

# Configurable Computing Solutions for Automatic Target Recognition

John Villasenor, Brian Schoner, Kang-Ngee Chia, Charles Zapata,  
Hea Joung Kim, Chris Jones, Shane Lansing, and Bill Mangione-Smith  
Electrical Engineering Department  
University of California, Los Angeles  
Los Angeles, CA 90095-1594

*FPGAs can be used to build systems for automatic target recognition (ATR) that achieve an order of magnitude increase in performance over systems built using general purpose processors. This improvement is possible because the bit-level operations that comprise much of the ATR computational burden map extremely efficiently into FPGAs, and because the specificity of ATR target templates can be leveraged via fast reconfiguration. We describe here algorithms, design tools, and implementation strategies that are being used in a configurable computing system for ATR.*

## 1. Introduction

The ability to rapidly modify the gate level logic of an FPGA during execution opens a number of new computing possibilities that have only recently begun to be explored. Configurable computing machines that exploit this ability will involve architectures that differ in important ways from those used in current machines, and will support a wide range of new and powerful capabilities. At the simplest level, a single FPGA can implement an arbitrary number of designs in rapid succession, and can therefore deliver the functionality of a device many times its size. More sophisticated implementations in which the configuration control receives input from the results of previous computations or from the external operating environment can also be envisioned. Finally, rapid reconfiguration makes feasible the implementation of dedicated logic circuits to support large numbers of highly specific computational tasks that are wholly unsuited to ASIC implementation.

Configurable computing requires 1) commercially available FPGAs with sufficiently fast configuration times, 2) design tools that understand and take advantage of hardware dynamisms, and 3) boards and other higher level interface and support hardware that will make fast-reconfigurable FPGAs viable in real systems. Although there is

not yet a base of commercial FPGAs with submillisecond reconfiguration times to satisfy the first of these requirements, there has been prototype development in both industry and academia to explore the hardware issues of fast reconfiguration. It is our belief that this is an area which the FPGA vendors are well prepared to address, and that fast reconfiguration will receive increased commercial attention as the payoffs on the application side become clear. By contrast, design tools and systems to support use of dynamic computing devices have been in a state of relative immaturity. We describe here results from an ongoing project to build a configurable computing system for automatic target recognition (ATR). Focusing on this application has furnished quantitative results on design time and challenges, configuration overhead, and computational efficiency of configurable computing systems. More generally, it has led to an understanding of the requirements and hurdles involved in extending configurable computing to more general applications.

The rest of this paper is organized as follows: The remainder of the introduction includes a description of related work in configurable computing, and an overview of the automatic target recognition (ATR) problem that serves as the application focus for the new results presented here. Section 2 introduces the basic mapping of ATR target templates into FPGA adder trees that forms the core of the processing, and discusses some of the trade-offs in using rapid reconfiguration to support ATR. Section 3 discusses design and partitioning issues to support rapid implementation of a large set of target templates. Section 4 presents a more detailed analysis of design trade-offs and considers some the issues of I/O and board design for a FPGA-based ATR system. Conclusions and a brief description of ongoing and future work are contained in Section 5.

### 1.1 Overview of related work

Configurable computing has been explored in both academia and industry for several years. On the hardware

side, several fast-reconfiguring FPGAs have been developed. One of these is the Configurable Logic Array (CLay) from National Semiconductor, which is a fine-grained device consisting of an array of 56 by 56 cells, each of which is roughly equivalent to a half adder and D-flip flop. Configuration bitstreams can be loaded into the CLay using 8 pins, allowing a complete reconfiguration of the device in approximately 750 microseconds. The CLay also supports partial reconfiguration, which reduces the reconfiguration time linearly in accordance with the fraction of the gate array concerned.

An alternative to external loading of bitstreams used in the CLay and in most other FPGAs is the context-switched approach advocated by a group at MIT [1,2]. In this device, referred to as a Dynamically Programmable Gate Array (DPGA), multiple configurations reside simultaneously on chip. One of these configurations occupies the active layer, and is the one which is actually executing. Any of the others can be switched to the active layer in one clock cycle. The increase in chip area to support three extra contexts is approximately 20%.

