Automatic Target Recognition with Dynamic Reconfiguration

Jack Jean, Xuejun Liang, Brian Drozd, Karen Tomko and Yan Wang
Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435, USA

July 23, 1999

Abstract. This paper describes the acceleration of an infrared automatic target recognition (IR ATR) application with a co-processor board that contains multiple field programmable gate array (FPGA) chips. Template and pixel level parallelism is exploited in an FPGA design for the bottleneck portion of the application. The implementation of this design achieved a speedup of 21 compared to running on the host processor. The paper then describes an FPGA resource manager (RM) developed to support concurrent applications sharing the FPGA board. With the RM, the system is dynamically reconfigurable. That is, while part of the co-processor board is busy computing, another part can be reconfigured for other purposes. The IR ATR application was ported to work with the RM and has been shown to adapt to the amount of reconfigurable hardware that is available at the time the application is executed.

Keywords: Automatic Target Recognition (ATR), Configurable Computing, Field Programmable Gate Array (FPGA), Reconfiguration

1. Introduction

Computing systems that use co-processor boards based on field programmable gate array (FPGA) chips may adapt their hardware resources to the application requirements. The technology has been demonstrated for the acceleration of various applications, such as automatic target recognition (ATR)[1], neural networks[2, 3, 4], Adobe Photoshop[5], Solar Polarimetry[6], and machine vision[7]. This paper describes the acceleration of an infrared (IR) ATR application with a multiple-FPGA board and the development of an FPGA resource manager software system. The software system supports dynamic reconfiguration so that the IR ATR application can concurrently execute with other applications. In addition, in the concurrent execution environment, the application can adapt to the amount of FPGA resources available at the application startup time.

The purpose of the IR ATR application is to locate and identify ground vehicles from infrared images. The algorithm does not use inter-frame information and therefore the processing is based on individual image frames. It is one of the research challenge problems identified by the Adaptive Computing Systems (ACS) program of the U.S. Defense Advanced Research Projects Agency. Similar to the ATR processing of synthetic aperture radar (SAR) images as in [1], the IR ATR application consists of many conceptually simple steps that each evaluates the *matching* between an image area and a template pair of target and background. However, because of the difference in the way *matching* is defined in the two different ATR algorithms, the adder-tree approach used in [1] cannot be adopted for the IR ATR algorithm. Instead, a different FPGA design that explored the template level parallelism and the pixel level parallelism was developed. The achieved performance is reported and analyzed in this paper.

In a deployable system, the IR ATR algorithm is probably only one among many other algorithms. For example, there may be one algorithm to perform target tracking based on motion detection, one to perform ATR with a different sensor, and one to perform sensor fusion. Most of these algorithms can take advantage of the FPGA resources and, depending on the application environment, the distribution of the FPGA resources to different algorithms should be adjusted accordingly. That means, there is a need to support concurrent programs sharing the FPGA board and to dynamically allocate FPGA resources to those programs. The second part of this paper describes an FPGA resource manager (RM) developed for this purpose.

The RM provides an operating system like interface for the programmable hardware to hide the architectural details of the coprocessor board, to manage reconfiguration of the hardware during application execution, and to fairly allocate FPGA resources among multiple programs. With the RM, the system is dynamically reconfigurable. That is, while part of the reconfigurable hardware is busy computing, another part can be reconfigured for other purposes. The IR ATR application was ported to work with the RM. The resulting ATR design was able to adapt to the amount of reconfigurable hardware that was available at the time the application was executed.

Compared to *static* reconfiguration schemes, which do not reconfigure the hardware during the execution of an application, a dynamic reconfiguration scheme such as the one with the RM can accommodate more applications, typically those that require more FPGA resources than what is available and their usage of FPGA resources can be satisfied once spread out over time. Compared to other dynamic reconfiguration schemes that statically determine how to reuse the FPGA resources [3, 8], the RM of our system allocates FPGA resources at run time and relieves application developers from the management of FPGA resources. The RAGE project [9] is similar to our own, but emphasizes partial reconfiguration. It does not support pre-loading of configurations.

The rest of the paper is organized as follows. Section 2 describes the hardware platform and the design used to accelerate the IR ATR application. The achieved performance is reported and analyzed. Section 3 shows the design and the implementation of the RM. The IR ATR application is modified to work with the RM and the results are summarized. Section 4 concludes the paper.

