CHAPTER 28

AUTOMATIC TARGET RECOGNITION SYSTEMS ON RECONFIGURABLE DEVICES

Young H. Cho
Open Acceleration Systems Research

An Automatic Target Recognition (ATR) system analyzes a digital image or video sequence to locate and identify all objects of a certain class. There are several ways to implement ATR systems, and the right one is dependent, in large part, on the operating environment and the signal source. In this chapter we focus on the implementations of reconfigurable ATR designs based on the algorithms from Sandia National Laboratories (SNL) for the U.S. Department of Defense Joint STARS airborne radar imaging platform. STARS is similar to an aircraft AWACS system, but detects ground targets.

ATR in Synthetic Aperture Radar (SAR) imagery requires tremendous processing throughput. In this application, data come from high-bandwidth sensors, and the processing is time critical. On the other hand, there is limited space and power for processing the data in the sensor platforms. One way to meet the high computational requirement is to build custom circuits as an ASIC. However, very high nonrecurring engineering (NRE) costs for low-volume ASICs, and often evolving algorithms, limit the feasibility of using custom hardware. Therefore, reconfigurable devices can play a prominent role in meeting the challenges with greater flexibility and lower costs.

This chapter is organized as follows: Section 28.1 describes a highly parallelizable Automatic Target Recognition (ATR) algorithm. The system based on it is implemented using a mix of software and hardware processing, where the most computationally demanding tasks are accelerated using field-programmable gate arrays (FPGAs). We present two high-performance implementations that exercise the FPGA's benefits. Section 28.2 describes the system that automatically builds algorithm-specific and resource-efficient "hardwired" accelerators. It relies on the dynamic reconfiguration feature of FPGAs to obtain high performance using limited logic resources.

The system in Section 28.3 is based on an architecture that does not require frequent reconfiguration. The architecture is modular, easily scalable, and highly tuned for the ATR application. These application-specific processors are automatically generated based on application and environment parameters. In Section 28.4 we compare the implementations to discuss the benefits and the trade-offs of designing ATR systems using FPGAs. In Section 28.5, we draw our conclusions on FPGA-based ATR system design.

28.1 AUTOMATIC TARGET RECOGNITION ALGORITHMS

Sandia real-time SAR ATR systems use a hierarchy of algorithms to reduce the processing demands for SAR images in order to yield a high probability of detection (PD) and a low false alarm rate (FAR).

28.1.1 Focus of Attention

As shown in Figure 28.1, the first step in the SNL algorithm is a Focus of Attention (FOA) algorithm that runs over a downsampled version of the entire image to find regions of interest that are of approximately the right size and brightness. These regions are then extracted and processed by an indexing stage to further reduce the datastream, which includes target hypotheses, orientation estimations, and target center locations. The surviving hypotheses have the full resolution data sent to an identification executive that schedules multiple identification algorithms and then fuses their results.

The FOA stage identifies interesting image areas called "chips." Then it composes a list of targets suspected to be in a chip. Having access to range and altitude information, the FOA algorithm also determines the elevation for the chip, without having to identify the target first. It then tasks the next stage with evaluating the likelihood that the suspected targets are actually in the given image chip and exactly where.

28.1.2 Second-level Detection

The next stage of the algorithm, called Second Level Detection (SLD), takes the extracted imagery (an image chip), matches it against a list of provided target

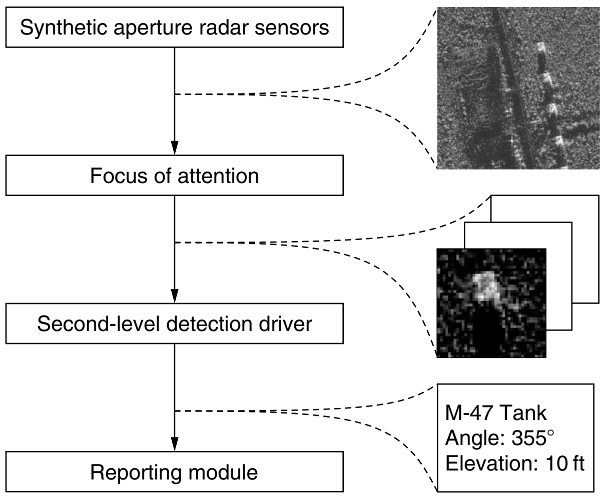

FIGURE 28.1 ■ The Sandia Automatic Target Recognition algorithm.

hypotheses, and returns the hit information for each image chip consisting of the best two orientation matches and other relevant information.

The system has a database of target models. For each target, and for each of its three different elevations, 72 templates are defined corresponding to its all-around views. The orientations of adjacent views are separated by 5 degrees.

SLD is a binary silhouette matcher that has a bright mask and a surround mask that are mutually exclusive. Each template is composed of several parameters along with a “bright mask” and a “surround mask,” where the former defines the image pixels that should be bright for a match, and the latter defines the ones that should not. The bright and surround masks are 32×32 bitmaps, each with about 100 asserted bits. “Bright” is defined relative to a dynamic threshold.

On receiving tasks from the FOA, the SLD unit compares all of the stored templates for this target and elevation and the applicable orientations with the image chip, and computes the level of matching (the “hit quality”). The two hits with the highest quality are reported to the SLD driver as the most likely candidates to include targets. For each hit, the template index number, the exact position of the hit in the search area, and the hit quality are provided. After receiving this information, the SLD driver reports it to the ATR system.

The purpose of the first step in the SLD algorithm, called the shape sum, is to distinguish the target from its surrounding background. This consists of adaptively estimating the illumination for each position in the search area, assuming that the target is at that orientation and location. If the energy is too little or too much, no further processing for that position for that template match is required. Hence, for each mask position in the search area, a specific threshold value is computed as in equation 28.1.

$$SM_{x,y} = \sum_{u=0}^{31} \sum_{v=0}^{31} B_{u,v} M_{x+u,\, y+v} \tag{28.1}$$

$$TH_{x,y} = \frac{SM_{x,y}}{BC} - Bias \tag{28.2}$$

The next step in the algorithm distinguishes the target from the background by thresholding each image pixel with respect to the threshold of the current mask position, as computed before. The same pixel may be above the threshold for some mask positions but below it for others. This threshold calculation determines the actual bright and surround pixel for each position. As shown in equation 28.2, it consists of dividing the shape sum by the number of pixels in the bright mask and subtracting a template-specific *Bias* constant.