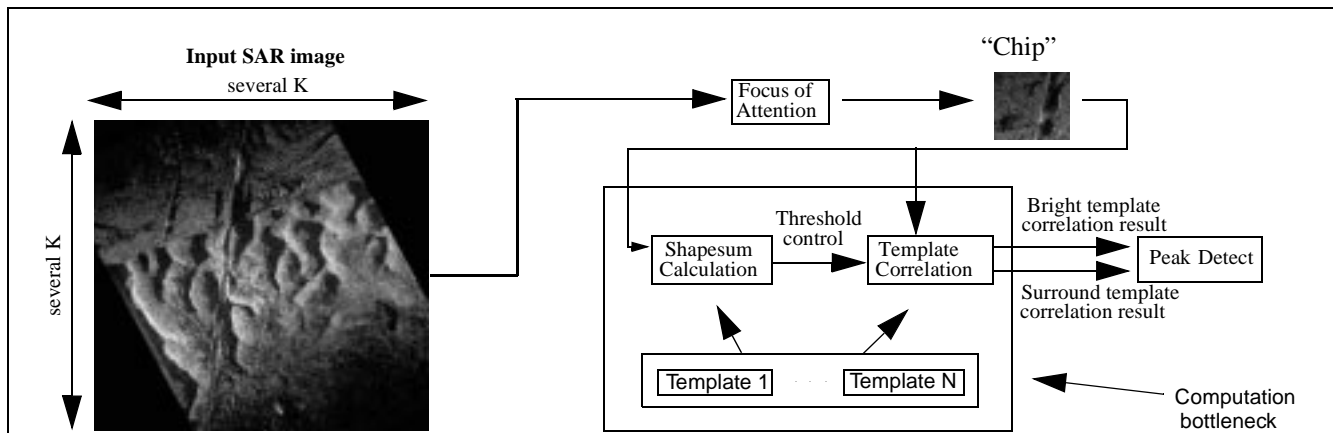
Utilization of FPGAs as dynamic computing devices has been explored by several groups including a team led by Hutchings at Brigham Young University. Hutchings has performed a series of thorough studies in which the benefits of partial reconfiguration was explored using the application example of neural nets [3]. The Brigham Young group has also investigated the use of partial reconfiguration as a means to construct a computer with a dynamic instruction set [4]. This idea, which has also been discussed by Athanas and Silverman in [5], achieves increased efficiency by using FPGA resources to hold the

instructions that are needed on an application-specific basis. These experiences have led to the formulation of design methodologies for partially reconfiguring systems [6], and to important quantitative results on the benefits of partial reconfiguration. For example, for the neural net application partial reconfiguration enabled a 25% reduction in configuration time and a 50% increase in functional density compared with a system based on complete reconfiguration.

In a previous publication [7] we described the implementation of a video communications system implemented using configurable computing techniques. This system delivers real time video at a rate of 8 frames/second, and includes the steps of image transformation, quantization and run-length coding, and BPSK modulation/demodulation. These functions are implemented using a single 5000 gate CLay, with rapid swapping of designs used to time share the gate array hardware. The rapid swapping of designs used in the video system has some commonalities, as well as some significant differences with the approach for ATR that we describe in the present paper.

### 1.2 Application Description: ATR

Automatic target recognition is among the most demanding real time computational problems in existence. The challenge addressed by an ATR system is conceptually simple -- to analyze a digitally represented input image or video sequence in order to automatically locate and identify all objects within the scene of interest to the observer. Since there are many types of imaging devices and many algorithmic choices available to a designer, there are clearly a large number of possible ways to imple-



**Figure 1** High level block diagram for ATR processing. The focus of attention algorithm identifies regions of interest, referred to as “chips”, in SAR images. Chips are correlated against a series of binary target template pairs, with each pair containing a bright template (identifying pixels of strong expected radar return) and a surround template (strong radar absorption). Templates with highest correlation are selected in the peak detection step.

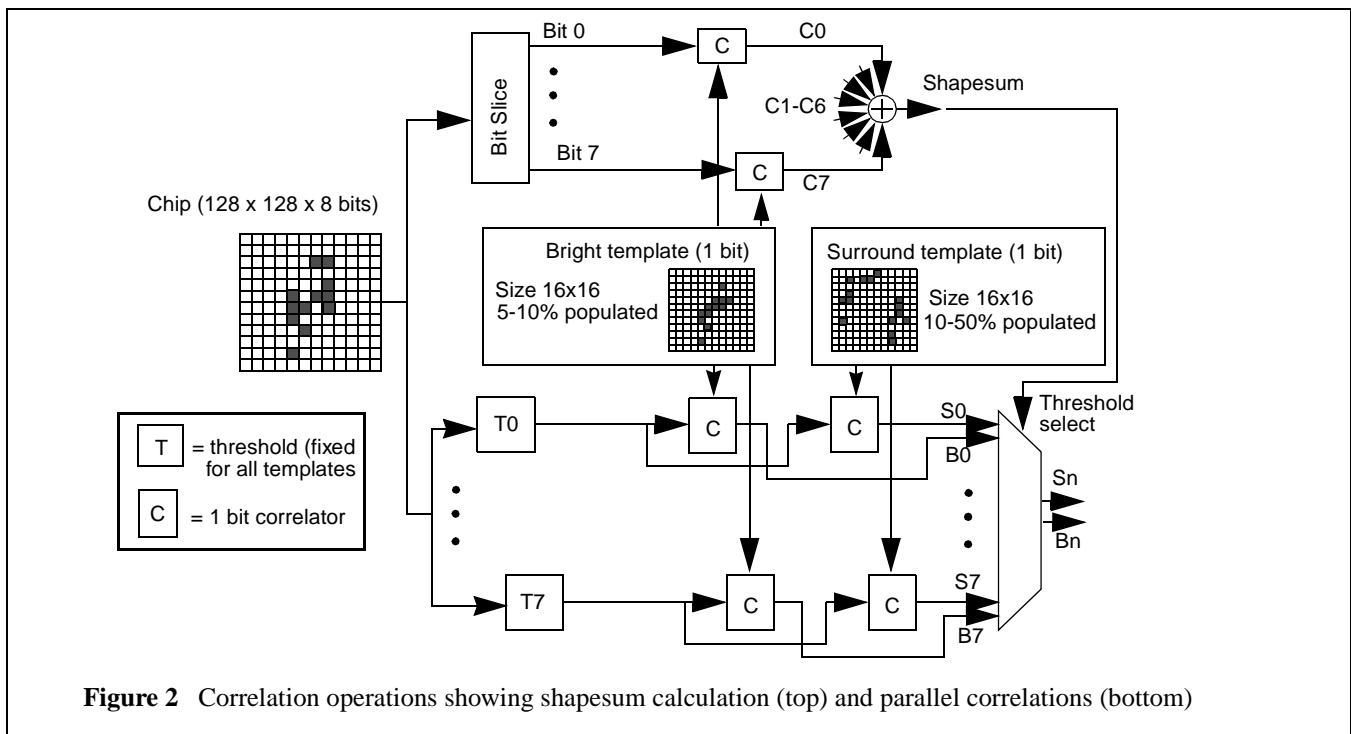
ment an ATR system. In this paper we focus on a particular approach which is currently being applied in the U.S. Department of Defense Joint STARS airborne radar imaging platform, and which therefore has high current relevance and interest.