2. Accelerating the IR ATR Application

2.1. Hardware Platform

The reconfigurable computing platform used in this project is a 180 MHz Pentium-pro personal computer hosting a G900 FPGA board which is a PCI bus based board manufactured by Giga Operations Corporation. The board has a *modular design*, making it suitable for resource sharing among applications. This design consists of eight computing modules (XMODs) where each XMOD contains two XC4020E FPGA chips, 2 MB DRAM, and 256 KB SRAM (see Figure 1). Each XC4020E chip on an XMOD is connected to a 128 KB SRAM through a 16-bit wide data port. Note that a maximum of sixteen XMODs can be configured in one G900 board.

The XMODs are connected together by 128 wires, called the XBUS. Among those 128 wires, 21 of them are used to support a custom bus protocol, called HBUS, which defines the pins and timing of signals used for the host (or more specifically, the PPGA) to FPGA interface. The XBUS also contains six 16-bit busses that provide inter-XMOD connectivity. The host processor cannot access SRAM directly. All accesses to SRAM need to go through the XBUS and therefore through the FPGA chips.

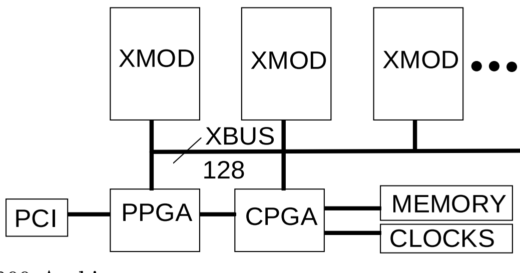


Figure 1. G900 Architecture

There are two special purpose onboard FPGAs that are not part of any XMOD. They are the PPGA and the CPGA. The PPGA (Xilinx

XC4013E-2) controls communication between the host computer and the XMODS (Figure 1), by acting as the PCI bus interface to the board. The CPGA (Xilinx XC5210-5) implements clock generation, runtime configuration and power up functions. While the FPGAs can run at clock rates up to 66MHz, the G900 board and host interface is currently limited to 16MHz.

2.2. The IR ATR Algorithm

The IR ATR algorithm locates and identifies ground vehicles based on a single IR image frame. Similar to the ATR processing of SAR images as in [1], the IR ATR algorithm consists of many conceptually simple steps that each evaluates the matching between an image area and a template pair of target and background. Unlike the ATR algorithm in [1] where huge number of templates are applied for the identification of one image area, the IR ATR algorithm uses a *decision tree* to improve the computational efficiency.

An overview of the algorithm is shown in Figure 2. The algorithm contains several steps, called Round 0, Round 1, Round 2, ..., and Round 5. The first step, Round 0, is applied to the whole image of size 480×640 to identify the location and the *target super-group* of *regions of interests* (ROIs). Here an ROI is an image pixel whose surrounding area meets a certain criterion and is considered a candidate for further investigation. Usually no more than 20% of the image pixels become ROIs. (For the four test images available to the authors, less than 5% of pixels are ROIs.) Six template pairs are used in Round 0.

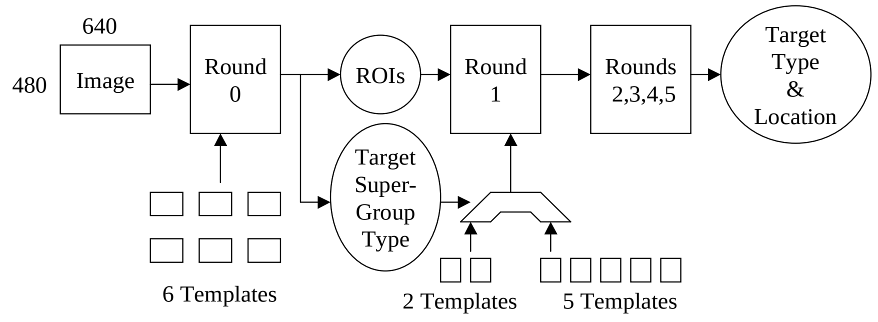


Figure 2. The ATR algorithm

Only ROIs are passed to the second step, Round 1, for further hypothesis testing and classification into *target groups*. Depending on which *target super-group* an ROI belongs to, Round 1 uses either two or five templates. ROIs that passed the Round 1 test are further tested through the remaining rounds and at the end, several pixels survived with their target types identified. Because of the pruning process, Round 0 is computationally the most expensive step, followed by Round 1. And these two rounds were targeted for FPGA acceleration with the G900 board.

