The Journal of Supercomputing, 21, 117–130, 2002

Mapping a Single Assignment Programming Language to Reconfigurable Systems*

W. BÖHM, J. HAMMES, B. DRAPER, M. CHAWATHE, C. ROSS AND R. RINKER bohm@cs.colostate.edu
Colorado State University

W. NAJJAR
University of California Riverside

Abstract. This paper presents the high level, machine independent, algorithmic, single-assignment programming language SA-C and its optimizing compiler targeting reconfigurable systems. SA-C is intended for Image Processing applications. Language features are introduced and discussed. The intermediate forms DDCF, DFG and AHA, used in the optimization and code-generation phases, are described. Conventional and reconfigurable system specific optimizations are introduced. The code generation process is described. The performance for these systems is analyzed, using a range of applications from simple Image Processing Library functions to more comprehensive applications, such as the ARAGTAP target acquisition prescreener.

Keywords: reconfigurable computing systems, FPGA, image processing, high level languages, optimizing compilation

1. Introduction

Recently, the computer vision and image processing communities have become aware of the potential for massive parallelism and high computational density in FPGAs. FPGAs have been used for real-time point tracking [6], stereo vision [37], color-based detection [7], image compression [16], and neural networks [9]. The biggest obstacle to the more widespread use of reconfigurable computing systems lies in the difficulty of developing application programs. FPGAs are typically programmed using hardware description languages such as VHDL [26]. Application programmers are typically not trained in these hardware description languages and usually prefer a higher level, algorithmic programming language to express their applications.

The Cameron Project [15] has created a high-level algorithmic language, named SA-C [14], for expressing image processing applications and compiling them to FPGAs. The SA-C compiler provides one-step compilation to host executable and FPGA configurations. After parsing and type checking, the SA-C compiler converts the program to a hierarchical data dependence and control flow (DDCF) graph representation. DDCF graphs are used in many optimizations, some general and some

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

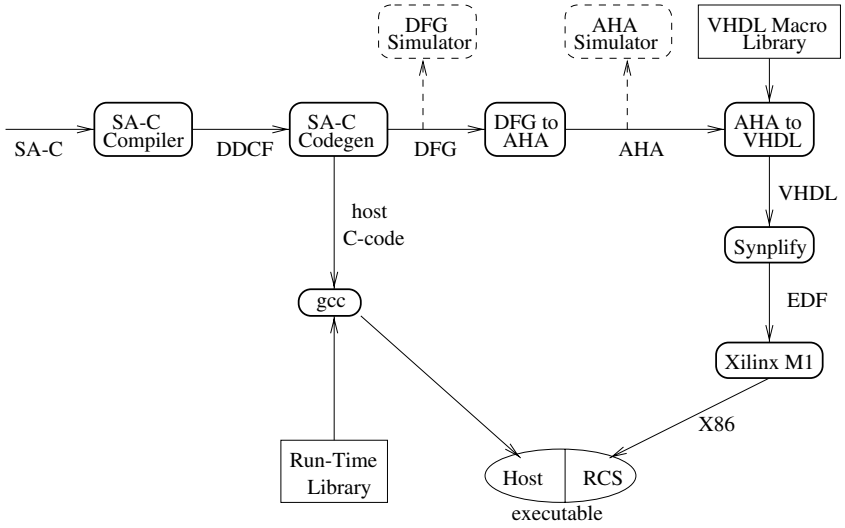

Figure 1. SA-C compilation system.

specific to SA-C and its target platform. After optimization, the program is converted to a combination of dataflow graphs (DFGs) and host code. DFGs are then compiled to VHDL code via Abstract Hardware Architecture (AHA) graphs. The VHDL code is synthesized and place-and-routed to FPGAs by commercial software tools. Figure 1 shows a high-level view of the system.

To aid in program development, it is possible to view and simulate intermediate forms. For initial debugging the complete SA-C program can be executed on the host. All intermediate graph forms can be viewed, and DFG and AHA graphs can be simulated. The SA-C compiler can run in stand-alone mode, but it also has been integrated into the Khoros$^{(TM)}$ [19] graphical software development environment.

The rest of this paper is organized as follows. An overview of the SA-C language is presented in Section 2. Compiler optimizations and pragmas are discussed in Section 3. Translations to dataflow graphs and then to VHDL via AHA are discussed in Section 4. Applications and their performance data are presented in Section 5. References to related work are given in Section 6, and Section 7 concludes and describes future work.