The processing used in ATR is illustrated in simplified format in Figure 1. Synthetic aperture radar (SAR) images consisting of 8-bit pixels and measuring several thousand pixels on a side and are generated in real time by the radar imager. Images are input to a focus-of-attention processor which identifies a set of regions of interest, each of which contains a potential target. These regions of interest, known by the potentially confusing term “chip”, must then be correlated with a very large number of target templates. Target templates are binary; e.g. each pixel is represented using one bit. The correlation results are output to a peak detector which identifies the template and relative offset at which the peak correlation value occurs. The correlation of chips with templates is the computational bottleneck in the system, involving data rates and computational requirements that exceed by several orders of magnitude the processing load in any other steps in the algorithm. While the precise system parameters vary with implementation, in the work described here we use chip sizes of 128 by 128 and template sizes of 16 by 16. A correlation of a single chip with a single template in this case involves consideration of approximately  $128^2$  relative offsets, corresponding to  $10^5$  bits of output data if the correlation outputs are represented using 6 bits. If there are  $10^3$

templates to be evaluated per chip, the magnitude of the processing task becomes readily apparent when one considers that the imaging system produces many frames per second, each of which contains many chips.

Figure 2 illustrates the correlation operation targeted for FPGA implementation in more detail. Target templates occur in pairs, one member of which is called the bright template and contains pixels from which a strong radar return is expected, and the other member of which is the surround template and identifies pixels where strong radar absorption is expected. In both cases the template is of size 16 by 16, with pixels represented using only one bit. The templates tend to be sparsely populated, with only a relatively small percentage of the pixels set to 1. As will be discussed later, this property is important in obtaining high performance in FPGA implementations. The first step of the correlation is known as a shapsum calculation, in which the 8-bit SAR chip is correlated with the bright template, providing for every pixel in the chip a number which is used for local gain control. The second step is the actual correlation, which is performed in parallel on eight different binary images, each of which is created by applying a different threshold to the chip. The binary images are correlated with both the bright template and the surround template, producing eight pairs of correlation outputs. The shapsum value is used to select which output pair will be processed in the peak detection step.

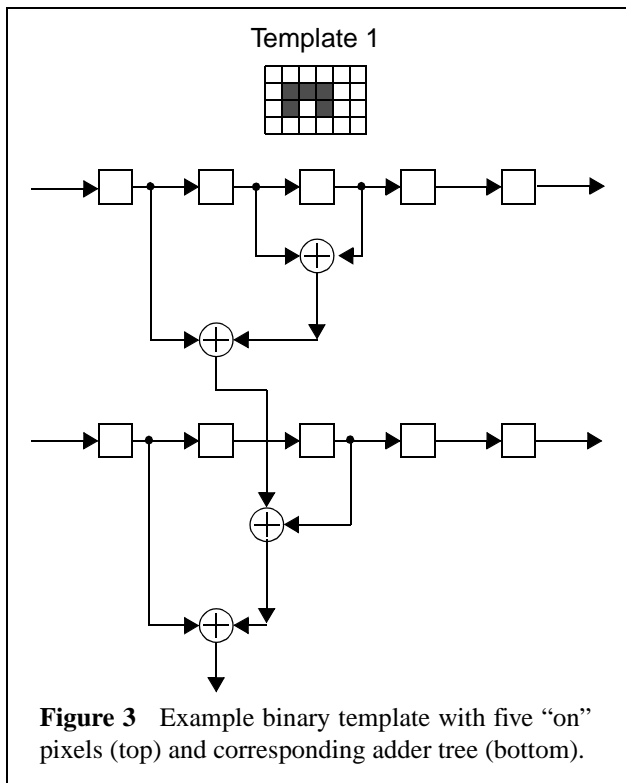
## 2. Mapping and dynamic reconfiguration of



