in image processing operations, such as image sharpening.

```
uint8[:,:] main (uint8 image[:,:])
{
 // ***** defines the Prewitt masks *****
 int3 vertmask[3,3] = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
 int3 horzmask[3,3] = {{-1,-1,-1}, { 0, 0, 0}, { 1, 1, 1}};

 // ***** computes the gradient for the modified image  *****
 uint8 res[:,:] =
   //PRAGMA (stripmine (6,4))
   for window win[3,3] in image
    { int11 vert =  for elem1 in win dot elem2 in vertmask
            return(sum((int11)elem1*elem2));

     int11 horz =  for elem3 in win dot elem4 in horzmask
            return(sum((int11)elem3*elem4));

     int12 val = sq_root((int23)((int22)vert*vert)+(int22)horz*horz);
          uint8 val1 = (val > 255 ? 255 : (val < 0 ? 0: val));
     }
     return(array(val1));
} return (res);
```

Figure 3 Prewitt Edge Detection code in SA-C

The outer loop slides a 3×3 window through the image array in unit steps, while the loop body applies vertical and horizontal masks to the window; the square root function calculates the magnitude val; an array of magnitude val is returned as the final result. Because the bus width between FPGA and memory is 32 bits (one word), only four pixels per clock cycle can be fed into the FPGA.

We write VHDL for Prewitt edge detection in Figure 3 making use of the same square root algorithm [9]. The performance comparison results on a 256×256 8-bit image (Lena) are listed in Table 2.

Table 2 Performance Comparison of Prewitt Edge Detection

	Area	Clock Cycles	Clock Frequency (MHz)	Execution time (us)
VHDL	13%	16128	66	250
SA-C	24%	24481	50	490

We take wavelet transform as another complex algorithm example. Figure 4 shows the SA-C code. The performance comparison results are listed in Table 3.

```
export main;
int10, int10 Valsuint8(uint8 col[5])
{
  int10 mask[3] = {-1,2,-1};
  int11 d0 = for p in col[0:2] dot m in mask return(sum((int11)p*m));
  int11 d1 = for p in col[2:4] dot m in mask return(sum((int11)p*m));
  int11 d01 = d0 + d1;
  int11 ud01 = if (d01 < 0) return(-d01) else return(d01);
  bits11 b01 = ud01;
  bits11 bdiv8 = b01 >> 3;
  int11 udiv8 = bdiv8;
  int11 adj  = if (d01 < 0) return(-udiv8) else return(udiv8);
} return(col[2]+adj, d0);

int11, int11 Valsint10(int10 col[5])
{
  int11 mask[3] = {-1,2,-1};
   int12 d0 = for p in col[0:2] dot m in mask return(sum((int12)p*m));
  int12 d1 = for p in col[2:4] dot m in mask return(sum((int12)p*m));
  int12 d01 = d0 + d1;
  int12 ud01 = if (d01 < 0) return(-d01) else return(d01);
  bits12 b01 = ud01;
  bits12 bdiv8 = b01 >> 3;
  int12 udiv8 = bdiv8;
  int12 adj  = if (d01 < 0) return(-udiv8) else return(udiv8);
} return(col[2]+adj, d0);

int11[:,:], int11[:,:], int11[:,:], int11[:,:] main(uint8 src[:,:])
{
  int11 s[:,:], int11 dx2[:,:], int11 dy2[:,:], int11 dxy[:,:] =
  // PRAGMA (nextify_cse, scrunch, part_unroll(16,1), hardware(memout: 2 2 3 3))
    for window w[5,5] in src step(2,2)
    {
      int10 sy[5], int10 dy[5] =
        for uint3 colnum in [0~4]
        {
        int10 sval, int10 dval = Valsuint8(w[:,colnum]);
        } return(array(sval), array(dval));
      int11 s, int11 dx2 = Valsint10(sy);
      int11 dy2, int11 dxy = Valsint10(dy);
    } return(array(s), array(dx2), array(dy2), array(dxy));
} return(s, dx2, dy2, dxy);
```

Figure 4 Wavelet Transform code in SA-C

Table 3 Performance Comparison of Wavelet Transform

	Area	Clock Cycles	Clock Frequency (MHz)	Execution time (us)
VHDL	25%	43092	41	1051
SA-C	54%	73710	35	2100

From these comparisons it can be seen that the ratio between SA-C performance and handwritten VHDL performance is about 1:2. The main reasons for this difference are as follows.

