

An Application-Specific Compiler for High-Speed Binary Image Morphology *

Scott Hemmert, Brad Hutchings and Anshul Malvi
Brigham Young University
{hemmert, hutch, amalvi}@ee.byu.edu

Abstract

This paper discusses a two-level compilation scheme used for generating high-speed binary image morphology pipelines from a textual description of the algorithm. The first-level compiler generates a generic morphology machine which is customized for the specified set of instructions by the second-level compiler. Because the generic machine is reused, we are able to avoid long synthesis times and achieve compile times similar to software compile times, while still achieving a 10X speed-up over the software implementation.

1 Introduction

The Focus of Attention (FOA) algorithms are used by Sandia National Laboratories as a preprocessing stage to their Automatic Target Recognition engine. FOA is used to process synthetic aperture radar (SAR) images with the goal of finding regions in the image which contain likely targets. The exact FOA algorithm used is dependent on the type and size of targets as well as the radar system generating the images. Weather conditions can also cause changes to be made to the algorithm. The goal of FOA is to reduce the size of the data set for the more computationally complex stages of ATR which identify the existence and type of targets.

Because of its demanding computational requirements, FOA was originally implemented on a specialized morphology computer, called the CYTO computer [1], manufactured by the Environmental Research Institute of Michigan (ERIM). The CYTO computer was constructed using custom ASICs to create a semi-programmable, image-processing pipeline computer. The CYTO computer could be programmed to perform a wide variety of image morphology operations using an arcane language known as C4PL (CYTO Portable Parallel Picture Processing Language)[2]. An FOA script then is implemented as a list of C4PL operators performed in sequence. Due to improvements in IC fabrication, the CYTO computer quickly became obsolete and has been

replaced by a software implementation. However, C4PL outlived the CYTO computer and is still in use as the programming (or scripting) language used to describe FOA applications.

The current software implementation by Sandia National Laboratories is a collection of optimized C functions which emulate the operation of a subset of C4PL instructions. This software reads the FOA script and chains together the operations specified in the script. Due to advances in microprocessors, this sequential software approach is many times faster than the original CYTO computer.

Even though the software version of FOA was a vast improvement over the CYTO computer, it still did not reach the performance goals set forth by Sandia, and accelerating FOA with custom ASICs, as was done with the CYTO computer, is not an option because the final solution must use COTS (Commercial off-the-shelf) hardware. In addition to performance requirements, Sandia also put some requirements on programmability and programmer usability:

- Assume programmers have no hardware experience. Image Processing specialists write the algorithm.
- Actual FOA algorithms used are classified. The algorithm must be specified and mapped to hardware without involving unclassified personnel.
- The algorithms are to be specified in the C4PL programming language.
- Because operating conditions may dictate an algorithm change, in-field modifications must be possible. The maximum compile time that can be tolerated is half a work day (4 hours).
- Final solution must be embeddable in an approved ruggedized form factor.

To meet these requirements, we mapped the design to Xilinx XC4000 family FPGAs and utilized a two-level compilation approach. The first stage of the compiler generates a structural design specific to a particular platform, which provides the generic hardware structure sufficiently large to perform the desired operations. The second phase customizes the generic pipeline to perform the specified C4PL operations. This paper discusses the evolutionary steps along the path leading to the final approach and implementation and compares the various approaches both for circuit size and com-

*Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

pilation speed. We will also compare our final hardware approach with Sandia’s software implementation.

2 Background

The C4PL instructions have their roots in binary image morphology. In this section, we will discuss some of the basics of binary image morphology as well as the general structure of the relevant C4PL instructions.

2.1 Binary Morphology

Binary morphology¹ consists of a set of operations used to study and change geometric properties in binary images. It can be used to find, enhance and/or remove certain geometric features in an image. For example, it can be used to look for corners, close small gaps in an object, etc. The most important operations for our purposes are dilation, erosion and the hit-and-miss transform. All of these operations are based on operations on two or more sets. One of these sets is the image on which we are operating and the members of the set represent the *on* pixels in the image; the other sets are referred to as structuring elements. In general, a structuring element can be any size and have its origin at any location within its bounds. A possible 3x3 structuring element is shown in Figure 1.

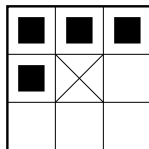


Figure 1: Possible structuring element used in binary morphology. The X indicates the origin of the image, and black squares indicate *on* pixels.

Elements in the sets are represented by ordered pairs in a two dimensional Euclidean space (E^2). An ordered pair indicates the position relative to the origin of the image. For example, the structuring element in Figure 1 would be represented by the set $\{(-1,0),(-1,1),(0,1),(1,1)\}$.

The following sections will briefly describe the purpose and operation of the three morphological operations mentioned above. For a more detailed discussion of binary morphology, please refer to [3] or [4].

2.1.1 Dilation

Dilation, denoted by the operator \oplus , combines the image set with the structuring element set using Minkowski addition²,

¹Morphology is derived from *morph* meaning shape and *-ology* meaning the study of. Thus, morphology means the study of shape.