**Figure 2** Correlation operations showing shapsum calculation (top) and parallel correlations (bottom)

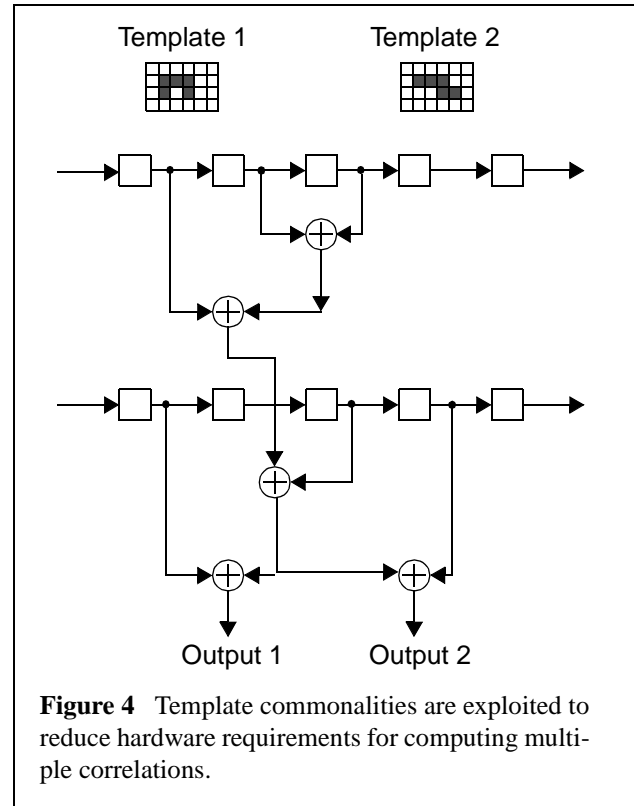
## target templates

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 which would make relatively poor utilization of FPGA resources. More importantly, 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 correlator. This can be illustrated using the example of the simple template shown in Figure 3.



In this example template, only 5 of the 24 pixels are “on”. At any given relative offset between the template and chip, the correlation output is the sum of the five binary pixels in the chip that lie immediately above the “on” pixels in the template. The template can therefore be implemented in the FPGA as a simple adder tree as shown in Figure 3. 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 hard-wired 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, one can seek to identify templates having some topological commonality, and which can therefore share parts of adder trees. This is illustrated in Figure 4, which



shows two templates which share several pixels in common, and which can be mapped using a set of adder trees which leverage this overlap.

The advantage of using FPGAs over ASICs is that FPGAs can be dynamically optimized at the gate level to exploit template characteristics. An ASIC would have to provide large general purpose adder trees to handle the worst case condition of summing all possible template bits. The FPGA, however, exploits the sparse nature of the templates, and only constructs the small adder trees required. We have also shown that FPGAs can exploit other factors such as collapsing adder trees with common elements, and packing unused data points into space-saving RAM-based shift registers. The end result is that a single FPGA can efficiently compute several templates in parallel more efficiently than several general purpose correlating ASICs.

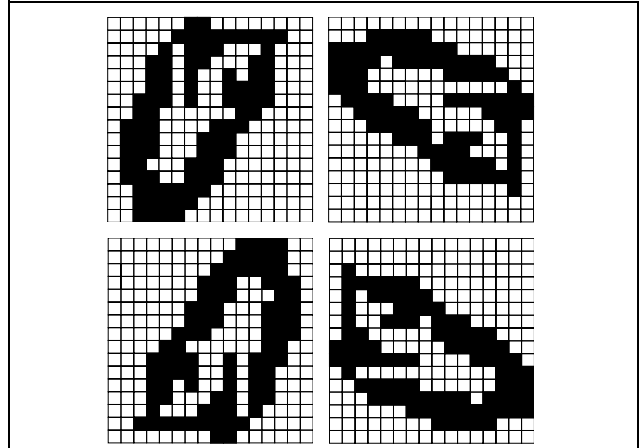
There are many factors that determine the perfor-

mance gain that results from using an FPGA. One of the most important is FPGA reconfiguration time. Assuming that performing the correlation of a single chip requires approximately  $128^2 = 16K$  clock cycles, the reconfiguration must be performed in  $\sim 10^3$  or fewer clock cycles to avoid prohibitive overhead. In this respect a context-switched FPGA with two contexts would be extremely useful. If the idle context could be loaded while the active context was processing, then reconfiguration overhead would vanish. The achievable parallelism is also a critical parameter. Based on the work to date, we estimate that we can map an average of 20 bright templates or 5 surround templates on a single 13000-gate FPGA.