A special feature of the IR ATR algorithm is that each image is partitioned into five horizontal strips of different heights. Each strip corresponds to a range of distances for the IR sensor. All the templates need to be re-scaled from one strip to the next in order to calibrate to the effects of different distance ranges. The computation involved in Round 0 and Round 1 and their corresponding FPGA designs are described as follows.

2.3. Round 0 Computation and the FPGA design

Round 0 locates the ROIs by applying six template pairs through the whole image except the image boundary. (The number of template pairs is predetermined based on the number of targets and the target types.) Each template pair contains one template for a target super-group and one for background. Each template contains a pattern of exactly 30 pixels in a relatively large area, say, of size 20 by 50. (Different templates have different sizes.) Each image frame is partitioned into five strips of different heights and each template needs to be resized across strip boundary.

2.3.1. *Round 0 Computation*

For each image pixel that is not along the boundary of an image frame, its surrounding area is tested against a template pair to see if it is an ROI. As a result, it is possible for an image pixel to be identified as an ROI six times, each time as a candidate of a different target super-group. The computation involved in the testing of one image pixel against one template pair is as follows.

1. Use the background template that contains a pattern of 30 points to locate 30 *background pixels* in the surrounding area. Let **MEAN** be the averaged values of these 30 background pixels. Use the target template that contains a pattern of 30 points to locate 30 *target pixels* in the surrounding area.

2. Compute the HOT and COLD values of background and target as follows.

```
FOR I FROM 1 TO 30
    IF(background_pixel[I] > MEAN)
        BKG_Hot = BKG_Hot + (background_pixel[I] − MEAN)
    ELSE
        BKG_Cold = BKG_Cold + (**MEAN** − background_pixel[I])
    END of IF
END of FOR

FOR I FROM 1 TO 30
    IF(target_pixel[I] > **MEAN**)
        TRG_Hot = TRG_Hot + (target_pixel[I] − **MEAN**)
    ELSE
        TRG_Cold = TRG_Cold + (**MEAN** − target_pixel[I])
    END of IF
END of FOR
```

3. Based on the COLD and HOT values of target and background compute the *correlation*.

```
    Hot_cor = (TRG_Hot − BKG_Hot)/(TRG_Hot + BKG_Hot)
    if(Hot_cor < 0) Hot_cor = 0
    Cold_Cor = (TRG_Cold − BKG_Cold)/(TRG_Cold + BKG_Cold)
    if(Cold_Cor < 0) Cold_cor = 0
    correlation = Hot_cor + Cold_cor
```

4. If (*correlation* ≥ 0.65), then the pixel is a ROI.

In [1] a SAR ATR algorithm was mapped to an FPGA board by processing individual image bit slices, storing several lines of image pixels on chip, and using adder trees for processing. The same technique is not feasible for Round 0 of this IR ATR algorithm for two reasons. First, using image bit slices is not practical because of the comparisons and the conditional additions in Step 2 and the need to compute **MEAN** in Step 1 and use it in Step 2. Second, without bit slicing, the buffering of 20 rows of image data on chip requires more than 2,000 Xilinx XC4020 CLBs (Configurable Logic Blocks) while a XC4020 chip contains only 784 CLBs. Note that one row contains 640 pixels in our case while there are only 128 pixels per (chip) row in [1].

The original source code uses division to compute **MEAN**, Hot_cor, and Cold_Cor. It also uses floating point computations for the majority of Round 0 computation. Since both division and floating point computations need large number of CLBs, the original computation is reformulated as follows.

1. Let **SUM** be the sum of those 30 background pixels.

2. Compute the HOT and COLD values of background and target as follows.

```
FOR I FROM 1 TO 30
    IF(30 x background_pixel[I] $>$ **SUM**)
        BKG_Hot30 = BKG_Hot30 + (30 x background_pixel[I] $-$ **SUM**)
    ELSE
        BKG_Cold30 = BKG_Cold30 + (**SUM** $-$ 30 x background_pixel[I])
    END of IF
END of FOR

FOR I FROM 1 TO 30
    IF(30 x target_pixel[I] $>$ **SUM**)
        TRG_Hot30 = TRG_Hot30 + (30 x target_pixel[I] $-$ **SUM**)
    ELSE
        TRG_Cold30 = TRG_Cold30 + (**SUM** $-$ 30 x target_pixel[I])
    END of IF
END of FOR
```

This step is later on referred to as the TEMPERATURE step.