²Minkowski addition is an element wise addition. For example $(1, 2) + (3, 4) = (1 + 3, 2 + 4) = (4, 6)$.

pairwise on all elements of one set with all elements of the other. This is described mathematically as:

$$A \oplus B = \{c \in E^2 : c = a + b, a \in A \text{ and } b \in B\}$$

As an example, consider the following sets:

$$\begin{aligned} A &= \{(1, 3), (1, 2), (2, 2), (2, 1)\} \\ B &= \{(-1, 0), (0, 0), (1, 0)\} \\ A \oplus B &= \{(0, 3), (0, 2), (1, 2), (1, 1), (1, 3), \\ &\quad (2, 2), (2, 1), 2, 3), (3, 2), (3, 1)\} \end{aligned}$$

This is illustrated graphically in Figure 2(a).

Dilation is generally used to fill small gaps in images, or to grow objects. It can also be used in conjunction with other operations to remove noise, etc.

2.1.2 Erosion

Erosion, denoted by \ominus , is the dual of dilation. It combines the image and structuring element sets using a subtraction-like operation. The operation is described mathematically as:

$$A \ominus B = \{c \in E^2 : c + b \in X \text{ for every } b \in B\}$$

In practice, this means that the structuring element is compared against every pixel in the image. If all *on* pixels in the structuring element are also *on* in the image, then the pixel under the origin of the structuring element is *on* in the output image. An example of erosion is given in Figure 2(b).

2.1.3 Hit-and-Miss

The hit-and-miss transform is used to select pixels that have particular geometric properties. The hit-and-miss transform can be computed many different ways. The most intuitive way to think of this transform is to look at it as a specialized template match. Along with checking whether some points belong to the image set, we can also check to make sure that other points do not belong to that set. The template is specified by two sets, B_1 and B_2 . The B_1 set specifies positions which must match *on* pixels and the B_2 set specifies positions which must match *off* pixels. The output set contains all those points which match both the B_1 template with *on* pixels and the B_2 template with *off* pixels. We describe this mathematically as follows:

$$A \otimes (B_1, B_2) = \{x : B_1 \subset A \text{ and } B_2 \subset A^c\}$$

Since the elements of the B_1 and B_2 sets are required to be mutually exclusive, it is possible to specify the set of templates as a single template where each location has one of

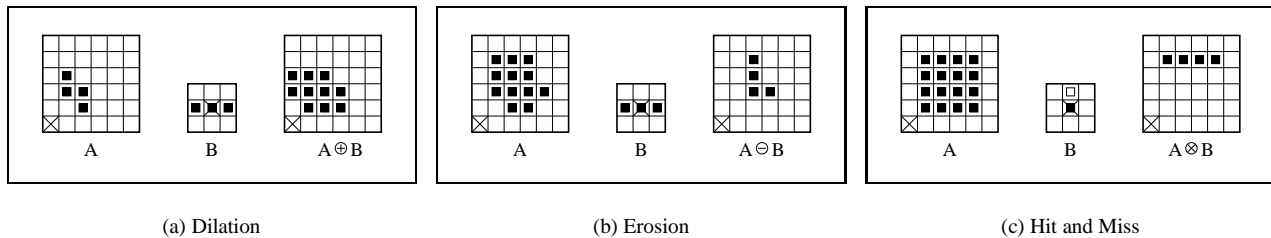


Figure 2: Binary Morphology Examples. Dark squares denote *on* pixels. The white square in the hit-and-miss template denotes a pixel which must match an *off* pixel.

three possible values: *on*, *off* and *don't care*. Pixels in the B_1 set are marked as *on*, and pixels in the B_2 set are marked as *off*. All others are *don't care*. We call this combined template B .

An example of a hit-and-miss transform is given in Figure 2(c). The template in this example picks out pixels along upper edges.

2.2 C4PL Instructions

As with morphological operations, C4PL instructions aim to modify or enhance certain geometric properties. The C4PL instructions are both a generalization and a narrowing of binary image morphology. They are more narrow than binary morphology in that all C4PL instructions use only 3x3 templates with the origin in the center. This limits the size of patterns that can be matched, but still leaves enough flexibility for the problem at hand.

However, C4PL instructions are much more general in that an image may be separated into more than two sets. In binary morphology, an image is divided into two sets, an *on* set and an *off* set; CYTO allows the image to be split into multiple sets. This means that each pixel in an image is assigned to one of many possible sets. In this context, we refer to these different sets as planes or states. For example, referring to plane 3 would refer to all pixels in the image which have been assigned to the set labeled 3.

There are two main ways to represent the assignment of pixels to the different planes. The first and most straightforward approach, called pixel-level encoding, is to use a single multi-bit value for each pixel. The value representing the pixel would be set to the plane number to which that pixel has been assigned. This method was used by the ERIM CYTO Computer, and has the advantage of being compact, but the disadvantage of requiring the instructions to decode the pixel to determine if that pixel is or is not in a particular set.

The second approach, referred to as plane-level encoding, is used by the Sandia software implementation and uses multiple mutually exclusive single-bit (binary) images. Each binary image represents one of the possible sets to which the pixels can be assigned, and each pixel in the image has a corresponding pixel in each of these binary images. While this approach takes more memory to represent a given image, it

is more computationally efficient for software, as each plane can be packed into the native word size of the microprocessor, thus allowing the microprocessor to operate on multiple pixels at a time.