As shown in equation 28.3, the pixel values under the bright mask that are greater than or equal to the threshold are counted; if this count exceeds the minimal bright sum, the processing continues. On the other hand, the pixel values under the surround mask that are less than the threshold are counted to calculate the surround sum as shown in equation 28.4. If this count exceeds the minimal surround sum, it is declared a hit.

$$BS_{x,y} = \sum_{u=0}^{31} \sum_{v=0}^{31} B_{u,v} \left[M_{x+u,y+v} \geq TH_{x,y} \right] \tag{28.3}$$

$$SS_{x,y} = \sum_{u=0}^{31} \sum_{v=0}^{31} S_{u,v} \left[M_{x+u,y+v} < TH_{x,y} \right] \tag{28.4}$$

Once the position of the hit is determined, we can calculate its quality by taking the average of bright and surround pixels that were correct, as shown in equation 28.5. This quality value is sent back to the driver with the position to determine the two best targets.

$$Q_{x,y} = \frac{1}{2} \left(\frac{BS_{x,y}}{BC} + \frac{SS_{x,y}}{SC} \right) \tag{28.5}$$

28.2 DYNAMICALLY RECONFIGURABLE DESIGNS

FPGAs can be reconfigured to perform multiple functions with the same logic resources by providing a number of corresponding configuration bit files. This ability allows us to develop dynamically reconfigurable designs. In this section, we present an ATR system implementation of UCLA's Mojave project that uses an FPGA's dynamic reconfigurability.

28.2.1 Algorithm Modifications

As described previously, the current Sandia system uses 64×64 pixel chips and 32×32 pixel templates. However, the Mojave system uses chip sizes of 128×128 pixels and template sizes of 8×8 pixels. It uses different chip and template sizes in order to map into existing FPGA devices that are relatively small. A single template moves through a single chip to yield 14,641 (121×121) image correlation results. Assuming that each output can be represented with 6 bits, the 87,846 bits are produced by the system.

There is also a divide step in the Sandia algorithm that follows the shape sum operation and guides the selection of threshold bin for the chip. This system does not implement the divide, mainly because it is expensive relative to available FPGA resources for the design platform.

28.2.2 Image Correlation Circuit

FPGAs offer an extremely attractive solution to the correlation problem. First of all, the operations being performed occur directly at the bit level and are dominated by shifts and adds, making them easy to map into the hardware provided by the FPGA. This contrasts, for example, with multiply-intensive algorithms that would make relatively poor utilization of FPGA resources. More important, the sparse nature of the templates can be utilized to achieve a far more efficient implementation in the FPGA than could be realized in a general-purpose correlation device. This can be illustrated using the example of the simple template shown in Figure 28.2.

In the example template shown in the figure, only 5 of the 20 pixels are asserted. At any given relative offset between the template and the chip, the correlation output is the sum of the 5 binary pixels in the chip that match the asserted bits in the template. The template can therefore be implemented in the FPGA as a simple multiple-port adder. The chip pixel values can be stored in flip-flops and are shifted to the right by one flip-flop with each clock cycle. Though correlation of a large image with a small mask is often understood conceptually in terms of the mask being scanned across the image, in this case the opposite is occurring—the template is hardwired into the FPGA while the image pixels are clocked past it.

Another important opportunity for increased efficiency lies in the potential to combine multiple templates on a single FPGA. The simplest way to do this is to spatially partition the FPGA into several smaller blocks, each of which handles the logic for a single template. Alternatively, we can try to identify templates that have some topological commonality and can therefore share parts of their adder trees. This is illustrated in Figure 28.3, which shows two templates sharing several pixels that can be mapped using a set of adder trees to leverage this overlap.

A potential advantage FPGAs have over ASICs is that they can be dynamically optimized at the gate level to exploit template characteristics. For our application, a programmable ASIC design would need to provide large general-purpose adder trees to handle the worst-case condition of summing all possible template bits, as shown in Figure 28.4. In constrast, an FPGA exploits the sparse nature of the templates and constructs only the small adder trees required. Additionally, FPGAs can optimize the design based on other application-specific characteristics.

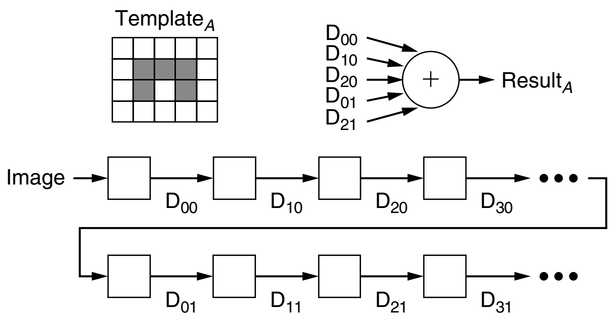

FIGURE 28.2 ■ An example template and a corresponding register chain with an adder tree.

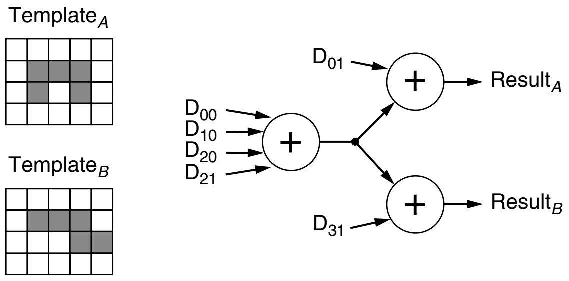

FIGURE 28.3 ■ Common hardware shared between two templates.

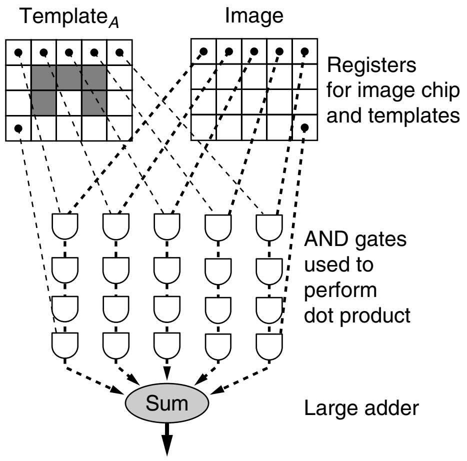
Template$_A$
Image
Registers for image chip and templates
AND gates used to perform dot product
Sum
Large adder

FIGURE 28.4 ■ The ASIC version of the equivalent function.

28.2.3 Performance Analysis

Using a template-specific adder tree achieves significant reduction in routing complexity over a general correlation device, which must include logic to support arbitrary templates. The extent of this reduction is inversely proportional to the fraction of asserted pixels in the template. While this complexity reduction is important, alone it is not sufficient to lead to efficient implementations on FPGAs. The number of D-flip-flops required for storing the data points can cause inefficiencies in the design. Implementing these on the FPGA using the usual flip-flop–based shift registers is inefficient.