2. The SA-C language

The design goals of SA-C are to have a language that can express image processing (IP) applications elegantly, and to allow seamless compilation to reconfigurable hardware. IP applications are supported by data parallel for loops with structured access to rectangular multidimensional arrays. Reconfigurable computing requires fine grain expression level parallelism, which is easily extracted from a SA-C program because of its *single assignment* semantics. Variables in SA-C are associated with wires, not with memory locations. This avoids von Neumann memory model

complications and allows for better compiler analysis and translation to DFGs. Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. The extents of SA-C arrays can be determined either dynamically or statically. The type declaration int14 M[:,6] for example, is a declaration of a matrix M of 14-bit signed integers. The left dimension will be determined dynamically; the right dimension has been specified.

The most important aspect of SA-C is its treatment of for loops and their close interaction with arrays. SA-C is expression oriented, so every construct (including a loop) returns one or more values. A loop has three parts: one or more generators, a loop body and one or more return values. The generators provide parallel array access operators that are concise and easy for the compiler to analyze. There are four kinds of loop generators: *scalar*, *array-element*, *array-slice* and *window*. The scalar generator produces a linear sequence of scalar values, similar to Fortran's do loop. The array-element generator extracts scalar values from a source array, one per iteration. The array-slice generator extracts lower dimensional sub-arrays (e.g. vectors out of a matrix). Finally, window generators allow rectangular sub-arrays to be extracted from a source array. All possible sub-arrays of the specified size are produced, one per iteration. The dot product operator combines generators and runs them in lock step. A loop can return arrays and reductions built from values that are produced in the loop iterations, such as sum, product, min, and max.

Figure 2 shows SA-C code for the Prewitt edge detector [27], a standard IP operator. The outer for loop is driven by the extraction of 3×3 sub-arrays from array Image. Its loop body applies two masks to the extracted window W, producing a magnitude. An array of these magnitudes is collected and returned as the program's result. The shape of the return array is derived from the shape of Image and the loop's generator. If Image were a 100×200 array, the result array M would have a shape of 98×198.

Loop carried values are allowed in SA-C using the keyword next instead of a type specifier in a loop body. This indicates that an initial value is available outside the loop, and that each iteration can use the current value to compute a next value.

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

Figure 2. Prewitt edge detector code in SA-C.

3. Optimizations and pragmas

The compiler's internal program representation is a hierarchical graph form called the "Data Dependence and Control Flow" (DDCF) graph. DDCF subgraphs correspond to source language constructs. Edges in the DDCF express data dependencies, opening up a wide range of loop- and array-related optimization opportunities.

Figure 3 shows the initial DDCF graph of the Prewitt program of Figure 2. The FORALL and DOT nodes are compound, containing subgraphs. Black rectangles