The FOA algorithms use only a small subset of the available C4PL instructions. The operations used in FOA scripts as well as a brief description of their functions are as follows:

- *span*: Dilation operation which uses a 3x3 structuring element filled with 1's.
- *spandisk*: Dilation operation which alternates using an 8-way connected neighborhood and a 4-way connected neighborhood as its structuring element.
- *tranb*: Dilation or erosion using user specified structuring elements.
- *skelrec8*: Skeletonization operation.
- *cover*: Moves pixels from one set into another.
- *exch*: Exchanges two sets.

The above instructions can be grouped into two different types of operations: neighborhood and scalar. Neighborhood operations look at a pixel as well as its eight neighbors and act dependent on some template criteria in the neighborhood being met. Scalar operations act on a single pixel at a time and are unconditional.

Since a general knowledge of the structure of these operations is necessary to understand the motivation for our implementation, we will show the general form of each type of instruction and give a detailed description of one instruction of each type.

2.2.1 Neighborhood Operations

We will first look at a simple example of a neighborhood operation, then we will look at the general form of this class of instructions. The *span* instruction is a conditional dilation and has the following form:

span f m o iter

The *span* operation simply passes through a pixel unchanged unless the following conditions are met: First, the pixel belongs to plane m , and second, at least one of the

pixel's neighbors is in plane f . If these conditions are met, then the pixel is moved to plane o . This operation is repeated $iter$ times.

All neighborhood operations are similar to the *span* operation, in that they simply pass a pixel through unchanged unless a certain set of criteria is met. This criteria is very similar to that used to determine if a pixel is turned *on* in the output of a hit-and-miss transform. However, where the hit-and-miss transform uses a single template, a neighborhood operation can use any number of templates. The neighborhood is said to match if any of the templates matches. Not only must one of the templates match, but the pixel of interest must also be in the specified plane. Thus, a neighborhood instruction only effects pixels in a single plane.

2.2.2 Scalar operations

The scalar operations in the portion of C4PL we implement are very simple operations. These operations depend only upon the value of a single pixel and can typically be implemented with a single gate. For example, the instruction *cover i o* would move any pixel in plane i into plane o . This is essentially the *or* of these two planes.

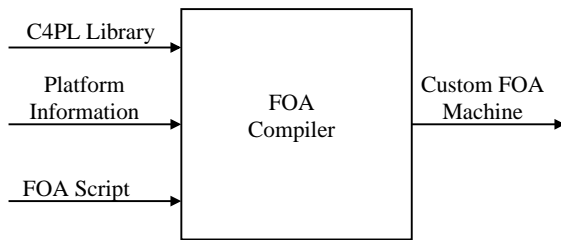


Figure 3: General approach for the FOA compiler.

3 General Approach

Our general approach to the problem was to create a compiler that would generate a highly parallel, deeply pipelined circuit which implements a specified FOA script. This approach is described graphically in Figure 3. The compiler takes as input a C4PL instruction library, information on the platform to be compiled to and the FOA script of interest. The road to our final solution led through three evolutionary steps. The compilers progress from generating highly optimized circuits with slow compile times to creating a more generic circuit with extremely fast compile times. Each of these steps uses a slightly different form of the three inputs.

The circuits generated by the compiler are capable of accepting and processing a new pixel every clock cycle. This is done by streaming the image into the circuit in raster order starting in the upper left hand corner of the image and proceeding line by line. Each of the circuits has the same

general structure. Each one uses delay lines to generate a 3x3 neighborhood. The contents of this neighborhood are passed to some calculation logic which actually computes the result of the instruction. This structure is show in Figure 4.

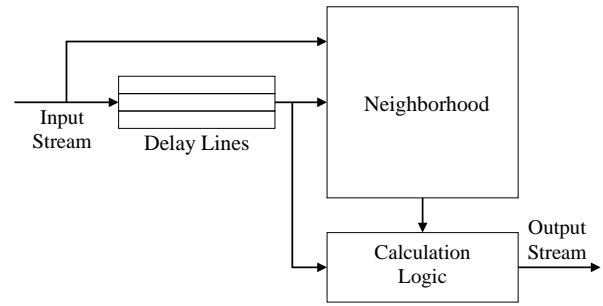


Figure 4: General structure of the FOA circuits.

A consequence of using delay lines to generate the neighborhoods is the need to “pad” the image. Padding the image consists of marking the end of each line so that the hardware knows where one line ends and a new line begins.

The following section will discuss the two types of circuits we used in our compiler implementations. The next section will then discuss the three FOA compilers in greater detail.

4 FOA Circuits

We took two separate approaches to designing FOA circuits. The first approach was to create an optimized circuit for each C4PL instruction; the second approach uses a more generic circuit which has all the circuitry needed for all instructions and is easily specialized for specific instructions. The following sections will discuss the general structure of these two approaches, as well as approximate design times for each. Specific information about a circuit implementation will be discussed with the compiler(s) which use the circuit.

4.1 Instruction Specific Circuits

The instruction specific circuits consist of a different circuit description for each C4PL instruction used in writing FOA scripts, both neighborhood and scalar operations. The scalar operations can be implemented in a single gate or less. The neighborhood operations are all similar and have the general structure shown in Figure 4. The main difference in each instruction is the logic found in the box labeled calculation logic (see Figure 4). Each instruction uses the minimum amount of logic to implement the desired function. In this way, a given FOA script can be implemented in the smallest circuit area possible.