This problem can be resolved by collapsing the long strings of image pixels—those not being actively correlated against a template—into shift registers, which can be implemented very efficiently on some lookup table (LUT)–based FPGAs. For example, LUTs in the Xilinx XC4000 library can be used as shift registers that delay data by some predetermined number of clock cycles. Each 16×1-bit

LUT can implement an element that is effectively a 16-bit shift register in which the internal bits cannot be accessed. A flip-flop is also needed at the output of each RAM to act as a buffer and synchronizer. A single control circuit is used to control the stepping of the address lines and the timely assertion of the write-enable and output-enable signals for all RAM-based shift register elements. This is a small price to pay for the savings in configurable logic block (CLB) usage relative to a brute-force implementation using flip-flops.

In contrast, the 256-pixel template images, like those shown in Figure 28.5, can be stored easily using flip-flop–based registers. This is because sufficient flip-flops are available to do this, and the adder tree structures do not consume them. Also, using standard flip-flop–based shift registers for image pixels in the template simplifies the mapping process by allowing every pixel to be accessed. New templates can be implemented by simply connecting the template pixels of concern to the inputs of the adder tree structures. This leads to significant simplification of automated template-mapping tools.

The resources used by the two components of target correlation—namely, storage of active pixels on the FPGA and implementation of the adder tree corresponding to the templates—are independent of each other. The resources used by the pixel storage are determined by the template size and are independent of the number of templates being implemented. Adding templates involves adding new adder tree structures and hence increases the number of function generators being used. The total number of templates on an FPGA is bounded by the number of usable function generators.

The experimental results suggest that in practice we can expect to fit 6 to 10 surround templates having a higher number of overlapping pixels onto a 13,000-gate FPGA. However, intelligent grouping of compatible templates is important. Because the bright templates are less populated than the surround templates, we estimate that 15 to 20 of them can be mapped onto the same FPGA.

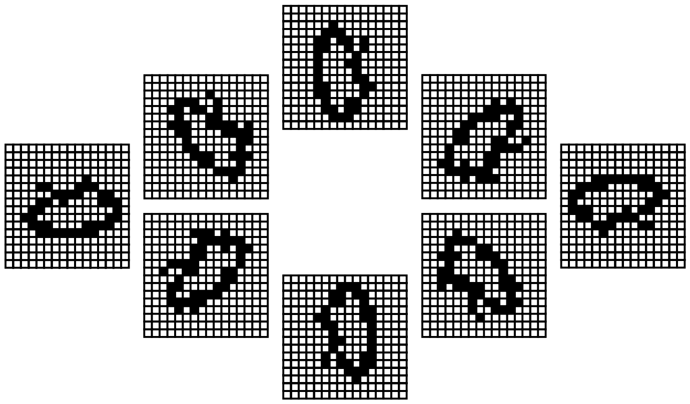

FIGURE 28.5 ■ Example of eight rotation templates of a SAR 16×16 bitmap image.

28.2.4 Template Partitioning

To minimize the number of FPGA reconfigurations necessary to correlate a given target image against the entire set of templates, it is necessary to maximize the number of templates in every configuration of the FPGA. To accomplish this optimization goal, we want to partition the set of templates into groups that can share adder trees so that fewer resources are used per template. The set of templates may number in the thousands, and the goal may be to place 10 to 20 of them per configuration; thus, exhaustive enumeration of all of the possible groupings is not an option. Instead, we use a heuristic method that furnishes a good, although perhaps suboptimal, solution.

Correlation between two templates can establish the number of pixels in common, and it is a good starting point for comparing and selecting templates. However, some extra analysis, beyond iterative correlations on the template set, is necessary. For example, a template with many pixels correlates well with several smaller templates, perhaps even completely subsuming them, but the smaller templates may not correlate with each other and involve no redundant computations. There are two possible solutions to this. The first is to ensure that any template added to an existing group is approximately the same size as the templates already in it. The second is to compute the number of additions required each time a new template is brought in—effectively recomputing the adder tree each time.

Recomputing the entire adder tree is computationally expensive and not a good method of partitioning a set of templates into subsets. However, one of the heuristics used in deciding whether or not to include a template in a newly formed partition is to determine the number of new terms that its inclusion would create in the partition's adder tree. The assumption is that more terms would result in a significant number of new additions, resulting in a wider and deeper adder tree. Thus, by keeping to a minimum the number of new terms created, newly added templates do not increase the number of additions by a significant amount.

Using C++, we have created a design tool to implement the partitioning process that uses an iterative approach to partitioning templates. Templates that compare well to a chosen "base" template (usually selected by largest area) are removed from the main template set and placed in a separate partition. This process is repeated until all templates are partitioned. After the partitions have been selected, the tool computes the adder tree for each partition.

Figure 28.6 shows the creation of an adder tree from the templates in a partition. Within each partition, the templates are searched for shared subsets of pixels. Called *terms*, these subsets can be automatically added together, leading to a template description that uses terms instead of pixels.

The most common addition of two terms is chosen to be grouped together, to form a new term that can be used by the templates. In this way, each template is rebuilt by combining terms in such a way that the most redundant additions are shared between templates; the final result is terms that compute entire templates. For the sample templates shown in Figure 28.6, 39 additions would be required to compute the correlations for all 5 in a naive approach. However,

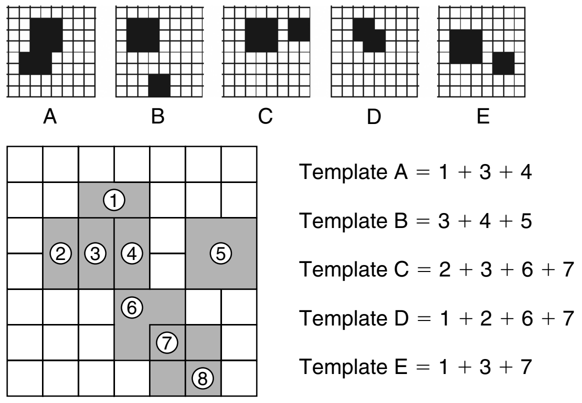

FIGURE 28.6 ■ Example of template grouping and rewritten as sums of terms.

after combining the templates through the process just described, only 17 additions are required.

28.2.5 Implementation Method

For a configurable computing system, the problem of dividing hardware and software is particularly interesting because it is both a hardware and a software issue. Consider the two methods for performing addition shown in Figure 28.7. Method A, a straightforward parallel implementation requiring several FPGAs, has several drawbacks. First, the outputs from several FPGAs converge at the addition operation, which may create a severe I/O bottleneck. Second, the system is not scalable—if it requires more precision, and therefore more bit planes, more FPGAs must be added.