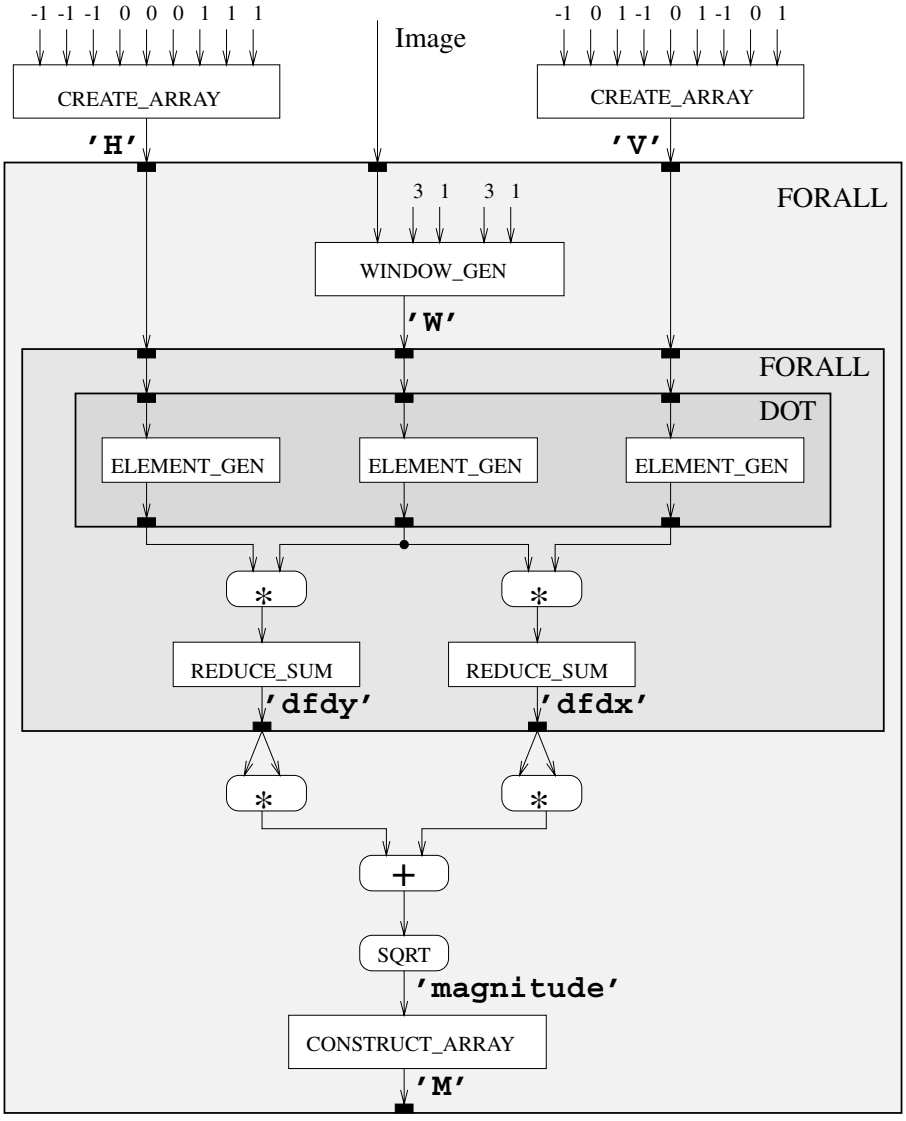

Figure 3. DDCF graph for Prewitt program.

along the top and bottom of a compound node represent input ports and output ports. The outer FORALL has a single window generator operating on a two-dimensional image, so it requires window size and step inputs for each of the two dimensions. In this example, both dimensions are size three, with unit step sizes. The output of the WINDOW_GEN node is a 3×3 array that is passed into the inner FORALL loop. This loop has a DOT graph that runs three generators in parallel, each producing a stream of nine values from its source array. Each REDUCE_SUM node sums a stream of values to a single value. Finally, the CONSTRUCT_ARRAY node at the bottom of the outer loop takes a stream of values and builds an array with them.

Many IP operators involve fixed size and often constant convolution masks. A ***Size Inference*** pass propagates information about constant size loops and arrays through the dependence graph. Array size information can propagate from a loop or source array to its target and vice versa. In addition, size information from one generator can be used to infer sizes of other generators. Effective size inference allows other optimizations, such as Full Loop Unrolling and Array Elimination, to take place.

Full Unrolling of loops with small, compile time assessable numbers of iterations can be important when generating code for FPGAs, because it spreads the iterations in code space rather than in time. Small loops occur frequently as inner loops in IP codes, for example in convolutions with fixed size masks.

Array Value Propagation searches for array references with constant indices, and replaces such references with the values of the array elements. When the value is a compile time constant, this enables constant propagation. In the Prewitt example, this optimization removes the arrays H and V entirely.

Loop Carried Array Elimination The most efficient representation of arrays in loop bodies is to have their values reside in registers. The important case is that of a loop carried array that changes values but not size during each iteration. To allocate a fixed number of registers for these arrays two requirements need to be met. 1) The compiler must be able to infer the size of the initial array computed outside the loop. 2) Given this size, the compiler must be able to infer that the next array value inside the loop is of the same size.

N-dimensional Stripmining extends stripmining [36] and creates an intermediate loop with fixed bounds. The inner loop can be fully unrolled with respect to the newly created intermediate loop, generating a larger, more parallel circuit. The compiler generates code to compute left over fringes.

Some (combinations of) operators can be inefficient to implement directly in hardware. For example the computation of magnitude in Prewitt requires multiplications and square root operators. The evaluation of the whole expression can be replaced by an access to a ***Lookup Table***, which the compiler creates by wrapping a loop around the expression, recursively compiling and executing the loop, and reading the results.

The performance of many systems is limited by the time required to move data to the processing units. ***Fusion*** of producer-consumer loops is often helpful, since it reduces data traffic and may eliminate intermediate data structures. In simple cases, where arrays are processed element-by-element, this is straightforward. However,

the windowing behavior of many IP operators presents a challenge. Consider the following loop pair:

```
uint8 R0[:,:] =
  for window W[2,2] in Image return (array (f(W)));
uint8 R1[:,:] =
  for window W[2,2] in R0 return (array (g(W)));
```

If Image is a $d_0 \times d_1$ array, R0 is a $(d_0 - 1) \times (d_1 - 1)$ array, and R1 will be $(d_0 - 2) \times (d_1 - 2)$, so the two loops do not have the same number of iterations. Nevertheless, it is possible to fuse such a loop pair by examining their data dependencies. One element of R1 depends on a 2×2 sub-array of R0, and the four values in that sub-array together depend on a 3×3 sub-array of Image. It is possible to replace the loop pair with one new loop that uses a 3×3 window and has a loop body that computes one element of R1 from nine elements of Image.