The library of instruction specific circuits was created, optimized and verified by two undergraduate students in four

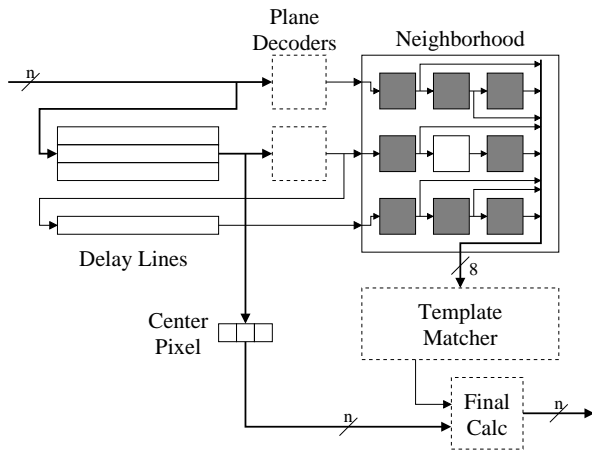


Figure 5: Block diagram of generic C4PL instruction (SuperCell). Bold lines represent multi-bit signals and regular lines represent single bit signals. Elements drawn with dashed lines are the reconfigurable elements which are modified to create specific instructions.

months time. Much of this time is due to the general complexity of writing parameterized VHDL, as well as gaining an understanding of the algorithm.

4.2 Generic C4PL Circuit

The main feature of this type of circuit is that all instructions are represented by a single circuit with configurable elements which are used to implement specific instructions. Using a single generic circuit, instead of many custom circuits, allowed a single graduate student to design and optimize the first implementation of the circuit in only two weeks time, which is one-eighth of the time needed to generate all the instruction specific circuits.

In order to create a Generic C4PL Instruction (which we call a *Supercell*), it was necessary to identify the common features in all the instructions used to write FOA scripts. As was discussed in Section 2.2.1, all neighborhood instructions must do the following:

1. generate an 8-way connected neighborhood,
2. retrieve the value of the center pixel of the neighborhood,
3. match any number of templates based on the generated neighborhood, and
4. calculate the output of the instruction based on the output of the template matchers and the value of the center pixel.

Of these four functions, the first two are identical for every neighborhood instruction. In order to change the instruction, only the structure of the last two must change. Since this circuit uses pixel-level encoding, decoders are needed at

the input of the neighborhood generation circuitry in order to pick out pixels from the proper plane. The following sections will describe the six main features of a Supercell as seen in Figure 5. The circuitry can be split up into two main types: fixed circuitry and programmable circuitry.

Fixed Circuitry The majority of the circuit is made up of elements which are identical for all C4PL neighborhood instructions, and therefore need not change in order to implement different instructions. The following are brief descriptions of this circuitry:

- **Delay Lines.** The delay lines are generated using RAM16x1s (16x1 synchronous RAM) primitives and a linear feedback shift-register (LFSR)[5].
- **Neighborhood.** In conjunction with the delay lines, the neighborhood circuitry builds the 8-way connected neighborhood for a pixel. The circuit is built from 9 flip-flops.
- **Center Pixel.** This is simply a register used to delay the center pixel two extra cycles so that it is synchronized with the center pixel in the neighborhood generator.

Programmable Circuitry The programmable circuitry is the heart of this implementation, and allows the generic circuit to implement any neighborhood operation. The circuit uses lookup tables to generate all the programmable circuitry. The generic instruction is made into a specific instruction by modifying the contents of these tables.

As all the programmable elements are built from lookup tables, it is interesting to look at the number of bits needed to implement each unit. This number determines the size of the lookup table as well the number of bits needed to reprogram the generic instructions. Table 1 shows the size of the lookup table required for each element. The following sections will briefly discuss the purpose of each of the programmable circuit elements.

- **Plane Decoders.** The plane decoders convert the multi-bit pixel to a single bit which indicates whether or not the pixel was a member of the designated neighborhood plane.
- **Template Matcher.** The template matcher takes the eight bits surrounding the center pixel and determines if they match any of the templates given in the instruction definition.
- **Final Calculation.** The final calculation unit looks at the output of the template matcher and the value of the center pixel and decides what the output should be.

4.2.1 Scalar Folding.

Since the final calculation is implemented as a lookup table, it is possible to use it to “fold” in any number of scalar operations which follow it. This is easy to see if you consider

Unit	Lookup Table Size		Number of Bits in the Table			
	Input Bits	Output Bits	$n = \text{bits per pixel}$	$n = 3$	$n = 4$	$n = 5$
Template Matcher	8	1	256	256	256	256
Plane Decoder	n	1	2^n	8	16	32
Final Calculation	$n + 1$	n	$n(2^{n+1})$	48	128	320

Table 1: Details of the input and output width of the lookup tables used to implement the programmable circuitry, as well as the number of programming bits needed to program each unit of a Generic C4PL instruction. This number corresponds directly to the size and number of RAMs needed to implement the necessary logic. n represents the number of bits used to represent a pixel.

that any scalar operation can be implemented as an n input n output lookup table (where n is again the number of bits used to represent a pixel). This allows any plane to be mapped to any other plane. The new lookup table is simply the cascaded result of the original final calculation unit followed by the lookup tables associated with each scalar operation.