3. Compute the numerators and the denominators of the COLD and HOT correlation values.

```
IF (TRG_Hot30 $<$ BKG_Hot30) Hot_N = 0
ELSE Hot_N = TRG_Hot30 $-$ BKG_Hot30
Hot_D = TRG_Hot30 + BKG_Hot30
IF (TRG_Cold30 $<$ BKG_Cold30) Cold_N = 0
ELSE Cold_N = TRG_Cold30 $-$ BKG_Cold30
Cold_D = TRG_Cold30 + BKG_Cold30
```

This step is later on referred to as the CONVERT step.

4. If (20 x (Hot_N x Cold_D + Cold_N x Hot_D) $-$ 13 x (Hot_D x Cold_D)) ≥ 0), then the pixel is an ROI. (This is based on the fact that the threshold constant 0.65 is equal to $\frac{13}{20}$.) This step is later on referred to as the ASSERT step.

With the new formulation, the divisions are replaced with multiplications and all the floating point computations are replaced with integer operations. In addition, a multiplication with a constant, such as 13, 20, and 30, can be replaced with shifting and addition.

2.3.2. *Round 0 FPGA Design*
An FPGA design that explored the parallelism at the template level and the pixel level was developed. Figure 3 is an overview of the design. Six XC4020 chips, each on a XMOD, are used. Each chip processes the computation for one template pair. The image is processed strip by strip. After an image strip is broadcast to the six FPGA chips, each FPGA chip is loaded with a template pair. All the six chips then start the computation in parallel and the locations of identified ROIs are stored in SRAMs. When a chip finishes its computation, it interrupts the host processor which then reads back the ROI locations. A program running on the host processor would do everything as required in the original source code except the locating of ROIs. To overlap the FPGA computation and the host machine computation, the host program uses two threads. When the child thread is waiting for the FPGA signal of computation completion for one strip, the parent thread can go ahead process ROIs identified for the previous strips.

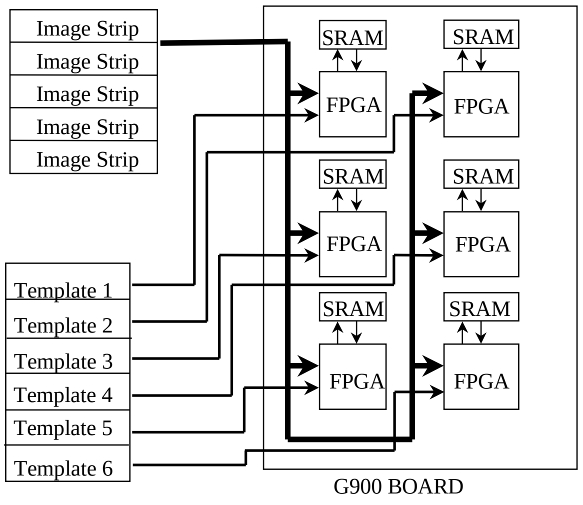


Figure 3. Round 0 FPGA Design: each XC4020 chip handles one template

The design inside each FPGA chip is shown in Figure 4. The SRAM in the figure is on an XMOD next to the FPGA chip. It is a 128 KB memory arranged as 64 K by 2 Bytes. The left hand side of the figure shows the data that need to be sent from the host processor to the FPGA chip. These include the image strip, the template pair, and three parameters, base address, delta_x, and delta_y, that specify the image area to be processed. The 60 valid points on a template pair are stored in a FIFO on chip. The FIFO content is used together with the base address to produce the 16-bit SRAM addresses. To test if a pixel is an ROI or not, 60 addresses are produced to get 60 test points from

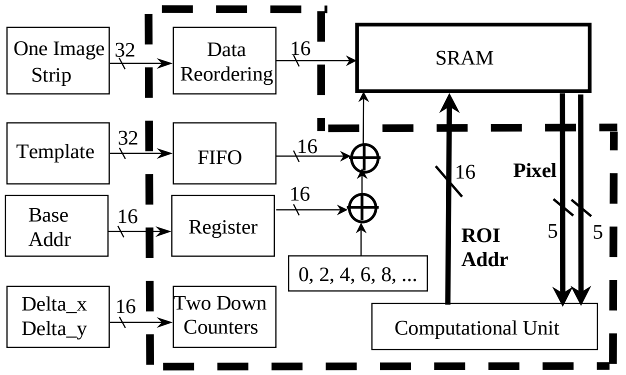


Figure 4. Round 0 FPGA Design: XC4020 internal; the FPGA chip is enclosed within the dashed lines.

the SRAM, 30 for the target and 30 for the background. If a pixel is identified as an ROI, its location in the format of a 16-bit address is written to the SRAM.