First, it is relatively easy to store and reuse input data in handwritten code, while SA-C does not have this facility. This is significant because computer vision and image processing algorithms usually involve window sliding, meaning that the same input data are referenced in two or more continuous iterations, and I/O bandwidth is always a main bottleneck of FPGAs. Triggered by compiler options, SA-C compiler can eliminate redundant nodes if they compute the same value, and can replace the redundant computation by a chain of registers if the values were computed in earlier iterations. However, the bottom rows of the input data in the new window still have to be read in twice. At the same time, the new window cannot be made too big, because stripmining utilizes MIMD parallelism, which is very resource-intensive. On the other hand, the capacity of dual port block RAM in present FPGAs ranges from dozens of Kbit up to hundreds of Kbit, which is big enough to store the bottom rows of data in any resized window. Handwritten VHDL can apply this strategy elaborately, while the current version of SA-C does not use on-chip RAM at all. In fact it is doable to add VHDL RAM modules into AHA library and have the SA-C compiler treat these modules as other common components.

Secondly, SA-C generates VHDL code from the AHA library. In order to make the modules in AHA interconnect easily, instead of using state machines (apart from the FOREMAN unit) they employ a number of input and output control signals to communicate one another. So between outer loop iterations, which are pipelined, there is a seam (usually 4 idle clock cycles). In comparison, handwritten VHDL tends to use state machines to control the circuit actions, which naturally fulfill the control task using the current and previous circuit states. These kinds of 'protocols' avoid the seam and save clock cycles.

The difference in clock cycle number between handwritten and SA-C generated code is mostly due to these two reasons. Take the array_max and Prewitt edge detection examples, which make use of the same loop unrolling mechanism:

For an image file which is already in the on-board memory, the SA-C generated VHDL will require

$$N_{SA-C} = \left(m\frac{N}{4} + 4 \right) \bullet \frac{M}{m-2} \qquad (2)$$

clock cycles, where m is the row number of the resized sliding window, and the image size is M×N (N is divided by 4 because the memory bus is 32-bit wide, while each pixel is 8-bit). As indicated above, when the sliding window reaches the end of current row the circuit takes 4 extra clock cycles to restart reading the next row. The resized window has m rows, and the last 2 rows need to be fetched again as we're carrying out a 3×3 array operation. So we have M/(m-2) lines of resized window sliding in the whole image array, instead of M/m. In fact, the output array is 2 columns smaller than the input one, though this effect can be ignored for a 256×256 or larger image.

For handwritten VHDL, the number of clock cycle is

$$N_{VHDL} = \left(m \times \frac{N}{4} \right) \bullet \frac{M}{m} = \frac{MN}{4} \qquad (3)$$

(Of course, both SA-C and VHDL would take more clock cycles to wait for the memory I/O, which is time consuming).

From the above two equations we can obtain

$$\frac{N_{VHDL}}{N_{SA-C}} = \frac{m-2}{m+\frac{16}{N}} \qquad (4)$$

In the array_max and Prewitt edge detection examples, N=256 and m=6, giving a clock cycle ratio of 66%, which is in agreement with the experimental results in Tables 1 and 2.

A third reason that hand-written VHDL performs better is that SA-C has a lot of extra 'glue logic'

between its AHA modules, while handwritten code has relative fewer control signals between modules. Glitches due to this 'glue logic', which show up clearly in simulation waveforms, damage SA-C clock rate performance.

Finally, handwritten VHDL has an innate advantage in doing bit-precision manipulation compared with SA-C. In the wavelet transform example, SA-C uses extra shifter, multiplexer and pipeline stages just for a '×8' operation on a signed integer, while VHDL only spends one shifter and does not require a multiplexer.

4. Comparison in other aspects

The most attractive aspect of using a HLL is productivity. Take the wavelet transform as an example: a programmer familiar with SA-C can write and debug the code in Figure 3 in under 3 hours, while it would take about 150 hours to write and debug VHDL code for the same algorithm. Typically, the ratio of development time for the examples in this report ranged from 10 to 100 in favor of SA-C. This is, of course, the very reason that the language was created.

Another important issue is ease of acquisition of the necessary skills. A computer vision or image processing researcher who only knows C can learn SA-C within 1 week, while someone with appropriate digital circuit background might spend more than 3 months to learn VHDL. (Clearly, a SA-C programmer still needs some hardware background: For instance, the designer should be aware of the nature of loop unrolling even though it does not have to carried out manually).

The ratio between the number of lines of SA-C generated VHDL and that of handwritten VHDL ranges from 2 to 10 for the examples in this report. However, this apparent advantage of hand-written code is more than offset by the ease of code maintenance in the case of SA-C: SA-C generated VHDL code is only one step removed from the corresponding SA-C code, which being a HLL is vastly easier to maintain than HDL code.