4.2.2 Control Characters

An advantage of using pixel-level encoding is that we are able to use some of the pixel values as control characters. We need only two control characters: valid character and edge character. The valid character is used to tag the beginning and end of an image. This information is needed by the back-end circuitry so that it can properly write the processed image to memory. The edge character is used to pad the image as discussed in Section 3. We use the two highest possible pixel values for these signals. These values always get passed through instructions unchanged. Further, plane decoders always output zero for either of these values. This ensures that these characters will not effect the processing of the image.

4.2.3 Pixel Size

This implementation allows us to change the number of bits used to represent a pixel. Choosing the size requires striking a balance between the number of usable planes and the size of the circuit. Using more bits to represent a pixel gives the FOA programmer a larger number of planes to use, but also greatly increases the size of the circuitry. From discussions with Sandia, we determined that 3 bits per pixel would provide enough available planes for the FOA programmer. So, all of our *Supercell* implementations use 3 bits per pixel.

5 FOA Compilers

As previously discussed, we went through three steps to reach our final implementation. The first approach attempted to make the most efficient use of circuit area and therefore used custom instruction specific circuits. This first approach met the desired performance goals, but did not meet the required compile times, so the next two approaches aimed at speeding up the compile time. These approaches used the *Supercell* circuit described in Section 4.2, and introduced a

two-level compilation scheme which would dramatically reduce the in-field compilation time.

The following sections will discuss the three compiler approaches in greater detail. In addition to providing a description of the compilers, we will also compare the implementations with respect to the following areas:

1. Average size (in Xilinx XC4000 CLBs) for a neighborhood operation. The neighborhood size is dependent on the width of the image to be processed. Typically, the circuit would process a 1024 pixel width image. However, because of limited resources on the board, our circuitry was built to handle images with a width of 576 pixels. The image is then broken up and processed separately.
2. Compiler speed.

We will also compare the throughput of our final approach with Sandia’s software implementation.

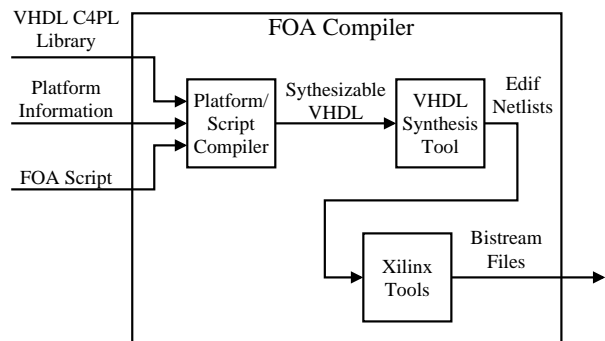


Figure 6: Structure of Direct Synthesis Compiler.

5.1 Direct Synthesis

Our first compiler generated a new custom machine for each FOA script using instruction specific circuitry. The C4PL library for this compiler consisted of synthesizable, parameterized VHDL models for each C4PL instruction of interest. The library also includes the circuits which read image data from memory and write the processed image to memory. The platform information was built directly into the compiler and targeted the Annapolis Microsystems Wildforce board.

This approach can be seen in Figure 6. The output of the compiler was bitstreams which contained designs implementing the input FOA script. To properly create these bitstreams, the compiler was responsible for completing the following:

- **Instruction sequencing.** The main responsibility of the compiler was to instance library elements in the order specified in the FOA script.
- **Chip-level partitioning.** Typical FOA scripts were too large to fit in a single FPGA, so the compiler had to determine how many instructions would fit in an FPGA, and do chip-level partitioning of the design. The compiler would output multiple bitstreams, each one corresponding to a single FPGA.
- **Unused plane synchronization.** Since this implementation used plane-level encoding and planes used in an instruction incur a latency equal to the width of the image being processed plus 2, planes not used in instructions must be delayed this same number of cycles so that the planes remain in synchronization.

Because the number of planes used is dependent on the FOA script, the actual size of the circuit is also script dependent. Typical FOA scripts generated average neighborhood operation sizes of 81 CLBs. Because of the long tool flow, the design did not meet the constraints given us for fast compile times; average compile times were on the order of 10 hours for a typical script; 99+% of that time was spent in the synthesis tool and Xilinx tools.

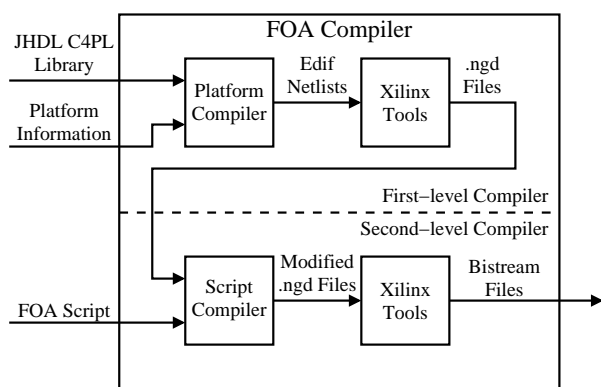


Figure 7: Structure of the Generic FOA Machine compiler.