Method B in Figure 28.7 illustrates our approach. Each bit plane is correlated individually and then added to the previous results in temporary storage. It is completely scalable to any image or template precision, and it can implement all correlation, normalization, and peak detection routines required for ATR. One drawback of method B is the cost and power required for the resulting wide temporary SRAM. Another possible drawback is the extra execution time required to run ATR correlations in serial. The ratio of performance to number of FPGAs is roughly equivalent for the two methods, and the performance gap can be closed simply by using more of the smaller method B boards.

The approach of a reconfigurable FPGA connected to an intermediate memory allows us a fairly complicated flow of control. For example, the sum calculation in ATR tends to be more difficult than the image–template correlation. Thus, we may want a program that performs two sum operations and forwards the results to a single correlation.

Reconfigurations for 10K-gate FPGAs are typically around 20 kB in length. Reconfiguring every 20 milliseconds gives a reconfiguration bandwidth of approximately 1 MB per FPGA per second. Coupled with the complexity of the

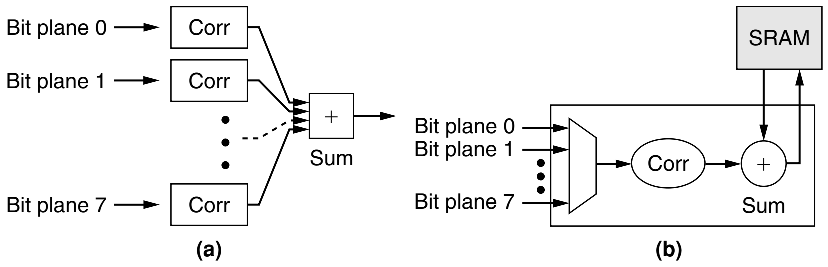

FIGURE 28.7 ■ Each of eight FPGAs correlating each bit plane of the template (a). A single FPGA correlating bit planes and adding the partial sums serially (b).

flow control, this reconfiguration bandwidth can be handled by placing a small microcontroller and configuration RAM next to every FPGA. The microcontroller permits complicated flow of control, and since it addresses the configuration RAM, it frees up valuable I/O on the FPGA. The microcontroller is also important for debugging, which is a major issue in configurable systems because the many different hardware configurations can make it difficult to isolate problems.

The principal components include a “dynamic” FPGA, which is reconfigured on the fly and performs most of the computing functions, and a “static” FPGA, which is configured only once and performs control and some computational functions. The EPROM holds configuration bitstreams, and the SRAM holds the input image data (e.g., the chip). Because the correlation operation involves the application of a small target template to a large chip, a first in, first out (FIFO) is needed to hold the pixels being wrapped around to the next row of the template mask. The templates used in this implementation are of size 8×8, whereas the correlation image is 128×128. Each configuration of the dynamic FPGA implements a total of four template pairs (four bright templates and four surround templates).

The large amount of sum in the algorithm can be performed in parallel. This requires a total of D clock cycles, where D is each pixel’s depth of representation. Once the sum results are obtained, the correlation outputs are produced at the rate of 1 per clock cycle. Parallelism cannot be as directly exploited in this step because different pixels are asserted for different templates. However, in the limit of very large FPGAs the number of clock cycles to compute the correlation is upper-bounded by the number of possible thresholds, as opposed to the number of templates.

28.3 RECONFIGURABLE STATIC DESIGN

Although the idea of reusing reconfigurable hardware to dynamically perform different functions is unique to FPGAs, the main weaknesses of dynamic FPGA reconfiguration are the lengthy time and additional resources required for FPGA reconfiguration and design compilation. Although reconfiguration time has improved dramatically over the years, any time spent on reconfiguration is time that could be used to process more data.

Unlike the dynamic reconfigurable architecture describe in the previous section, we describe another efficient FPGA design that does not require complete design reconfiguration. However, like the previous system, it uses a number of parameters to design a highly pipelined custom design to maximize utilization of the design space to exploit the parallelism in the algorithm.

28.3.1 Design-specific Parameters

To verify our understanding of the algorithm, we first implemented a software simulator and ran it on a sample dataset. Our simulations reproduced the expected results. Over time this algorithm simulator became a full hardware simulator and verifier. It also allowed us to investigate various design options before implementing them in hardware.

The dataset includes 2 targets, each with 72 templates for 5-degree orientation intervals. In total, then, we have 144 bright masks and 144 surround masks, each a 32×32 bitmap. The dataset also includes 16 image chips, each with 64×64 pixels at 1 byte per pixel. Given a template and an image, there are 441 matrix correlations that must take place for each mask. This corresponds to 21 search rows, each 21 positions wide. The total number of search row correlations for the sample data and templates is thus 48,384. The behavior of the simulator on the sample dataset revealed a number of algorithm-specific characteristics. Because the design architecture was developed for reconfigurable devices, these characteristics are incorporated to tune the hardware engine for the best cost and performance.

28.3.2 Order of Correlation Tasks

Correlation tasks for threshold calculation (equation 28.2), bright sum (equation 28.3), and surround sum (equation 28.4) are very closely related. Valid results for all three must exist in order to calculate the quality of the hit, so invalid results from any one of them make other calculations unnecessary.

For the data samples, about 60 percent of the surround sums and 40 percent of the threshold results were invalid, while all of the bright sum results were valid. The low rejection rate by bright sum is the result of the threshold being computed using only the bright mask, regardless of the surround mask. The threshold is computed by the same pixels used for computing bright sum, so we find that, for a typical dataset, checking for invalid surround sums before the other calculations drastically reduces the total number of calculations needed.

Zero mask rows

Each mask has 32 rows. However, many have all-zero rows that can be skipped. By storing with each template a pointer to its first nonzero row we can skip directly to that row "for free." Embedded all-zero rows are also skipped.

The simulation tools showed that, for our template set, this optimization significantly reduces the total computational requirements. For the sample template set, there are total of 4608 bitmap rows to use in the correlation tasks. Out of 4608 bright rows, only 2206 are nonzero, and out of 4608 surround rows, 2815 are nonzero. Since the bright mask is used for both threshold and bright sum calculations, and the surround mask is used once, skipping the zero rows reduces the number of row operations from 13,824 to 7227, which produces a savings of about 52 percent.

It is also possible to reduce the computation by skipping zero columns. However, as will be described in following section, the FPGA implementation works on an entire search row concurrently. Hence, skipping rows reduces time but skipping columns reduces the number of active elements that work in parallel, yielding no savings.