Common Subexpression Elimination (CSE) is a well known compiler optimization that eliminates redundancies by looking for identical subexpressions that compute the same value. This could be called "spatial CSE" since it looks for common subexpressions within a block of code. The SA-C compiler performs conventional spatial CSE, but it also performs ***Temporal CSE***, looking for values computed in one loop iteration that were already computed in previous loop iterations. In such cases, the redundant computation can be eliminated by holding such values in registers so that they are available later and need not be recomputed. Here is a simple example containing a temporal common subexpression:

```
for window W[3,2] in A {
  uint8 s0 = array_sum (W[:,0]);
  uint8 s1 = array_sum (W[:,1]);
} return (array (s0+s1));
```

This code computes a separate sum of each of the two columns of the window, then adds the two. Notice that after the first iteration of the loop, the window slides to the right one step, and the column sum s1 in the first iteration will be the same as the column sum s0 in the next iteration. By saving s1 in a register, the compiler can eliminate one of the two column sums, nearly halving the space required for the loop body.

A useful phenomenon often occurs with Temporal CSE: one or more columns in the left part of the window are unreferenced, making it possible to eliminate those columns. ***Narrowing*** the window lessens the FPGA space required to store the window's values.

In many cases the performance tradeoffs of various optimizations are not obvious; sometimes they can only be assessed empirically. The SA-C compiler allows many of its optimizations to be controlled by ***user pragmas*** in the source code. This allows the user to experiment with different approaches and evaluate the space-time tradeoffs.

4. Compiler backend

A dataflow graph (DFG) is a non-hierarchical and asynchronous program representation. DFGs can be viewed as abstract hardware circuit diagrams without timing or resource contention taken into account. Nodes are operators and edges are data paths. DFGs have token driven semantics.

The SA-C compiler attempts to translate every innermost loop to a DFG. The innermost loops the compiler finds may not be the innermost loops of the original program, as loops may have been fully unrolled or stripmined. 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.

In the Prewitt program shown earlier, the DDCF graph is transformed to the DFG shown in Figure 4. The SUM nodes can be implemented in a variety of ways, including a tree of simple additions. The window generator also allows a variety of

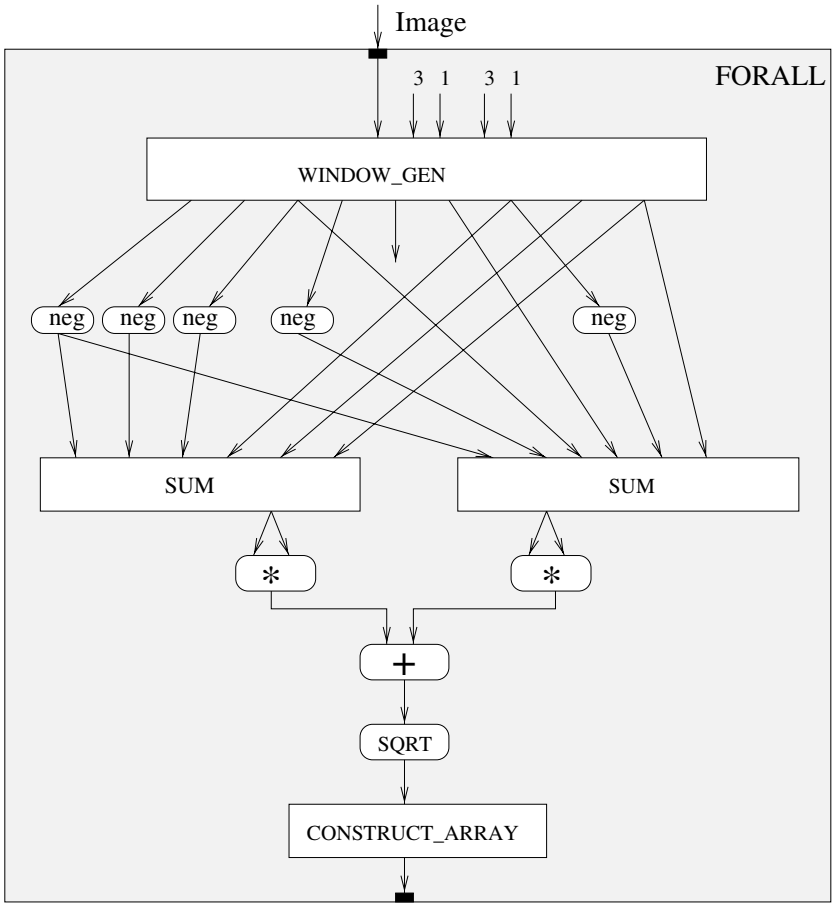

Figure 4. DFG for Prewitt after optimizations.

implementations, based on the use of shift registers. The CONSTRUCT_ARRAY node streams its values out to a local memory.