## 2.1 Experimental results for FPGA resource utilization

The approach of using a template-specific adder tree achieves significant reduction in routing complexity over a general correlator which must include logic to support arbitrary templates. To a first approximation, the extent of this reduction is inversely proportional to the fraction of “on” pixels in the template. While this complexity reduction is important, it alone is not sufficient to lead to efficient implementations on FPGAs. This is due primarily to the limited number of flip flops available on commercial FPGAs (for example, the Xilinx XC4010 and ATT ORCA 2C10 contain 800 and 1024 flip flops respectively). This would not generally be sufficient to support buffering the 112 pixels per chip row that are not actually under the template, but need to be wrapped around to the next row of the template. The total number of 1-bit storage elements needed to hold buffered pixel values for all 16 rows is  $16 * 112 = 1792$ . Implementing these on the FPGA using the usual flip-flops based shift registers is inefficient, and for many FPGAs impossible.

This problem can be resolved by collapsing the long strings of image pixels that are not being actively correlated against a template into shift registers, which can be implemented very efficiently on some look-up-table based FPGAs. For example, RAMs in the Xilinx XC4000 library can be used as shift registers which delay data by some predetermined number of clock cycles. Each 16x1 bit RAM primitive uses up a function generator on the FPGA, and can implement an element which 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 the RAM-based shift register elements. This is a small over head price to pay for the savings in CLB usage relative to a brute force implementation using flip flops.



**Figure 5** Four templates that were mapped onto the Xilinx 4010 to explore resource utilization. These examples were generated by 4 rotations of a synthetic template. The number of “on” pixels is 91, which is higher than what would be expected for most templates.

By contrast, the 256 image pixels that lie within the 16 by 16 template boundary at any given time can be stored easily using flip-flop based registers, since there are sufficient flip-flops available to do this, and the adder tree structures do not consume flip-flops. Also, using standard flip-flop based shift registers for image pixels within the template simplifies the mapping process by allowing access to every pixel in the template. 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.

To gain a fuller understanding of the FPGA resource trade-offs involved in template mapping, we implemented in parallel the four templates shown in Figure 5 onto the Xilinx 4010 using the techniques described above. Each template had 91 “on” pixels, few of which are shared with other templates. Since parallelism was low, this exercise gave a worst-case estimate for the capacity of the 4010. The resulting FPGA resource utilization, as summarized in Table 1, shows that 318 flip-flops and 756 function generators were used. The resources used by the two components of target correlation, namely storage of active pixels on the FPGA (1st row of table) and implementation of the adder tree corresponding to the templates (2nd row) 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 will hence increase the number of function generators being used. The total number of templates

	Flip Flops	Fcn. Gen.	I/O pins
Pixel storage	318	116	4
Adder trees	0	640	28
Total used	318	756	32
Total available	800	800	160

**Table 1** Xilinx 4010 resource utilization for the four sample templates shown in Figure 5. Resources for the two basic functions implemented (pixel storage and adder trees) are shown separately. The adder trees account for the majority of the function generator utilization, and are a key factor in limiting the number of templates that can be implemented simultaneously in a single configuration.

implementable on an FPGA will be bounded by the number of usable function generators.

The four templates in the example above exhibit a very low number of common pixels and would probably not be grouped together by the partitioning algorithm. These results suggest that in practice, we can expect to fit 6-10 surround templates having a higher number of overlapping pixels onto a 13000 gate FPGA. Since the bright templates are less populated than the surround templates, we estimate that 15-20 bright templates can be mapped onto a 13000 gate FPGA.

### 3. ATR template partitioning issues

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 placed in every configuration of the FPGA. To accomplish this optimization goal, we want to partition the set of templates into groups that can share computations (i.e., share adder trees) so that fewer resources are used per template. Since the set of templates may number in the thousands, and the goal may be to place ten to twenty templates per configuration, exhaustive enumeration of all of the possible groupings is not an option. Instead, it is best to seek a heuristic method which will furnish a good, although perhaps suboptimal, solution.

Correlation between two templates can establish the number of pixels in common, and is a good starting point for comparing and selecting templates. However, some extra analysis beyond performing iterative correlations on

the template set is necessary. For example, a template with a large surface area may correlate well with several smaller templates, perhaps even completely subsuming them, but the smaller templates may not correlate with each other at all, and would 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 templates in the group. The second option is to compute the number of additions required each time a new template is brought in - effectively recomputing the adder tree every time.

Recomputing the entire adder tree is computationally expensive, and is 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 into a newly formed partition is to determine the number of new terms that adding the template 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. By keeping the number of new terms created to a minimum, 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. C++ allows us to abstract templates and sets of templates into easily manipulated objects. New methods of comparison can be implemented by adding new methods to the objects, making this system adaptable to different heuristics. Furthermore, the system is not tied down to any particular platform, as C++ compilers are available for many different platforms.

The tool uses an iterative approach to partitioning templates. Templates which 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 design tool computes the adder tree for each partition. Figures 6a-6c show the process of creating an adder tree from the templates in a partition. Within each partition, the templates (shown in Figure 6a) are searched for shared subsets of pixels (Figure 6b). These subsets, called terms, can be automatically added together, leading to a template description using terms instead of pixels (Figure 6c). The most commonly occurring addition of two terms is chosen to be added together, forming a new term which 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, and the final result of the process are terms which compute entire templates. For the sample templates shown here, 54 additions would be required to compute the correlations for all five templates in a naive approach. However, after combining the

Figure 6: Example of template grouping to leverage pixels common to multiple templates. In this example, the grouping reduces the number of adders needed from 54 to 29.