28.3.3 Reconfigurable Image Correlator

Although it is possible to reconfigure FPGAs dynamically, the time spent on context switching and reconfiguration could be used instead to process data on a register-based static design. For this reason, minimizing reconfiguration time during computation is essential in effective FPGA use. Nevertheless, when we use FPGAs as compute engines, reconfiguration allows the hardware to take on a large range of task parameters.

The SLD tasks represented in equations 28.1, 28.3, and 28.4 are image correlation calculations on sliding template masks with radar images. To explain our design strategies, we examine each equation by applying the algorithm on a small dataset consisting of a 6×6 pixel image, a 3×3 mask bitmap, and a 4×4 result matrix.

For this dataset, the shape sum calculation for a mask requires multiplying all 9 mask bits with the corresponding image pixels and summing them to find 1 of 16 results. To build an efficient circuit for the sum equations 28.3 and 28.4, we write out the subset of both equations as shown in Table 28.1. By expanding the summation equations, we expose opportunities for hardware to optimize the calculations. First, the same B_{uv} is used to calculate the nth term of all of the shape sum results. Thus, when the summation calculations are done in parallel, the B_{uv} coefficient can be broadcast to all of the units that calculate each result. Second, the image data in the nth term of the SM_{xy} is in the $(n+1)$th term of SM_{xy-1}, except when v returns to 0, the image pixel is located in the subsequent row. This is useful in implementing the pipeline datapath for the image pixels through the parallel summation units.

TABLE 28.1 ■ Expanded sum equations 28.3 and 28.4

Term	1	2	3	4	5	6	7	8	9
u	0	0	0	1	1	1	2	2	2
v	0	1	2	0	1	2	0	1	2
$SM_{00} =$	$B_{00}M_{00}+$	$B_{01}M_{01}+$	$B_{02}M_{02}+$	$B_{10}M_{10}+$	$B_{11}M_{11}+$	$B_{12}M_{12}+$	$B_{20}M_{20}+$	$B_{21}M_{21}+$	$B_{22}M_{22}$
$SM_{01} =$	$B_{00}M_{01}+$	$B_{01}M_{02}+$	$B_{02}M_{03}+$	$B_{10}M_{11}+$	$B_{11}M_{12}+$	$B_{12}M_{13}+$	$B_{20}M_{21}+$	$B_{21}M_{22}+$	$B_{22}M_{23}$
$SM_{02} =$	$B_{00}M_{02}+$	$B_{01}M_{03}+$	$B_{02}M_{04}+$	$B_{10}M_{12}+$	$B_{11}M_{13}+$	$B_{12}M_{14}+$	$B_{20}M_{22}+$	$B_{21}M_{23}+$	$B_{22}M_{24}$
$SM_{03} =$	$B_{00}M_{03}+$	$B_{01}M_{04}+$	$B_{02}M_{05}+$	$B_{10}M_{13}+$	$B_{11}M_{14}+$	$B_{12}M_{15}+$	$B_{20}M_{23}+$	$B_{21}M_{24}+$	$B_{22}M_{25}$

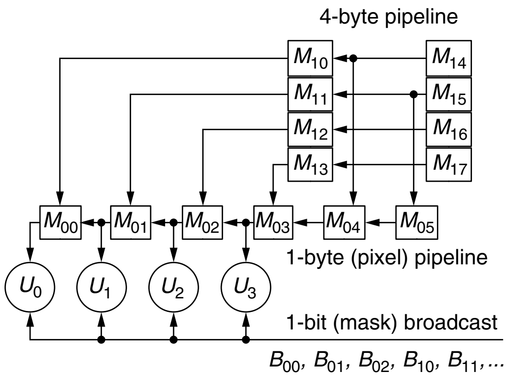

FIGURE 28.8 ■ A systolic image array pipeline.

Based on the characteristics of the expanded equations, we can build a systolic computation unit as in Figure 28.8. To save time while changing the rows of pixels, the pixel pipeline can either operate as a pipeline or be directly loaded from another set of registers. At every clock cycle, each U_y unit performs one operation, v is incremented modulo 3, and the pixel pipeline shifts by one stage (U_1 to U_0, U_2 to $U_1, \ldots$). When v returns to 0, u is incremented modulo 3, and the pixel pipeline is loaded with the entire $(u+x)$th row of the image. When u returns to 0, the results are offloaded from the U_y stage, their accumulators are cleared, and x is incremented modulo 4. When x returns to 0, this computing task is completed.

The initial loading of the image pixel pipeline is from the image word pipeline, which is word wide and so four times faster than the image pixel pipeline. This speed advantage guarantees that the pipeline will be ready with the next image row data when u returns to 0.

28.3.4 Application-specific Computation Unit

Developing different FPGA mappings for equations 28.1, 28.3, and 28.4 in parallel processing unit is one way to implement the design. At the end of each stage, the FPGA device is reconfigured with the optimal structure for the next task. As appealing as this may sound, current FPGA devices have typical reconfiguration times of tens of milliseconds, during which the reconfiguring logic cannot be used for computation.

As presented in Section 28.3, each set of template configurations also has to be designed and compiled before any computation can take place. This can be a time-consuming procedure that does not allow dynamic template sets to be immediately used in the system.

Fortunately, we can rely on the fact that FPGAs can be tuned to target-specific applications. From the equations, we derived one compact structure, shown in Figure 28.9, that can efficiently perform all ATR tasks. Since the target ATR

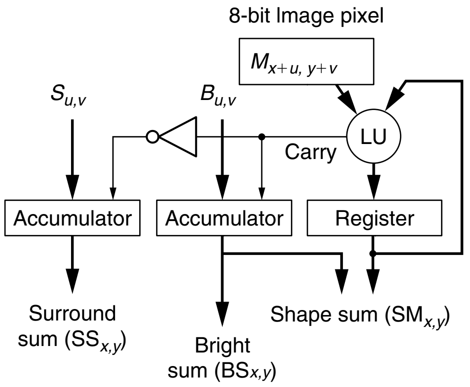

FIGURE 28.9 ■ Computation logic for equations 28.1, 28.3, and 28.4.

system can be seen as "embarrassingly parallel," the performance of the FPGA design is linearly scalable to the number of the application-specific units.

28.4 ATR IMPLEMENTATIONS

In this section we present the implementation results of two reconfigurable Sandia ATR systems, researched and developed on different reconfigurable platforms. Both designs leverage the unique characteristics of reconfigurable devices to accelerate ATR algorithms while making efficient use of available resources. Therefore, they both outperformed existing software as well as custom ASIC solutions. By analyzing the results of the reconfigurable solutions, we examine design trade-offs in cost and performance.