DFGs are translated into a lower level form called Abstract Hardware Architecture (AHA). This is also a graph form, but with nodes that are more fine-grained than DFG nodes and that can be translated to simple VHDL components. AHA graphs have *clocked*, *semi-clocked* and *non-clocked* nodes. The clocked and semi-clocked nodes have internal state but only the clocked nodes participate in the *handshaking* needed to synchronize computations and memory accesses. Some clocked nodes communicate via an arbitrator with a local memory. An AHA graph is organized as a sequence of *sections*, each with a top and a bottom boundary. A section boundary consists of clocked nodes, whereas its internal nodes are non-clocked or semi-clocked. In the AHA model, a section fires when all clocked nodes at its top boundary can produce new values and all clocked nodes at its bottom boundary can consume new values. This contrasts with DFGs, where each node independently determines when it can fire.

The AHA graph of the Prewitt code is too large and complex to display in this paper (1568 nodes). Figure 5 shows a dataflow graph on the left and an AHA graph on the right of a much simpler code fragment, which copies a one dimensional array A to another array B:

```
uint32 B[:] = for a in A return(array(a))
```

The DFG consists of an Element Generator node extracting elements out of A, and a Construct Tile node collecting the elements of B. The top three nodes in the AHA graph implement the Element Generator node, whereas the bottom three AHA nodes implement the Construct Tile node. The inputs 1, 2 and 3 in the AHA

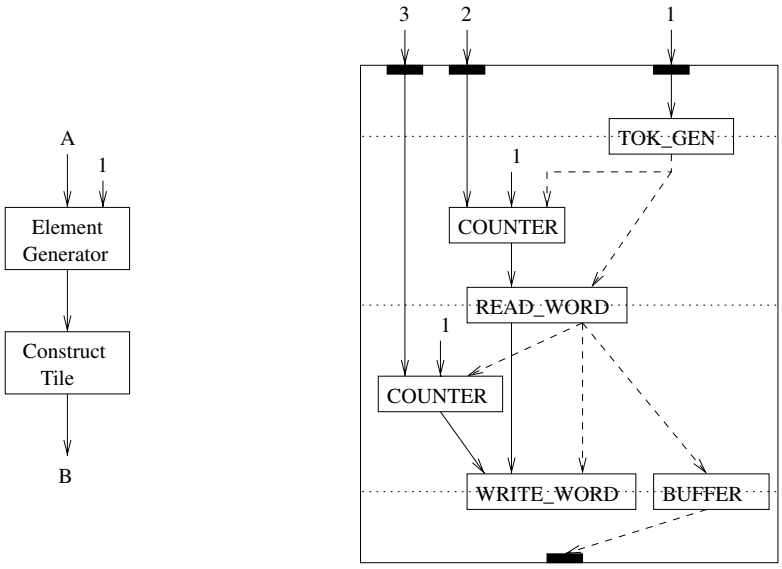

Figure 5. One-dimensional array copy in dataflow and AHA.

graph represent the extents of A, the start address of A, and the start address of B, respectively. Dotted horizontal lines represent section boundaries. Upon an input n, a TOK_GEN node produces n+1 control signals (dashed edges): n 0-s and a 1. A COUNTER is a semi-clocked node, starting at its left input and incrementing with its middle input (1 here). BUFFER, READ_WORD and WRITE_WORD speak for themselves.

Some low-level optimizations take place in this stage. A ***Bitwidth Narrowing*** phase is performed just before AHA graphs are generated. ***Dead code elimination*** and ***graph simplification*** sweeps are applied on the AHA graph. A ***Pipelining*** phase uses node propagation delay estimates to compute the delay for each AHA section, and adds layers of pipeline registers in sections that have large propagation delays. The maximum number of pipeline stages added to the AHA graph is user controlled. Reducing propagation delays is important because it increases clock frequency.

AHA graph simulation allows the user (or, more likely, the compiler or system developer) to verify program behavior. The AHA simulator strictly mimics the hardware behavior with respect to clock cycles and signals traveling over wires. This removes the need for time consuming VHDL simulation and hardware level debugging. AHA-to-VHDL translation is straightforward; AHA nodes translate to VHDL components, which are connected according to the AHA edges.

5. Applications

The current test platform in Cameron is the WildStar Board, produced by Annapolis Micro Systems [2]. The WildStar has three XCV2000E Virtex FPGAs made by Xilinx [39]. The WildStar board is capable of operating at frequencies from 25 MHz to 180 MHz. It communicates over the PCI bus with the host computer at 33 MHz. In our system, the board is housed in a 266-MHz Pentium-based PC. This section compares the performance of SA-C codes running one WildStar FPGA chip to the performance of C or assembly code running on an 800 MHz Pentium III. A 512×512 8 bit image is used for input.