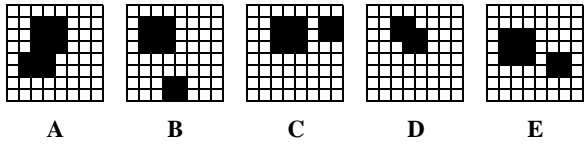


Figure 6a: Five sample templates to be allocated on the same FPGA configuration.

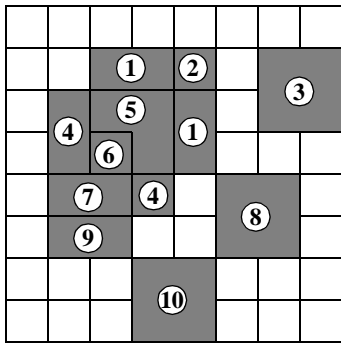


Figure 6b. The terms that result from overlapping templates. Each term corresponds to a group of pixels needed by a particular set of templates. These terms correspond to nodes on an adder tree.

<b>Template A</b>	① + ② + ⑤ + ⑥ + ⑦ + ⑨
<b>Template B</b>	④ + ⑤ + ⑥ + ⑦ + ⑩
<b>Template C</b>	① + ② + ③ + ⑤ + ⑥
<b>Template D</b>	① + ⑤
<b>Template E</b>	④ + ⑤ + ⑥ + ⑦ + ⑧

Figure 6c. The templates rewritten as sums of terms. The most common additions are coalesced into new terms. Here, 5 + 6 occurs four times, and is the first candidate to be replaced with a new term, 11.

templates through the process described here, only 29 additions are required.

#### 4. Configurable computing system design

The increased performance of configurable systems comes with several costs. These include the time and bandwidth required for reconfiguration, memory and I/O required to store intermediate results, and additional hard-

ware required for efficient implementation and debugging. Minimizing these costs requires innovative approaches to system design.

Figure 7 compares the configurable computing architecture discussed here (7c) with a traditional processor (7a) and a customizable computing architecture (7b). The traditional processor receives simple operands from a data memory, performs a simple operation in the program, and returns the result to data memory. Custom computers attempt to gain performance advantages over traditional processors by implementing common operations in hardware. Some fine-grain systems have been proposed [4] but the majority of successful custom computers implement complex operations in a large array of FPGAs [8,9] as depicted in Figure 7b. Large custom computers deliver tremendous performance for some applications, but have definite shortcomings. The number of FPGAs is typically fixed, or only slightly variable through the use of parallel cards. Large applications may not fit, and small applications may inefficiently use available resources. Also, large

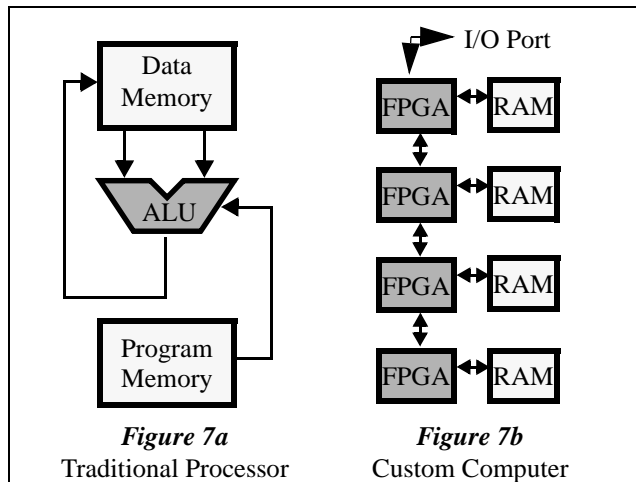
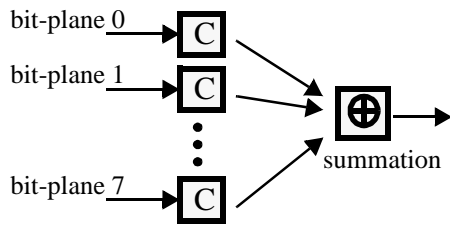


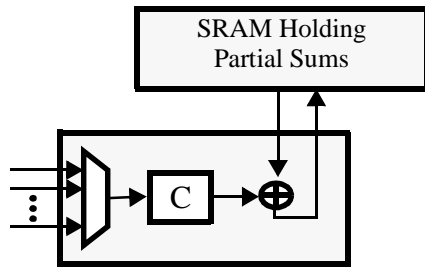
Figure 7a  
Traditional Processor

Figure 7b  
Custom Computer

Figure 7 Architecture comparisons. Traditional processors (7a) take simple operands from memory, perform an operation, and return the result to memory. Custom computers (7b) gain performance advantages by putting common operations into hardware. In dynamic machines (7c) logic is programmed by selecting configuration bitstreams stored in nearby RAM.