5.2 Generic FOA Machine

The second approach concentrated on reducing the compile time. At this point, we introduced a two-level compilation scheme as shown in Figure 7. The first stage generates a platform specific, script generic FOA circuit. The generic FOA circuit is made up of a linear array of *Supercells* as discussed in Section 4.2. The second stage uses the input FOA

script to customize the generic machine, by modifying the contents of the lookup tables. The following sections will describe these two compilation stages in greater detail.

5.2.1 First-level Compiler

The first-level compiler used as input a Viewlogic- and VHDL-based module library and targeted the Annapolis Microsystems Wildforce board. The module library included a *Supercell* module designed in Viewlogic, as well as circuitry to convert image data to/from the image streams needed by the circuit, which was written in VHDL. The *Supercells* generated by this compiler use simple ROM cells to implement the lookup tables used for reconfiguration.

The responsibility of the first-level compiler is to generate a generic FOA machine built from *Supercells* which utilizes all the available space on the Wildforce board, given the size of parts available on the board. The output of this stage is placed and routed `.ngd`³ files. To generate the `.ngd` file, the Platform Compiler (see Figure 7) creates an edif netlist for each FPGA on the Wildforce board. These netlists are then passed through the Xilinx tools which generate the `.ngd` file.

This compilation stage takes about 10 hours to complete. However, this stage need only be done once for a given Wildforce board. The `.ngd` files created by this stage are stored and used as input to the next compilation stage.

5.2.2 Second-level Compiler

The inputs to the second-level compiler are the `.ngd` files generated by the first level compiler and the FOA script to be implemented. In order to customize the generic machine created in the first stage, the second-level compiler must complete the following:

- **Determine Programming Bits.** For each neighborhood instruction in the FOA script, the compiler determines the contents of the lookup tables for the plane decoders, template matcher and final calculation unit, which will implement that instruction.
- **Scalar Folding.** Each scalar operation is folded into the final calculation unit of the neighborhood operation preceding it. If there are multiple consecutive scalar operations, they are folded in one at a time.
- **Modify `.ngd` Files.** The script compiler (as seen in Figure 7) parses the `.ngd` files looking for the proper ROM cells to modify. As it finds each ROM, it modifies its contents to reflect the operations specified in the FOA script.
- **Generate Bitstreams.** After the script compiler modified the `.ngd` files, they are passed through the remaining Xilinx tools to generate bitstreams.

³A `.ngd` file is a Xilinx proprietary file which represents mapped, placed and/or routed designs targeting Xilinx FPGAs.

This compilation stage takes, on average, 30 minutes to complete, about 95% of this time is spent in the Xilinx tools. Although this method was much faster than our previous approach, it has one large disadvantage: the .ngd file is a proprietary Xilinx file, and we have no guarantee that the format will not be changed in the future. If this file format was changed, the second-level compiler may no longer work correctly.

5.2.3 Compiler Speed and Circuit Size

As only the second stage was needed to customize a circuit in the field, this approach greatly decreased our “in-field” compile time; however, to accomplish this, we had to sacrifice some circuit size. Average compile time for this approach was about 30 minutes, a 20 times decrease over our previous approach. However, the size of a neighborhood operation grew to 88 CLBs, a 8.6% increase over the direct synthesis approach. Although the compile times were well within the maximum dictated by Sandia, the uncertainty of future compatibility with the .ngd file format made us look at other options.

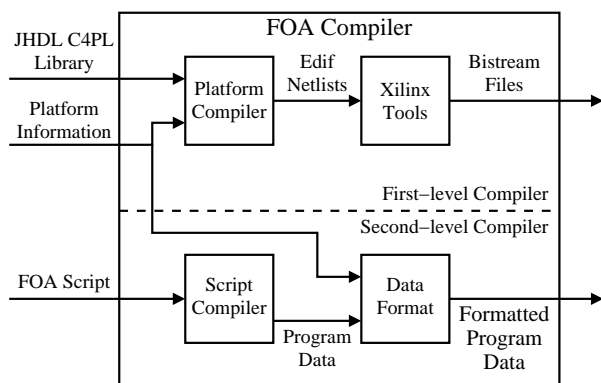


Figure 8: Structure of the Generic FOA Machine with Runtime Reconfiguration compiler.

5.3 Generic FOA Machine with Runtime Reconfiguration

Our final approach is similar to that discussed in Section 5.2. The main difference is that the ROMs containing the computation logic of the *Supercell* were converted to RAM cells so that the contents could be changed at runtime. This simple change allowed us to move from device-level configuration to user-level configuration. This meant that we could use fine-grain logic reconfiguration to change the function of our circuit at runtime.

The first-level compiler creates bitstreams which implement the generic FOA machine. The second-level compiler generates the programming bits needed to program this machine for a specific script. This compiler structure can be seen

in Figure 8. The remainder of this section will discuss each part of this two-level compilation scheme in greater detail.

5.3.1 First-level Compiler

The first-level compiler is responsible for creating the generic framework for a specific hardware platform. The compiler originally targeted the Wildforce board, as did the other two compilers, but was later abstracted so that it might support other boards as well, although we did not retarget the compiler to any other XC4000 based configurable computing boards. The C4PL library for this compiler was written in JHDL[6, 7] and included the *Supercell* module generator, as well as the circuits used to convert image data to/from a raster stream readable by the FOA circuit.