5.1. Intel image processing library

When comparing simple IP operators one might write corresponding SA-C and C codes and compare them on the WildStar and Pentium. However, neither the Microsoft nor the Gnu C++ compilers fully exploit the Pentium's MMX technology. Instead, we compare SA-C codes to corresponding operators from the Intel Image Processing Library (IPL). The Intel IPL library consists of a large number of low-level Image Processing operations. Many of these are simple point (pixel-) wise operations such as square, add, etc. These operations have been coded by Intel for highly efficient MMX execution.

For example, in case of a pointwise add, naively implemented SA-C runs four times slower on the FPGA than MMX assembly code on the Pentium. However, if care is taken by the SA-C programmer to strip-mine and interleave both input and

output memories to take advantage of 64 bit memory access, the FPGA is twice as fast as the Pentium. Overall, the difference in run times for simple operators is not significant. This result is not surprising. The compute/communicate ratio in these codes is very low, and thus the memory bandwidth governs the performance.

However, the Prewitt edge detector is more complex. It requires convolving the image with two 3×3 masks, squaring the results, summing the squares, and finally computing the square root of the sum (see Figures 2 and 3). Prewitt written in C using a single IPL routine (iplConvolve) runs on the Pentium, for our 512×512 test image, in 158 milliseconds. In comparison the equivalent SA-C code on the WildStar runs in less than 2 milliseconds, a speedup of 80 over the Pentium.

5.2. ARAGTAP

The ARAGTAP pre-screener [29] was developed by the U.S. Air Force at Wright Labs as the initial focus-of-attention mechanism for a SAR automatic target recognition application. Aragtap's components include down sampling, dilation, erosion, positive differencing, majority thresholding, bitwise and, percentile thresholding, labeling, label pruning, and image creation. All these components, apart from label pruning and image creation, have been written in SA-C. Most of the computation time is spent in a sequence of eight gray-scale morphological dilations, and a later sequence of eight gray-scale erosions. Four of these dilations are with a 3×3 mask of 1's in the shape of a square, the other four are with a 3×3 mask with 1's in the shape of a cross and with zeros at the edges.

The dilate and erode loops allow temporal CSE and window narrowing. A hand optimized C implementation of a dilate sequence running on the Pentium takes 66 milliseconds. The SA-C compiler fuses the whole dilate sequence into one loop, which takes 3 milliseconds to execute on the WildStar, delivering a speedup of 22 compared to the Pentium.

5.3. Canny

In section 5.1 we discussed the Prewitt operator. A more sophisticated edge detector is the Canny operator, which comprises a four step process. The four steps are 1) image smoothing, 2) computing edge magnitudes and (discretized) orientations, 3) non-maximal suppression in the gradient direction, and 4) hysteresis labeling, which is a connected components algorithm. For a clear and simple explanation of Canny and the reasoning behind it, see [34, Chapter 4]. The first three steps of Canny were implemented in SA-C and run on the reconfigurable hardware. Although connected components can be written in SA-C, it can currently not be compiled to FPGAs. Therefore, we assume that the last step will be performed on the host. The compiler performed eight-fold vertical stripmining, among other optimizations. SA-C execution time on the WildStar is 6 milliseconds.

Comparing the performance to a Pentium is hard. A version of the same program was written in C, using Intel's IPL whenever possible. The resulting program took

850 milliseconds on the Pentium. However, Intel's OpenCV library has a hand-optimized assembly-coded version of Canny that includes the fourth (connected components) step. By setting the high and low thresholds to be the same (so that connected components will not iterate), the OpenCV routine takes 135 milliseconds. So SA-C was 22 times faster than assembly code and 140 times faster than a C plus IPL implementation.

5.4. Wavelet

Wavelets are commonly used for multi-scale analysis in computer vision, as well as for image compression. Honeywell has defined a set of benchmarks for reconfigurable computing systems, including a wavelet-based image compression algorithm [20]. This code takes one image and returns four quarter sized images, three of which are derivatives of the original. The SA-C code takes 2 milliseconds execution time on the FPGA, whereas on the Pentium the Honeywell C code took 75 milliseconds. SA-C was 37 times faster.

Concluding, apart from very simple point wise operations, SA-C on the WildStar using one Virtex 2000E chip runs between 20 and 75 times faster than the fastest code we could run on the 800 MHz Pentium III.

6. Related work

Hardware and software research in reconfigurable computing is active and ongoing. Hardware projects fall into two categories—those using commercial off-the-shelf components (e.g., FPGAs), and those using custom designs.

The Splash-2 [8] is an early (circa 1991) implementation of an RCS, built from 17 Xilinx [38] 4010 FPGAs, and connected to a Sun host as a co-processor. Several different types of applications have been implemented on the Splash-2, including searching [17, 28], pattern matching [31], convolution [30] and image processing [3].