This design has two special features. First, the FPGA design uses five bits per image pixel even though the original code uses one byte per pixel. From simulation results, the recognition results of using five bits were considered comparable judging from the image output files and the number of ROIs. While using five bits per pixel does not help in any way on a general purpose processor, it helps reduce the area requirement in the FPGA design. Second, each template is evaluated on two neighboring pixels in parallel. Note that, when 128 K pixels are stored in the order of scanned lines in the 64K by 2 SRAM, two neighboring pixels cannot always be accessed in one clock cycle. For example, if pixels n and $n+1$ are stored at the same address, then pixels $n+1$ and $n+2$ are at two consecutive addresses and they cannot be accessed in one clock cycle. To facilitate the parallelism, the 128 KB SRAM stores only 64 K image pixels at 64 K different addresses and at each address the other byte stores the neighboring (and redundant) pixel. That is, if pixels n and $n+1$ are stored at one address, then pixels $n+1$ and $n+2$ are stored at the next address.

The FPGA design contains the following components in each chip.

1. SRAM controller: it has four functions. (a) It receives contiguous image data from the host processor and stores them in the SRAM on the XMOD. (b) For the test of each pixel, it allows the (almost random) access of 60 test points. (c) Store a pixel location into SRAM if the pixel is an ROI. (d) Allow the host program to read back all the ROIs. This component takes 186 CLBs.

2. Buffer to store 60 test point locations, each stored as a 16-bit (SRAM) address offset. This component takes 32 CLBs.
3. A computation unit as shown in Figure 5(a) that contains three parts:

   a) Two copies of the TEMPERATURE units. Each TEMPERATURE unit as shown in Figure 5(b) contains (1) an accumulator to compute the sum of those 30 background image values, (2) a buffer to store those 30 values, and (3) comparators and adders to compute HOT and COLD values. This component takes 220 CLBs.

   b) A multiplexer and the CONVERT unit. This component takes 86 CLBs.

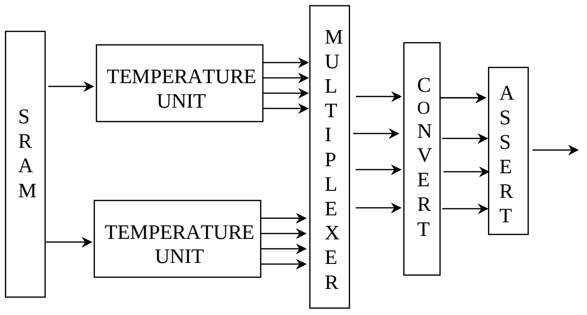
   c) One copy of the ASSERT unit to evaluate if the correlation is greater than or equal to a threshold. Bit-serial multipliers are used in this component to save area. This component takes 118 CLBs.

   The ASSERT unit is time-multiplexed to process the outputs of the two TEMPERATURE units. This arrangement saves one CONVERT unit and one ASSERT unit at the expense of requiring the multiplexer. In addition, a 33 MHz clock is used for the ASSERT unit while the rest of the chip runs with a 16 MHz clock.

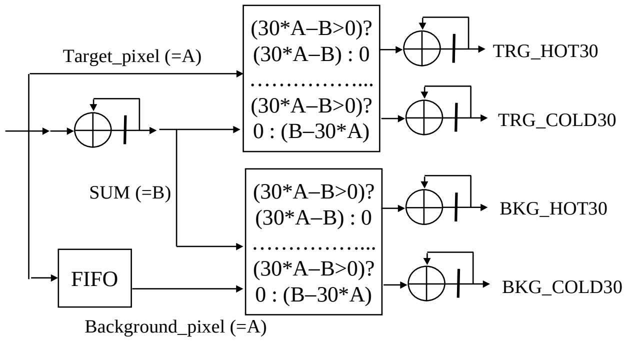
The whole Round 0 design takes 704 CLBs per chip which is 89.8% of the total CLBs on a XC4020 chip.