**Figure 8a** Eight FPGAs, each performing one bit correlations



**Figure 8b** One FPGA, performing the correlations and adding the results serially

custom computers can be difficult to partition and program.

Our approach to configurable computing is to use a small number of rapidly reconfiguring FPGAs tightly coupled to an intermediate result memory and a configuration memory. This configurable computing architecture attempts to gain some of the advantages of a traditional programmable processor and some of the application-specific performance of a custom computer. We have had success implementing a video compression and transmission system to this architecture [7], and will now briefly describe some of the trade-offs involved in implementing an automatic target recognition (ATR) algorithm on this architecture.

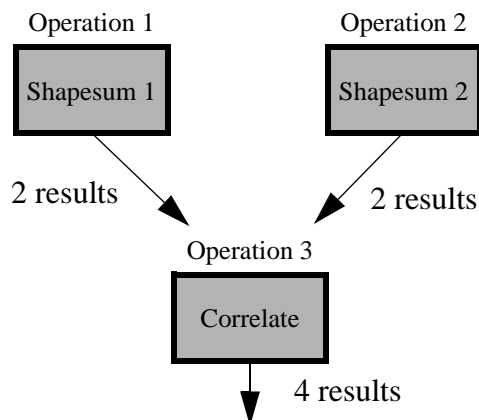
Partitioning of computations plays a major role in any parallel processing environment. For a configurable computing system, partitioning is particularly interesting because it is both a hardware and a software issue. Consider two methods for performing addition of shapsum bit-planes shown in Figure 8a. Method A is a straightforward parallel implementation requiring several FPGAs, and has several drawbacks. First, the outputs from several FPGAs converge at the addition operation. This may create a severe I/O bottleneck. Second, the system is not scalable. If the system requires more precision, and therefore more bit-planes, more FPGAs must be added to the system.

Method B in Figure 8 illustrates our approach. Each bit plane is correlated individually, and then added to the

previous results in the temporary storage. Method B is completely scalable to any image or template precision, and this simple architecture can implement all the correlation, normalization, and peak detection routines required for ATR. One drawback of method B is the cost and power required for the wide temporary result 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 by simply using more of the smaller method B boards.

The approach of a reconfigurable FPGA connected to an intermediate memory allows us to have a fairly complicated flow of control. For example, the shapsum calculation in ATR tends to be more difficult than the image-template correlation. A single FPGA might compute two shapsums or four correlations. For this reason, one may wish to have a program that performs two shapsum operations and forwards the results to a single correlation operation as shown in Figure 9. Looping and branching operations can also find uses for adjustable precision and adaptive filtering.

Reconfigurations for 10k gate FPGAs are typically around 20kB in length. Reconfiguring every 20ms gives a reconfiguration bandwidth of approximately 1MB per FPGA per second. This reconfiguration bandwidth, coupled with the complexity of the flow of control can be handled by placement of a small microcontroller and configuration RAM next to every FPGA. The microcontroller permits complicated flow of control, and since the microcontroller addresses the configuration RAM, it frees up valuable I/O on the FPGA. The microcontroller also is important for debugging, which is a major issue in config-



**Figure 9** Reconfiguration Flow of Control Example

urable systems because the many different hardware configurations can make, it difficult to isolate problems.

Figure 10 shows the current (as of April 1996) version of the evolving configurable computing system for performing ATR. The principal components are labeled in the graph and include a “dynamic” FPGA which is reconfigured on the fly, and which performs most of the computing functions, and a “static” FPGA which is configured only once, and which 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 FIFO is needed to hold the pixels that are being wrapped around to the next row of the template mask. The template sizes used in this implementation are of size 8 by 8; the image against which the templates are correlated are 128 by 128. The maximum clock rate in the current design is 12.5 MHz. Each configuration of the dynamic FPGA implements a total of four template pairs (four bright templates and four surround templates).

The shapsum (see Figure 2) computation is performed in parallel using the approach illustrated in Figure 4. This requires a total of  $D$  clock cycles, where  $D$  is the depth of representation of each pixel. Once the shapsum results are obtained, the correlation outputs are produced at the rate of 1 per clock cycle. Parallelism can not be as directly exploited in this step because different pixels are