The *Supercell* used by this compiler was slightly different than that used in the previous compiler. There are two major differences:

1. **Runtime Programmable Lookup Tables.** This version of *Supercell* used RAM cells instrumented such that they could operate in two modes: programming mode and lookup table mode. This was accomplished by inserting circuitry as shown in Figure 9.
2. **Cell ID.** Each *Supercell* is assigned a unique ID so that it is possible for the hardware to determine which programming bits belong in which *Supercell*. Programming bits will be ignored by all *Supercells* except the one which matches the specified ID. For more information, see Section 5.3.2.

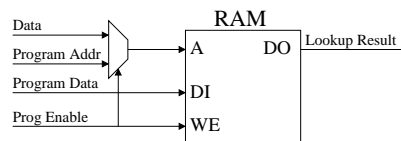


Figure 9: RAM cell instrumented to be a runtime programmable look-up table. Program mode is entered by asserting the Prog Enable signal. This will assert the write enable signal of the RAM as well as force the mux to pass the Program Addr signal to the address pins of the RAM.

The output of the first-level compiler is a set of bitstreams which implement a generic FOA machine capable of reading programming bits from memory and using this data to reprogramming itself for a specific FOA script. This process can take several hours to complete, but again, it need only be done once for each platform.

5.3.2 Second-Level Compiler

The second-level compiler is responsible for computing the programming bits needed to represent a given FOA script. It

is also responsible for generating any control signals needed for programming. The second level compiler is accomplished in two stages. The first stage is platform independent and is responsible for the following:

- **Determine Programming Bits.** The compiler parses the FOA script and determines the contents of the lookup tables in each *Supercell*.
- **Scalar Folding.** The compiler folds scalar operations into the final calculation unit of the neighborhood operation preceding it.
- **Generate Intermediate Format.** The first stage outputs an intermediate format, either as a file or a data structure, which simply contains the contents of the lookup tables that would implement the given FOA script.

The second stage takes the intermediate format produced by the first stage and information on the target platform and then formats the programming data, adding extra control/addressing data as necessary. This output is loaded into a memory on the board and the circuit uses it to reprogram itself. For the Wildforce board, we used a programming word as shown in Figure 10. Each control word specifies which cell (by ID number), element (plane decoder, template matcher, or final calculation) and address within the lookup table the data is intended for.

Both stages of the second-level compiler are completed in 1-2 seconds, depending on the size of the script.

Cell ID	Element	Addr	Programming Data
8	3	4	16

Figure 10: Programming format for the Wildforce implementation of FOA. The numbers represent the number of bits in each field.

5.3.3 Circuit Size and Compiler Speed

Because of the extra circuitry needed for the runtime reprogramming of the instruction pipeline, the size of a neighborhood operation grew 19.3% over the size for the last approach to 105 CLBs. However, by sacrificing some circuit area we were able to reach compile times of 1-2 seconds. This is a 900 times improvement over the previous two-level compilation scheme.

This improvement in compiler speed was made possible by the move away from device-level configuration to user-level configuration. This change was important because of the proprietary format of the `.ngd` file and other intermediate files. Because we cannot easily generate or modify a design at the bitstream level, device-level configuration requires us

to use vendor tools to generate the bitstreams. Even if it were possible to modify the bitstream directly, we would need to map all the logical elements in the circuit to physical locations in the bitstream.

For our purposes, user-level configuration also gives us another advantage: With user-level configuration, the vendor tools are not needed to reprogram the machine for a specific script, and it may not be easy to make the vendor tools readily available once the platform is deployed. User-level configuration in this manner is only feasible for the application, because we did not need to change any of the routing in the circuit. Since the routing programming bits are not user accessible at runtime, any routing changes would need to be implemented with tri-state or muxes, which would require extra circuitry.

Moving to user-level configuration also allowed us to reduce the size of the data used to represent the essential data of an FOA script. In the first approach, the script was represented by the complete set of bitstreams used to implement the script. This means that we required 7,169,020 bits to represent a script ⁴. The subsequent approaches represented the pertinent information as the contents of the lookup tables. Each Supercell can be reprogrammed using 312 bits. An average script uses about 125 Supercells; this would correspond to 39,000 bits. This is only 0.5% of the size needed to represent a script with the first approach.

5.3.4 Hardware vs Software Implementation

All of our approaches ran at about the same clock frequency. The last approach ran slightly slower because of the muxes needed on the input to all the programmable cells. The Xilinx tools reported that our final implementation would operate at 50 MHz. However, the memories on the Wildforce board we were using were limited to operating at 46 MHz, so the circuit was never tested faster than this.

Because the circuit is pipelined and capable of processing a new pixel every clock cycle, the 46 MHz circuit would have a peak throughput of 46 MegaPixel per second. Average throughput, taking into account the control overhead is about 44 MegaPixel per second. This throughput is the same no matter what FOA script is used. The software written by Sandia does not exhibit this same behavior. Because of the inherently sequential nature of the software implementation, the throughput is script dependent. On average, we measured the software implementation to get 4.3 MegaPixel per second on typical length scripts using a G4 PowerPC operating at 450 MHz. We chose to compare to this processor, as it is easily embeddable. For comparison, Table 2 includes numbers for other common microprocessors. This means that the hardware implementation has a 10X performance increase over the software implementation, for average size scripts.