2.3.3. *Round 0 FPGA Performance and Analysis*
The execution times with and without FPGA for Round 0 are summarized in Table I. Note that the time to initialize the board (2.31 seconds) and the time to load an FPGA configuration file (0.37 seconds) are not included in the table because they are considered as system initialization and can be amortized over an image video sequence. Table I shows that the computation in average takes 13.07 seconds on the 180 MHz Pentium-Pro host machine and 0.614 seconds with FPGA even though the FPGA chips run at a much slower clock frequency. As a comparison, the same computation in average takes 5.85 seconds on a 400 MHz Pentium II Xeon PC, takes 5.2 seconds on a SGI Onyx that uses a 195 MHz R10000 processor, and takes 13.46 seconds on a SGI O2 that uses a 180 MHz R5000 processor.

To analyze the execution times with FPGA, note that they come from three different parts that take no more than 0.655 (= 0.075 + 0.018 + 0.562) seconds.



(a)



(b)

Figure 5. Round 0 FPGA Design: (a) the computational unit: the CONVERT and ASSERT units are time-multiplexed and fed by two TEMPERATURE units, (b) the internal of the TEMPERATURE unit

Table I. Execution times (in seconds) for Round 0.

Image	On PC	With FPGA	Speedup
1	13.106	0.602	21.2
2	12.980	0.614	21.1
3	13.187	0.622	21.2
4	13.016	0.617	21.1

1. *Image data transfer from PC to XMODs*: Since the image size is 480 x 640, there are 300 KBytes of data. Suppose half of the data need to be sent twice because of the limited size of SRAM on XMODs, it should take no more than 0.075 seconds. (The data transfer bandwidth is about 6MB/sec for write and 4MB/sec for read from G900.)

2. *Other data transfer*: about 0.018 seconds. (1) The sending of templates to XMODs: less than 10 Kbytes (2) The reading of results from XMODs: Assume that less than 10% of pixel locations are ROIs. Since specifying each ROI location takes 2 bytes, there are about 60 (= 300 x 0.1 x 2) KBytes to read.

3. *Real computation*: During the computation, the image data need to be read from SRAM to FPGA chips. For SRAM, it takes one clock cycle to read two bytes at the same address, which are used for the ROI testing of two neighboring pixel locations. For an image frame of 300 K pixels, each pixel requires the evaluation of 60 bytes of test points. So totally it takes 9 M (= 60 x 300 K / 2) clock cycles, or 0.562 seconds with a 16 MHz clock.

2.4. Round 1 Computation and the FPGA design

Even though Round 1 is computationally an order of magnitude less expensive than Round 0, speeding up Round 1 is more difficult because of its relatively higher I/O requirement. The FPGA design described in this section does not speed up Round 1; it actually slows down Round 1. But it provides the CLB count for the computation unit and the insight into how to speed up Round 1.

2.4.1. *Round 1 Computation*

Each ROI identified in Round 0 needs to be tested with Round 1. Depending on its target supergroup, the ROI goes through a test that contains either two or five templates. In either case, the following computation is involved in each template calculation.

1. Given a ROI pixel, locate 40 pairs of test points based on the template. Let P_i and Q_i denote a pair of points, where $i = 1, 2, 3, ..., 40$.

2. Compute the following parameters.

$$\begin{aligned} SumP &= \textstyle\sum_{i=1}^{20} |P_i - Q_i| \\ SumM &= \textstyle\sum_{i=21}^{40} |P_i - Q_i| \\ Sum &= SumP + SumM \\ SSum &= \textstyle\sum_{i=1}^{40} (P_i - Q_i)^2 \end{aligned}$$

3. Calculate the correlation.

$$\begin{array}{l} SM = 40 * SSum - Sum * Sum \\ \text{IF } (SM = 0) \ correlation = 0 \\ \text{ELSE } correlation = (SumP - SumM)/\sqrt{SM} \end{array}$$

4. If ($correlation \geq 0.45$), then the ROI needs to be further processed through Round 2.

To avoid the computing of a square root and a division, the last two steps can be reformulated as follows since $0.45^2 = \frac{81}{400}$.

$SM = 40 * SSum - Sum * Sum$
IF $((SumP > SumM)$ AND $(400 * (SumP - SumM)^2 \geq 81 * SM))$
THEN $the\ ROI\ needs\ to\ be\ further\ processed\ through\ Round\ 2$.