Representing the current state of the art in FPGA-based RCS systems are the AMS WildStar [2] and the SLAAC project [32]. Both utilize Xilinx Virtex [39] FPGAs, which offer over an order of magnitude more programmable logic, and provide a several-fold improvement in clock speed, compared to the earlier chips.

Several projects are developing custom hardware. The Morphosys project [22] combines an on-chip RISC processor with an array of reconfigurable cells (RCs). Each RC contains an ALU, a shifter, and a small register file.

The RAW Project [35] also uses an array of computing cells, called *tiles*; each tile is itself a complete processor, coupled with an intelligent network controller and a section of FPGA-like configurable logic that is part of the processor data path. The PipeRench [12] architecture consists of a series of *stripes*, each of which is a pipeline stage with an input interconnection network, a lookup-table based PE, a results register, and an output network. The system allows a virtual pipeline of any size to be mapped onto the finite physical pipeline.

On the software front, a framework called "Nimble" compiles C codes to reconfigurable targets where the reconfigurable logic is closely coupled to an embedded CPU [21]. Several other hardware projects also involve software development. The RAW project includes a significant compiler effort [1] whose goal is to create a C compiler to target the architecture. For PipeRench, a low-level language called DIL [11] has been developed for expressing an application as a series of pipeline stages mapped to stripes.

Several projects (including Cameron) focus on hardware-independent software for reconfigurable computing; the goal—still quite distant—is to make development of RCS applications as easy as for conventional processors, using commonly known languages or application environments. Several projects use C as a starting point. DEFACTO [13] uses SUIF as a front end to compile C to FPGA-based hardware. Handel-C [24] both extends the C language to express important hardware functionality, such as bit-widths, explicit timing parameters, and parallelism, and limits the language to exclude C features that do not lend themselves to hardware translation, such as random pointers. Streams-C [10] does a similar thing, with particular emphasis on extensions to facilitate the expression of communication between parallel processes. SystemC [33] and Ocapi [18] provide C++ class libraries to add the functionality required of RCS programming to an existing language.

Finally, a couple of projects use higher-level application environments as input. The MATCH project [4, 5, 25] uses MATLAB as its input language—applications that have already been written for MATLAB can be compiled and committed to hardware, eliminating the need for re-coding them in another language. Similarly, CHAMPION [23] is using Khoros [19] for its input—common glyphs have been written in VHDL, so GUI-based applications can be created in Khoros and mapped to hardware.

7. Conclusions and future work

The Cameron Project has created a language, called SA-C, for one-step compilation of image processing applications that target FPGAs. Various optimizations, both conventional and novel, have been implemented in the SA-C compiler.

The system has been used to implement routines from the Intel IPL, as well as more comprehensive applications, such as the ARAGTAP target acquisition prescreener. Compared to Pentium III/MMX technology built at roughly the same time, the SA-C system running on an Annapolis WildStar board with one Virtex 2000 FPGA has similar performance when it comes to small IPL type operations, but shows speedups up to 75 when it comes to more complex operators such as Prewitt, Canny, Wavelet, and Dilate and Erode sequences. Performance evaluation of the SA-C system has just begun. 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 the tradeoffs between conventional functional code and lookup tables.

Currently, the VHDL generated from the AHA graphs ignores the structural information available in the AHA graph. We will soon be investigating the use of

Relatively Placed Macros (RPM) as a method to make some of the structural information explicit to the synthesis tools. Providing constraints to specify the placement of nodes relative to each other may prove to decrease synthesis and place and route time.

Also, stream data structures are being added to the SA-C language, which will allow multiple cooperating processes to be mapped onto FPGAs. This allows expression of streaming video applications, and partitioning of a program over multiple chips.