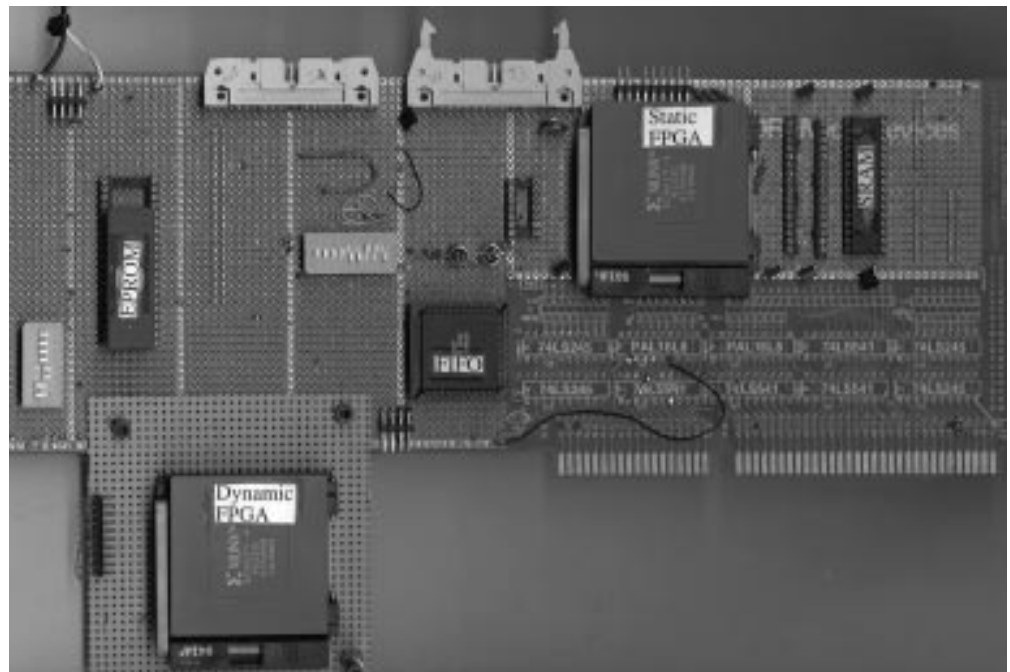
“on” for different templates. However, in the limit of very large FPGAs the number of clock cycles to compute the correlation is upper bounded by the number of possible thresholds as opposed to the number of templates.

## 5. Conclusions

The combination of efficient implementation of bit-level processing tasks and rapid reconfiguration makes FPGAs ideally suited to problems such as ATR. By associating each target template with a relatively simple adder tree and merging topologically similar templates, a processing parallelism of approximately 10 can be achieved using currently available FPGAs. Provided that the FPGA reconfiguration time is on the order of milliseconds, dynamic reconfiguration can be used to support sustained computation for ATR template matching at much higher throughput and lower cost than a general purpose correlator.

A demonstration system for implementing the ATR algorithm illustrated in Figure 2 has been constructed and is shown in Figure 10. This system includes a single dynamically-reconfigured FPGA (Xilinx 4013) for computation and a second FPGA for control and some post-processing functions. Each configuration of the dynamic FPGA performs the ATR matching of a 128 by 128 image against four template pairs, with each template of size 8 by 8. These results provide a proof-of-concept for the viability of performing ATR in a configurable computing environment.

**Figure 10** Configurable computing system for ATR. Most computational functions are performed on the dynamic FPGA. The static FPGA controls the processing and also performs some of the postprocessing. Other components on the board include an EPROM for configuration bitstreams, a FIFO for storage of “wraparound” pixels during the 2D correlation operation, and an SRAM for image storage. The board currently runs at a clock speed of 12.5 MHz, and correlates one 8-bit deep 128 by 128 image against sixteen 8 x 8 template pairs in approximately 210 msec (about 130 msec is used for configuration).



## 6. Acknowledgment

This work was supported by ARPA/ITO under contract number DABT63-95-C-0102.

## 7. References

- [1] E. Tau, I. Eslick, D. Chen, J. Brown and A. DeHon, "A first generation DPGA implementation," *Proceeding of the Third Canadian Workshop on Field-Programmable Devices*, pp. 138-143, May 1995.
- [2] A. DeHon, "DPGA Utilization and Application," to appear in *Proceedings of the 1996 International Symposium on Field Programmable Gate Arrays*, February 1996.
- [3] M.J. Wirthlin and B.L. Hutchings, "Density enhancement of a neural network using FPGAs and run-time reconfiguration," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180-188, April 1994.
- [4] M.J. Wirthlin and B.L. Hutchings, "A dynamic instruction set computer," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 99-107, April 1995.
- [5] P.M. Athanas and H.F. Silverman, "Processor reconfiguration through instruction set metamorphosis," *Computer*, vol. 26, pp. 11-18, March 1993.
- [6] J.D. Hadley and B.L. Hutchings, "Designing a partially reconfigured system," *Proceedings of SPIE Photonics East: FPGAs for Fast Board Development and Reconfigurable Computing*, October 1995.
- [7] J.D. Villasenor, B. Schoner, and C. Jones, "Video communications using rapidly reconfigurable hardware," *IEEE Transactions on Circuits and Systems for Video Technology*, January 1995.
- [8] R. Amerson, R.J. Carter, W.B. Culbertson, P. Kuekes, and G. Snider, "Teramac - Configurable custom computing," *Proceedings of the 1995 Symposium on FPGAs for Custom Computing Machines*, pp. 180-188, April 1995.
- [9] L. Moll, J. Vuillemin, and P. Boucard, "High-energy physics on DECPeRLE-1 programmable active memory," *ACM Third International Symposium on Field-Programmable Gate Arrays*, pp. 47-52, Monterey, CA 1995.