28.4.1 A Dynamically Reconfigurable System

All of the component technologies described in this chapter have been designed, implemented, tested, and debugged using the Mojave board shown in Figure 28.10. This section discusses various performance aspects of the complete system, from abstract template sets through application-specific CAD tools and finally down to the embedded processor and dynamic FPGA. The current hardware is connected to a video camera rather than a SAR data source, though this is only necessary for testing and early evaluation.

The results presented here are based on routing circuits to two devices: the Xilinx 4013PG223-4 FPGA and the Xilinx 4036. Xilinx rates the capacity of these parts as 13K and 36K equivalent gates.

Table 28.2 presents data on the effectiveness of the template-partitioning phase. Twelve templates were considered for this comparison: in one case they were randomly divided into three partitions; in the other, the CAD tool was used to guide the process. The randomly selected partitions required 33 percent more CLBs than those produced by the intelligent partitioning tool. These numbers

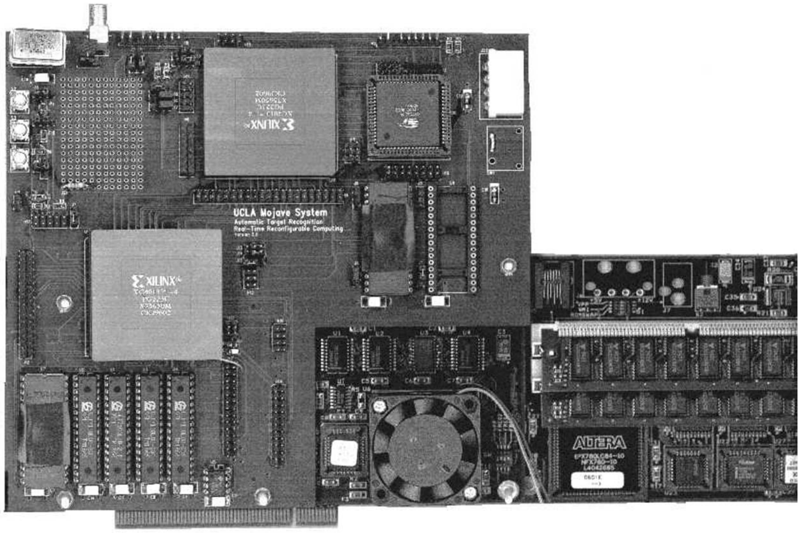

FIGURE 28.10 ■ Photograph of second-generation Mojave ATR system.

Table 28.2 ■ Comparison of scored and random partitioning on an Xilinx 4036

Random grouping CLB count	Initial partitioning CLB count
1961	1491
1959	1449
1958	1487

Table 28.3 ■ Comparison of resources used for the dynamic and static FPGAs

	Flip-flops	Function generators	I/O pins
Dynamic FPGA	532	939	54
Support FPGA	196	217	96
Available	1536	1536	192

account for the hardware requirements of the entire design, including the control hardware that is common to all designs as well as the template-specific adder trees. Relative savings in the adder trees alone are higher.

Table 28.3 lists the overall resources used for both FPGAs in the system, the dynamic devices used for correlation, and the static support device used to implement system control features. Because the image source is a standard video camera rather than a SAR sensor, the surround template is the complement of the bright template, resulting in more hardware than would be required for true SAR templates. The majority of the flip-flops in the dynamic FPGA are assigned to holding the 8-bit chip data in a set of shift registers. This load increases as a linear function of the template size.

Each configuration of the dynamic FPGA requires 16 milliseconds to complete an evaluation of the entire chip for four template pairs. The Xilinx 4013PG223-4 requires 30 milliseconds for reconfiguration. Thus, a total of 4 template pairs can be evaluated in 46 milliseconds, or 84 template pairs per second. This timing will increase logarithmically with the template size.

Comparing configurable machines with traditional ASIC solutions is necessary but complicated. Clearly, for almost any application, a bank of ASICs could be designed that used the same techniques as the multiple configurations of the FPGA and would likely achieve higher performance and consume less power. The principal advantage of configurable computing is that a single FPGA may act as many ASICs without the cost of manufacturing each device. If the comparison is restricted to a single IC (e.g., a single FPGA against a single ASIC of similar size), relative performance becomes a function of the hardware savings enabled by data specificity. For example, in the ATR application the templates used are quite sparse—only 5 to 10 percent of the pixels are important in the computation—which translates directly into a hardware savings that is much more difficult to realize in an ASIC. Further savings in the ATR application are possible by leveraging topological similarities across templates. Again, this is an advantage that ASICs cannot easily exploit.

If the power and speed advantages of ASICs over FPGAs are estimated at a factor of 10, the configurable computing approach achieves a factor of improvement anywhere from 2 and 10 (depending on sparseness and topological properties) for the ATR application.

28.4.2 A Statically Reconfigurable System

The FPGA nodes developed by Myricom integrate reconfigurable computing with a 2-level multicomputer to promote flexibility of programmable computational components in a highly scalable network architecture. The Myricom FPGA nodes and its motherboard are shown in Figure 28.11. The daughter nodes are 2-level multicomputers whose first level provides the general-purpose infrastructure of the Myrinet network using the LANai RISC microprocessor. The FPGA functions as a second-level processor responsible for application-specific tasks.

The host is a SparcStation IPX running SunOS 4.1.3 with a Myrinet interface board having a 512K memory. The FPGA node—consisting of Lucent Technologies' ORCA FPGA 40K and Myricom's LANai 4.1 running in 3.3 V at 40 MHz—communicates with the host through an 8-port Myrinet switch.

Without additional optimization, static implementation of the complete ATR algorithm on one FPGA node processes more than 900 templates per second. Each template requires about 450,000 iterations of 1-bit conditional accumulate for the complete shape sum calculation. The threshold calculation requires one division followed by subtraction. The bright and surround sum compares all the image pixels against the threshold results. Next, 1-bit conditional accumulate is

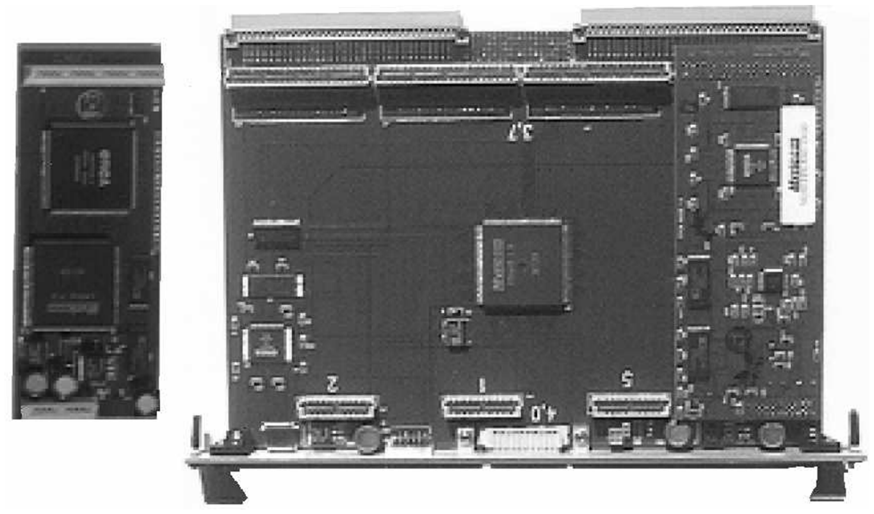

FIGURE 28.11 ■ A Myrinet 8-port switch motherboard with Myricom ORCA FPGA daughter nodes. Four FPGA nodes can be plugged into a single motherboard.

executed for each sum. And then the quality values are calculated using two divides, an add, and a multiply.