⁴An XC4062 bitstream contains 1,433,804 bits[8], and our designs used 5 such FPGAs

Approach	Neighborhood Size (In CLBs)	Size Increase	Compile Time	Compiler Speedup
Direct Synthesis	81	—	10 hours	—
Generic Machine	88	8.6%	30 minutes	20X
Generic Machine with runtime config.	105	29.6%	2 seconds	18000X

Table 3: Comparison of approaches for FOA mapping. Size increase and compiler speedup are given relative to the first implementation.

Processor	Throughput (MegaPixel)	Hardware Speedup
G4 PPC @450 MHz	4.3	10X
Pentium III @400 MHz	3.3	13X
Pentium III @750 MHz	4.2	10X

Table 2: Throughput of the software implementation of FOA on different processors. Of these processors, only the G4 is easily embeddable.

6 Summary and Conclusions

This paper discussed three evolutionary approaches to mapping FOA circuits to FPGAs. These approaches targeted increased throughput as well as reasonable compile times. All of the circuits we designed were capable of about 46 MegaPixel throughput. This is a 10 times increase over the fastest software implementation on average sized scripts. Moving from highly optimized designs to more general designs, as well as the introduction of runtime reprogrammable circuit elements allowed us to move from device-level configuration to user-level configuration. This in turn allowed us to bypass all of the back-end vendor tools, increasing compile times by a factor of 18000, while only sacrificing a 29.6% increase in circuit area. Table 3 shows the changes in circuit size and compiler speed for each of the three approaches. Note that these percentage increases would be much smaller for circuits designed for images with a 1024 pixel width, only 17.5%.

The two level compilation scheme we adopted allows us to generate a generic FOA machine which can be reprogrammed in the manner of seconds. This allowed us to generate a circuit with 10 times greater throughput than the software implementation, while still allowing the FOA programmer the same flexibility in quickly changing the script being used.

The two-level compilation scheme achieves both high performance and fast compilation times because it *reuses* a *generic* hardware structure that can be customized to implement a specific FOA script. High performance is achieved because the generic FOA image-processing pipeline has been carefully designed to meet the needs of basic FOA operations. Fast compilation times are achieved because only a very small amount of hardware (RAM content) is modified, based on the operations that comprise the FOA script. This is somewhat analogous to the situation that arises when programming a conventional microprocessor. Here, compilation times are quick (at least relative to general hardware synthesis) because the core hardware (the microprocessor) gets reused in every

program generated by the compiler. Rather than synthesizing new hardware operations each time the compiler is run, a 'C compiler (for example) simply implements a given computation by selecting from a set of previously-implemented operations (microprocessor instructions).

Contrast this with the typical configurable-computing application where the entire hardware organization is usually generated from scratch each time the compiler runs, usually involving some form of hardware synthesis. Thus, fast compilation will always be a challenge for configurable computing because it requires a basic tradeoff between design reuse (to reduce compile time) and hardware customization (to achieve high performance). This paper presented a basic two-level compiler strategy that worked because any of the operations found in an FOA script could be implemented with generic computational modules that require only a small amount of customization. Future applications of configurable computing that require fast compilation times will likely need to develop multi-level compilation schemes similar to that described in this paper in order to achieve reasonable compilation times.

7 Future Work

There are many more optimizations that can be done to the circuit implementation to increase throughput and/or decrease circuit size. We are currently working on porting *Supercell* to Xilinx Virtex family FPGAs, and are looking at optimizations that take advantage of new architectural features. We are also looking at modifying the circuit to accept more than one pixel per clock cycle. We have determined that such a modification would add only a small amount of circuitry.

We are currently taking advantage of the generic circuit to rewrite the second-level compiler to make it easily extensible by the end-user. This would make it possible for the end user to add instructions to the compiler in a straightforward, intuitive way. All the user need do is extend a class we provide and give us the details of the neighborhood plane, templates and final calculation. This is an important step forward, as adding instructions to the direct synthesis approach required circuit design. Now, the user need only describe instructions in software, in terms with which they are familiar.

We are also looking into other algorithms to see if the two-level compiler approach we used here would have application to other types of algorithms.

References

- [1] S. R. Sternberg, "Automatic image processor." U.S. Patent 4,167,728, September 1979.
- [2] ERIM (Environmental Research Institute of Michigan), Ann Arbor, Michigan, *C4PL Advanced Ptogramming Manual*, 3 ed., February 1993.
- [3] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis, and Machine Vision*. PWS Publishing, 1999.
- [4] R. M. Haralick and L. G. Shapiro, *Computer and Robot Vision*, vol. Volume I. Addison-Wesley Publishing Company, 1992.
- [5] P. Alfke, "Efficient shift registers, lfsr counters, and long pseudo-random sequence generators," Tech. Rep. XAPP 052, Xilinx, San Jose, CA, July 1996.
- [6] P. Bellows and B. L. Hutchings, "JHDL - an HDL for reconfigurable systems," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (J. M. Arnold and K. L. Pocek, eds.), (Napa, CA), pp. 175–184, Apr. 1998.
- [7] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A cad suite for high-performance fpga design," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (K. L. Pocek and J. M. Arnold, eds.), (Napa, CA), p. n/a, IEEE Computer Society, IEEE, April 1999.
- [8] Xilinx, *The Programmable Logic Data Book*, 1999.