References

1. A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, M. Srikrishna, and D. Taylor. The RAW compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, 1997.
2. Annapolis Micro Systems. *STARFIRE Reference Manual*. Annapolis Micro Systems, Inc., Annapolis, MD, 1999. Available at `www.annapmicro.com`.
3. P. M. Athanas and A. L. Abbott. Processing images in real time on a custom computing platform. In R. W. Hartenstein and M. Z. Servit, eds., *Field-Programming Logic Architecture, Synthesis, and Applications*, pp. 156–167. Springer-Verlag, Berlin, 1994.
4. P. Banerjee, N. Shenoy, A. Choudary, S. Hauck, and M. Halder. A MATLAB compiler for distributed heterogeneous, reconfigurable computing systems. In *The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
5. P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, N. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden. MATCH: A MATLAB compiler for configurable computing systems. Technical report CPDC-TR-9908-013. Center for Parallel and distributed Computing, Northwestern University, 1999.
6. A. Benedetti and P. Perona. Real-times 2-D feature detection on a reconfigurable computer. In *IEEE Conference on Computer Vision and Pattern Recognition*, Santa Barbara, CA, 1998.
7. D. Benitez and J. Cabrera. Reactive computer vision system with reconfigurable architecture. In *International Conference on Vision Systems*, Las Palmas de Gran Canaria, Spain, 1999.
8. D. Buell, J. Arnold, and W. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine.* IEEE CS Press, Los Alamitos, CA, 1996.
9. J. Eldredge and B. Hutchings. RRANN: A hardware implementation of the backpropagation algorithm using reconfigurable FPGAs. In *IEEE International Conference on Neural Networks*, Orlando, FL, 1994.
10. M. Gokhale, J. M. Strone, J. Arnold, and M. Kalinowski. Stream oriented FPGA computing in streams-C. In *The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
11. S. C. Goldstein and M. Budiu. *The DIL Language and Compiler Manual*. Carnegie Mellon University. Available at `www.ece.cmu.edu/research/piperench/dil.ps`.
12. S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the International Symposium on Computer Architecture (ISCA '99)*, 1999. Available at `www.cs.cmu.edu/~mihaib/research/isca99.ps.gz`.
13. M. Hall, P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain, and J. Granacki. DEFACTO: A design environment for adaptive computing technology. In *Proceedings of the 6th Reconfigurable Architectures Workshop (RAW'99)*, Springer-Verlag, Berlin, 1999.
14. J. Hammes and W. Böhm. The SA-C language—version 1.0. 1999. Available at `www.cs.colostate.edu/cameron`.
15. J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT'99*, 1999.

16. R. Hartenstein, J. Becker, R. Kress, H. Reinig, and K. Schmidt. A reconfigurable machine for applications in image and video compression. In *Conference on Compression Technologies and Standards for Image and Video Compression*, Amsterdam, Holland, 1995.
17. D. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 185–192. CS Press, Los Alamitos, CA, 1993.
18. IMEC. Ocapi overview, 2000.
19. K. Konstantinides and J. Rasure. The Khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing*, 3:243–252, 1994.
20. S. Kumar. A benchmark suite for evaluating configurable computing systems—status, reflections, and future directions. In *FPGA2000 Eighth International Symposium on FPGAs*, Feb. 10–12. Monterey, CA, 2000.
21. Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the 37th Design Automation Conference*, 1999.
22. G. Lu, H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi. The morphosis parallel reconfigurable system. In *Proceedings of EuroPar 99*, 1999.
23. 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.
24. Oxford Hardware Compiler Group. The Handel language. Technical report, Oxford University, 1997.
25. 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 *Proceedings of the 11th IASTED Parallel and Distributed Computing and Systems Conference (PDCS'99)*, 1999.
26. D. Perry. *VHDL*. McGraw-Hill, New York, 1993.
27. J. M. S. Prewitt. Object enhancement and extraction. In B. S. Lipkin and A. Rosenfeld, eds., *Picture Processing and Psychopictorics*. Academic Press, New York, 1970.
28. D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 172–178. CS Press, Los Alamitos, CA, 1993.
29. S. Raney, A. Nowicki, J. Record, and M. Justice. ARAGTAP ATR system overview. In *Theater Missile Defense 1993 National Fire Control Symposium*. Boulder, CO, 1993.
30. N. K. Ratha, D. T. Jain, and D. T. Rover. Convolution on Splash 2. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 204–213. CS Press, Loa Alamitos, CA, 1995.
31. N. K. Ratha, D. T. Jain, and D. T. Rover. Fingerprint matching on Splash 2. In *Splash 2: FPGAs in a Custom Computing Machine*, pp. 117–140. IEEE CS Press, 1996.
32. B. Schott, S. Crago, C. C., J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti. Reconfigurable architectures for systems level applications of adaptive computing, 1997. Available at `http://www.east.isi.edu/SLAAC/`.
33. SystemC. SystemC Homepage. 2000.
34. E. Trucco and A. Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, Englewood Cliffs, NJ, 1998.
35. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *Computer*, 1997.
36. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, Reading, MA, 1996.
37. J. Woodfill and B. V. Herzen. Real-time stereo vision on the PARTS reconfigurable computer. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1997.
38. Xilinx Incorporated. *The Programmable Logic Databook*. Xilinx, Inc., San Jose, CA, 1998. Available at `www.xilinx.com`.
39. Xilinx Incorporated. *Virtex 2.5V Field Programmable Gate Arrays: Preliminary Product Description*. Xilinx, Inc., San Jose, CA, 1999. Available at `www.xilinx.com`.