2.4.2. *Round 1 FPGA Design*

The Round 1 FPGA design is shown in Figure 6. The design is a straightforward translation of the (reformulated) computing process into hardware that fits into one XC4020 chip. Each time the host program sends two pairs of test points to the chip, one for the *SumP* parameter and the other for the *SumM* parameter. Once all the 40 pairs for one template have been received by the chip and after a computation latency of five clock cycles, the output of the design indicates if there is a need for the ROI to go through Round 2 computation. Because the computation latency is fixed after the hardware receives the last pair of test points, there is no need to interrupt the host processor when the output value is ready. The host program simply reads the output value with a delay that guarantees that the value is ready. As indicated in the figure, each test point uses only 4 bits of precision even though the original pixel uses 8 bits. The 4-bit precision was chosen after simulation on the four test images. The design uses 379 CLBs or 48.3% of the CLBs in a XC4020. When the on-chip register values can be read from the host processor for debugging purpose, the number of CLBs needs to increase to 441.

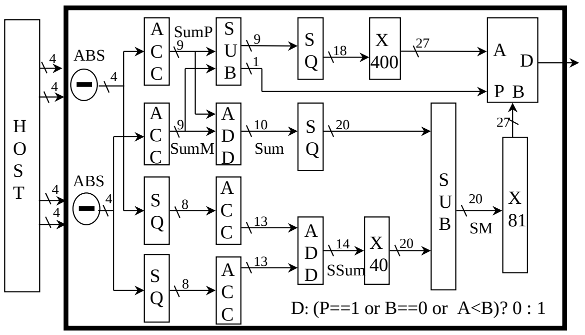


Figure 6. Round 1 FPGA Design

Table II. Execution times (in seconds) for Round 1. The second column specifies the respective numbers of ROIs that need five templates and two templates.

Image	No. of ROIs (5/2 Templates)	Time On PC	Time With FPGA
1	3337/2838	1.251	1.406
2	4058/4392	1.218	1.437
3	2357/924	0.501	0.657
4	5627/8154	1.359	1.671

2.4.3. *Round 1 FPGA Performance and Analysis*
The execution times with and without FPGA for Round 1 are summarized in Table II. Again the time to initialize the board and the time to load an FPGA configuration file are not included in the table. Table II shows that the computation in average takes 1.032 seconds on the 180 MHz Pentium-Pro host machine and 1.293 seconds with FPGA. As a comparison, the same computation in average takes 0.104 seconds on a 400 MHz Pentium II Xeon PC.

To analyze why the FPGA design is slow, the fourth image file is used as an example. In that case, 5627 ROIs use five templates and 8154 ROIs use two templates. So the total number of templates that are used in Round 1 is 44443. And therefore the host program spends

1. 0.39 seconds on reading, writing, and formating data.

2. at least 0.59 seconds on I/O over the PCI bus. For each template 80 test points, or 80 bytes, are sent over the bus per ROI. Therefore, assuming the bus is dedicated to Round 1, it takes 0.59 seconds (= 44443 x 80 bytes /(6 M bytes/sec)). It actually takes more than that amount of time because Round 0 and the host processor also share the bus.

3. about 0.29 seconds on FPGA computation. This is based on the assumption that it takes five clock cycles to receive two pairs of test points over the PCI bus. Receiving 40 pairs therefore takes 100 clock cycles. With the five clock cycles of computation latency, it takes 105 cycles per ROI per template. With a 16 MHz clock, that translated into 0.29 seconds (= 44443 x 105/(16 MHz)). Note that a fairly large amount of the time is overlapped with the time for I/O over the PCI bus.

A clear drawback of the design is its relying on the host processor to read those 80 test points and send them over the PCI bus. A better design would store the image data on the SRAM and let the FPGA chip read those 80 points directly. Another special feature of the IR ATR application that has not been explored is the fact that, when five templates are used for each ROI, there is a significant amount of overlapping among the 400 ($= 5 \times 80$) test points. This will lead to "template-specific" FPGA designs, similar to what was used in [1].

3. Dynamic Reconfiguration Support

In a deployable system, the IR ATR algorithm is probably only one among many other algorithms. For example, there may be one algorithm to perform target tracking based on motion detection, one to perform ATR with a different sensor, and one to perform sensor fusion. Most of these algorithms can take advantage of the FPGA resources and the distribution of the FPGA resources to different algorithms should adapt to the application environment. That means, there is a need to support concurrent programs sharing the FPGA board and to dynamically allocate FPGA resources to those programs. This section describes such a system software, called the FPGA resource manager (RM), and the implementation of the IR ATR program on the system. Since a previous version of the RM was reported in [11], the modifications made to the RM to accommodate the IR ATR are described in this section. The achieved performance is also reported.