Given that 1-bit conditional accumulate, subtract, divide, multiply, and 8-bit compare are one operation each, the total number of 8-bit operations to process one 32×32 template over a 64×64 image is approximately 3.1 million. Each FPGA node executes over 2.8 billion 8-bit operations per second (GOPS).

After the simulations, we found that the sparseness of the actual templates reduced their average valid rows to approximately one-half the number of total template rows. This optimization was implemented to increase the throughput by 40 percent. Further simulations revealed more room for improvements, such as dividing the shape sum in the FPGA, transposing narrow template masks, and skipping invalid threshold lines. Although these optimizations were not implemented in the FPGA, the simulation results indicated an additional 94 percent increase in throughput. Implementing all optimizations would yield a result equivalent to about a 7.75 GOPS correlator.

28.4.3 Reconfigurable Computing Models

The increased performance of configurable systems comes with several costs. These include the time and bandwidth required for reconfiguration, the memory and I/O required for intermediate results, and the additional hardware required for efficient implementation and debugging. Minimizing these costs requires innovative approaches to system design.

Figure 28.12 illustrates the fundamental difference between a traditional computing model and the two reconfigurable computing architectures discussed in this chapter. The traditional processor receives simple operands from data

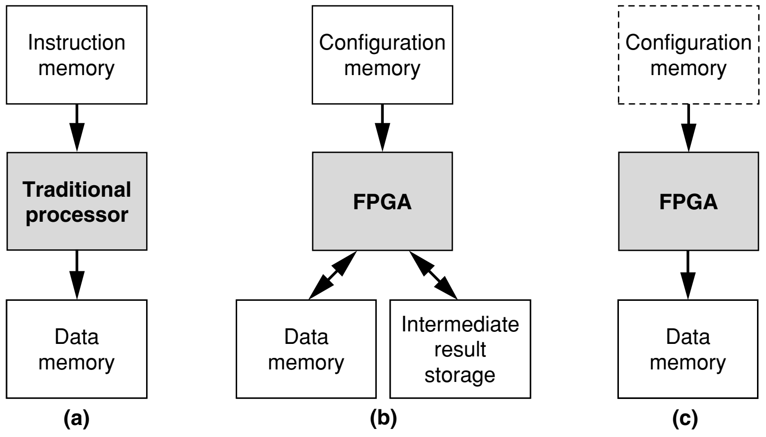

FIGURE 28.12 ■ A comparison of a traditional computing model (a) with a dynamically reconfigurable model (b) and a statically reconfigurable custom model (c).

memory, performs a simple operation in the program, and returns the result to data memory. Similarly, dynamic computing uses a small number of rapidly reconfiguring FPGAs tightly coupled to an intermediate result memory, data memory, and configuration memory. A reconfigurable custom computer is similar to a fixed ASIC device in that, usually, only one highly tuned design is configured on the FPGA—there is no need to reconfigure to perform a needed function.

In most cases, a custom ASIC performs far better than a traditional processor. However, traditional processors continue to be used for their programmability. FPGAs attempts to bridge the gap between custom ASICs and software by allowing designers to build custom hardware using programmable firmware. Therefore, unlike in pure ASIC designs, configuration memory is used to program the reconfigurable hardware as instructions in a traditional processor would dictate the functionality of a program. Unlike software, once the FPGA is configured, it can function just like a custom device.

As shown in previous sections, an ATR was implemented in an FPGA using two different methods. The first implementation uses the dynamic computer model, where parts of the entire algorithm are dynamically configured to produce the final results. The second design uses simulation results to produce a highly tuned fixed design in the FPGA that does not require more than a single reconfiguration. Because of algorithm modifications made to the first design, there is no clear way to compare the two designs. However, looking deeper, we find that there is not a drastic difference in the subcomponents or the algorithm; in fact, the number of required operations for the algorithm in either design should be the same.

The adders make up the critical path of both designs. Because both designs are reconfigurable, we expect the adders used to have approximately the same performance as long as pipelining is done properly. Clever use of adders in the static design allows it to execute more than one calculation simultaneously. However, it is possible to make similar use of the hardware to increase performance in the dynamic design.

The first design optimizes the use of adders to skip all unnecessary calculations, also making each configuration completely custom. The second design has to be more general to allow some programmability. Therefore, depending on the template data, not all of the adders may be in use at all times. If all of the templates for the first design can be mapped onto a single FPGA, the first method results in more resource efficiency than the second. The detrimental effect of idle adders in the static design becomes increasingly more prominent as template bitmap rows grow more sparse.

On the other hand, if the templates do not all fit in a single FPGA, the first method adds a relatively large overhead because of reconfiguration latency. Unfortunately, the customized method of the second design works against making the design smaller. Every bit in the template maps to a port of the adder engine, so the total size of the design is proportional to the number of total bits in all of the templates. Therefore, as the number of templates increases, the total design size must also increase. Ultimately, the design must be divided into several smaller configurations that are dynamically reconfigured to share a single device.

From these results, we observe the strengths and weaknesses of dynamic reconfiguration in such applications. Dynamic reconfiguration allows a large custom design to successfully run in a smaller FPGA device. The trade-off is significant time overhead in the system.

28.5 SUMMARY

Like many streaming image correlation algorithms, the Sandia ATR system discussed in this chapter can be efficiently implemented on an FPGA. Because of the high degree of parallelism in the algorithm, designers can take full advantage of parallel processing in hardware while linearly scaling total throughput with available hardware resources. In this chapter we presented two different ways of implementing such a system.

The first system employs a dynamic computing model to effectively implement a large custom design using a smaller reconfigurable device. To fit, high-performance custom designs can be divided into subcomponents, which can then share a single FPGA to execute parts of the algorithm at a high speed. For the ATR algorithm, this process produced a resource-efficient design that exceeded the performance of previous custom ASIC-based systems.