Ultimately, SA-C changes how the system is programmed from a circuit design paradigm to a software design paradigm. Users benefit principally from its high productivity and easy code maintenance, at the expense of some device utilization and speed performance.

5. Related Work

Several projects have implemented various mechanisms to translate high-level program descriptions into hardware: PRISM [15]'s target system consisted of a Motorola 68010 processor and four Xilinx FPGAs. Napa C [10] with its pragmas can map the computations to either RISC processor or FPGA. Streams-C [5] allows programmers to write parallel C programs and generate RTL VHDL codes for a target Annapolis Microsystems Firebird FPGA board. Ptolemy [11] focuses on assembly of concurrent systems. MATCH [12] maps MATLAB descriptions to heterogeneous computing system consisting of FPGA, embedded processor and DSP component. PICO-NPA [13] is one aspect of HP PICO. Loop nests are expressed in C and synthesized to non-programmable accelerators. Handel-C [14] is a C-like language that can generate EDIF netlist directly and VHDL code as well. Nimble [16]'s target architecture is a general-purpose processor with a dynamically reconfigurable datapath. DEFACTO [17] maps the applications to a microprocessor and FPGAs as coprocessors based on SUIF. Proceler's system [18] compiles C to a microprocessor and an FPGA.

6. Conclusion

The comparisons show that in terms of design time, SA-C's productivity is 10 to 100 times of that of handwritten VHDL. But device area and execution time are the penalties: Use of SA-C can double device utilization; handwritten designs have 10%~20% higher maximum clock frequencies; handwritten designs typically save about 40% in clock cycles with respect to SA-C generated circuits. Overall, the execution time of a handwritten design is approximately 50% of that of SA-C version.

From the comparison and analysis, we know that there is plenty of room for improvement in SA-C. For example, using on-chip dual-port RAM to store data for subsequent iterations can save I/O bandwidth, which is usually a major bottleneck in reconfigurable computing systems. In the current hardware system, it is possible to infer dual-port RAM for the Xilinx Virtex Architecture to AHA library. Also, the number of idle clock cycles might be reduced by using state machines to replace complex control signals between AHAs.

During the last ten years, increases in FPGA density and speed followed or exceeded Moore's law.

Unfortunately, design ability has not kept up the same pace. This application study shows that high-level language synthesis for reconfigurable computing is beginning to catch up and has the potential to bridge this gap to some extent.

7. References

[1] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems. The Journal of Supercomputing, Volume 21, pages 117-130, 2002.

[2] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar and A.P.W. Böhm. An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware. IEEE Trans. on VLSI, Vol. 9(1), February 2001.

[3] J.P. Hammes and A.P.W. Böhm. The SA-C Language - Version 1.0. http://www.cs.colostate.edu/cameron/

[4] J.P. Hammes, Monica Chawathe and A.P.W. Böhm. The SA-C Compiler -Version June 2001. http://www.cs.colostate.edu/cameron/

[5] M.B. Gokhale and J.M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high-level language. In IEEE international Symposium on FPGAs for Custom Computing Machines, 2000.

[6] Jan Frigo, Maya Gokhale and Dominique Lavenier. Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective. 9th ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 2001.

[7] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, D. Zaretsky. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In FCCM 00, Napa, CA, April 2000.

[8] Annapolis Microsystems Inc. WILDSTAR hardware Reference Manual.

[9] Yamin Li and Wanming Chu. A New Non-Restoring Square Root Algorithm and Its VLSI Implementations. ICCD'96, International Conference on Computer Design, October 7 - 9, 1996, Austin, Texas, USA

[10] Maya B. Gokhale, Janice M. Stone. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In IEEE Symposium on FPGAs for Custom Computing Machines April 1998.

[11] Edward A. Lee, Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 6, 2001.

[12] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, A MATLAB Compiler for Distributed Heterogeneous Reconfigurable Computing Systems. In Int. Symp. On FPGA Custom Computing Machines (FCCM-2000), Napa Valley, CA, 2000.

[13] R. Schreiber, S Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. Journal of VLSI Signal Processing, 2001.

[14] Handel-C Language Overview. Celoxica, Inc. http://www.celoxica.com

[15] Wazlowski M, Agarwal L, Lee T, Smith A, Lam E, Athanas P, Silverman H, Ghosh S. PRISM-II Compiler and Architecture. Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines. FCCM'93, Napa, CA, USA, April 1993.

[16] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In Design Automation Conf. (DAC), 1999.

[17] M. Hall, P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain, and Granacki J, "DEFACTO: A Design Environment for Adaptive Computing Technology," in 6th Reconfigurable Architectures Workshop (RAW'99), 1999.

[18] Proceler Corp. http://www.proceler.com