The second system is based on a more generic architecture highly tuned for a given set of templates. Through extensive simulations, many parameters of the algorithm are tuned to efficiently process the incoming data. With algorithm-specific optimizations, the throughput of the system increased threefold from an initial naive implementation. Because of the highly pipelined structure of the design, the maximum clock frequency is more than three times that of the dynamic computer design. Furthermore, a larger FPGA on the platform allowed the generic processing architecture to duplicate the specifications of the original algorithm. Therefore, the raw performance of the static design was faster than the dynamically reconfigurable system.

Although the second system is a static design, it is best suited for reconfigurable platforms because of its highly tuned parameters. Since this system is reconfigurable, it is conceivable that the dynamic computational model can be applied on top of it. Thus, the highly tuned design may be implemented efficiently, even on a device with enough resources for only a fraction of the entire design.

Acknowledgments I would like to acknowledge Professor William H. Mangione-Smith for permission to integrate publications on the Mojave project into this chapter.

References

[1] P. M. Athanas, H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer* 26, 1993.

[2] J. G. Eldredge, B. L. Hutchings. Run-time reconfiguration: A method for enhancing the functional density of SRAM-based FPGAs. *Journal of VLSI Signal Processing* 12, 1996.

[3] J. Villasenor, W. H. Mangione-Smith. Configurable computing. *Scientific American* 276, 1997.

[4] E. Mirsky, A. DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, 1996.

[5] R. Razdan, M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proceedings of the 27th Annual International Symposium on Microarchitecture,* pp. 172–180, 1994.

[6] G. Estrin. Organization of computer systems—the fixed plus variable structure computer. *Proceedings of the Western Joint Computer Conference*, 1960.

[7] M. Shand, J. Vuillemin. Fast implementations of RSA cryptography. *Proceedings of the Symposium on Computer Arithmetic*, 1993.

[8] K. W. Tse, T. I. Yuk, S. S. Chan. Implementation of the data encryption standard algorithm with FPGAs. *Proceedings of International Symposium on Field-Programmable Logic and Applications*, 1993.

[9] J. Leonard, W. H. Mangione-Smith. A case study of partially evaluated hardware circuits: Key-specific DES. *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications* 1304:151–160, 1997.

[10] P. M. Athanas, A. L. Abbott. Real-time image processing on a custom computing platform. *IEEE Computer* 28, 1995.

[11] J. G. Eldredge, B. L. Hutchings. Density enhancement of a neural network using FPGAs and run-time reconfiguration. *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, 1994.

[12] J. G. Eldredge, B. L. Hutchings. RRANN: The run-time reconfiguration artificial neural network. *Proceedings of the Custom Integrated Circuits Conference*, 1994.

[13] B. Schoner, C. Jones, J. Villasenor. Issues in wireless coding using run-time-reconfigurable FPGAs. *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, 1995.

[14] C. Chou, S. Mohanakrishnan, J. B. Evans. FPGA implementation of digital filters. *Proceedings of the Fourth International Conference on Signal Processing Applications and Technology*, pp. 80–88, 1993.

[15] G. Estrin, B. Bussell, R. Turn, J. Bibb. Parallel processing in a restructurable computer system. *IEEE Transactions on Electronic Computers* EC-12(5):747–755, December 1963.

[16] G. Estrin, R. Turn. Automatic assignment of computations in a variable structure computer system. *IEEE Transactions on Electronic Computers* EC-12(6):755–773, December 1963.

[17] M. J. Wirthlin, B. L. Hutchings. Improving functional density through run-time constant propagation. *Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays,* 1997.

[18] P. Lee, M. Leone. Optimizing ML with run-time code generation. *Proceedings of Programming Language Design and Implementation*, 1996.

[19] D. R. Engler, T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Proceedings of the Sixth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1994.

[20] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, Ph.D. thesis, Columbia University, Department of Computer Science, 1992.

[21] W. H. Mangione-Smith, B. Hutchings. Configurable computing: The road ahead. *Proceedings of the Reconfigurable Architectures Workshop*, 1997.

[22] P. Bertin, H. Touati. PAM programming environments: Practice and experience. *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 1994.

[23] Y. H. Cho. Optimized automatic target recognition algorithm on scalable Myrinet/field programmable array nodes. *Thirty-fourth IEEE Asilomar Conference on Signals, Systems, and Computers*, October 2000.

[24] K. N. Chia, H. J. Kim, S. Lansing, W. H. Mangione-Smith, J. Villasenor. High-performance automatic target recognition through data-specific very large scale integration. *IEEE Transactions on Very Large Scale Integration Systems* 6(3), 1998.

[25] J. Villasenor, B. Schoner, K. N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, W. H. Mangione-Smith. Configurable computing solutions for automatic target recognition. *Proceedings of the IEEE International Symposium on FPGAs for Custom Computing Machines*, April 1996.

[26] R. Sivilotti, Y. Cho, D. Cohen, W. Su, B. Bray. Scalable network based FPGA accelerators for an automatic target recognition application. *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[27] R. Sivilotti, Y. Cho, W. Su, D. Cohen. *Scalable, Network-connected, Reconfigurable, Hardware Accelerators for an Automatic-Target-Recognition Application*, Myricom technical report, May 1998.

[28] R. Sivilotti, Y. Cho, W. Su, D. Cohen. *Myricom's FPGA-based Approach to ATR/SLD*, DARPA ACS PI meeting slide presentation, November 1997.

[29] R. Sivilotti, Y. Cho, W. Su, D. Cohen. Production-quality, LANai-4-based quad-FPGA-node VME boards. *http://www.myri.com/research/darpa/97a-fpga.html*, October 1997.

[30] C. L. Seitz, Tactical network and multicomputer technology. *http://www.myri.com/research/darpa/index.html*, March 1997, July 1997, August 1998.

[31] C. L. Seitz. Two-level-multicomputer project: Summary. *http://www.myri.com/research/darpa/96summary.html*, July 1996.

[32] W. C. Athas, L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer* 21, 1988.

[33] M. Shand, J. Vullemin. Fast implementations of RSA cryptography. *Proceedings of 11th Symposium on Computer Arithmetic*, 1993.

[34] J. G. Eldredge, B. L. Hutchings. RRANN: The run-time reconfiguration artificial neural network. *Proceedings of the IEEE Custom Integrated Circuits Conference,* 1994.

[35] Xilinx, Inc. *RAM-based Shift Register v9.0, LogiCORE Datasheet*, Xilinx, Inc., July 13, 2006.