3.1. RM Overview

A block diagram illustrating a system with the RM is shown in Figure 7 where multiple applications can concurrently share the FPGA resources. Note that a single application may contain multiple code segments, say, one for computing correlation and one for computing a particular morphologic operation, and those segments may need to share the FPGA resources, sometimes concurrently (only if the application is multi-threaded).

In a system with the RM, each application consists of a program to be executed on the host machine and a flow graph representing the portion of the application to be executed on the FPGA resources. The host program is responsible for starting the execution of graph nodes through the RM. The flow graph is a weighted graph where each node represents FPGA computation and the weighted edges represent the control flow of the host program. With the information of multiple

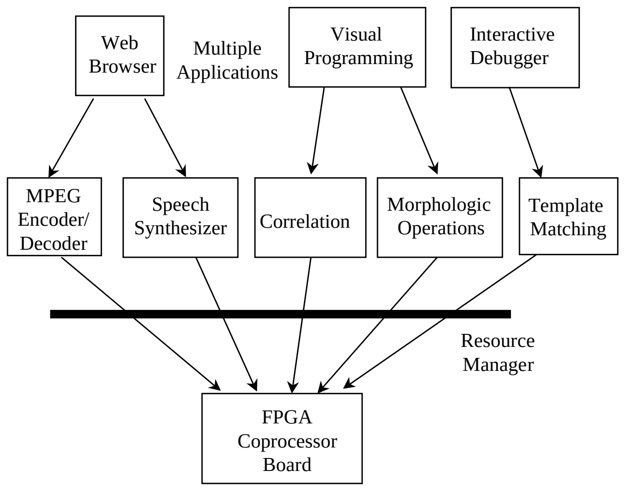


Figure 7. The dynamic reconfiguration system

flow graphs, one for each application, the RM allocates and de-allocates FPGA resources so that new nodes may be loaded into the system while other nodes are being executed. In addition, a speculative strategy is adopted by the RM in the “pre-loading” of FPGA configuration files to reduce and hide the reconfiguration overhead and to improve performance. The FPGA architecture is *modular* in the sense that the FPGA resources consist of a number of hardware units and each graph node uses an integer number of hardware units.

To provide the dynamic reconfiguration capability and to support concurrent applications on the G900 board, an XMOD-based RM and a set of library functions have been designed and implemented. With the XMOD as the basic resource unit, the RM allocates and de-allocates reconfigurable computing resources both *on-demand* and *speculatively.* A set of library functions is provided so that application developers can pass information from an application to the RM without worrying about the details of the inter-process communications or the details of the G900 board control.

The RM is implemented as a multi-threaded application as shown in Figure 8. The *main thread* is the first thread to be created and is the parent thread for the other threads. It first initializes the G900 board, then spawns the loader, interrupt handler and scheduler threads. It also sets up a server socket for incoming connection requests from applications and waits for requests. A new *application service thread* is created for each requesting application, which then interacts with the application on behalf of the RM. The main thread loops back to listen for new requests. Communication among the different threads of

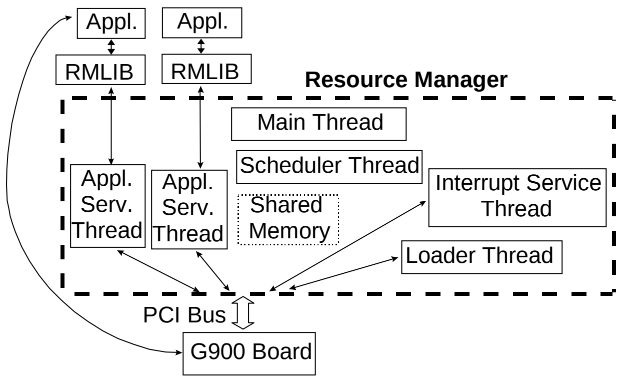


Figure 8. Overview of resource manager

the RM is accomplished through events, shared variables and shared memory.

The *application service thread* establishes a stream socket connection with its client application and services its requests. It receives the application flow graph and puts the graph into the shared memory and notifies the scheduler. Depending on the type of request sent from the application, the application service thread responds in different ways. There are six types of requests that can be sent from the application: *Load Graph Node*, *Input Data*, *Request Result*, *Execute Function*, *Release XMOD*, and *Release Flow Graph.*

When an application executes an FPGA function, it normally blocks until the function is completed. Because an FPGA function may use multiple XMODs, the *interrupt handler thread* of the RM checks for the completion of an FPGA function by servicing interrupts from the G900 board. Whenever an interrupt is received, the thread checks which XMODs have generated an interrupt, since more than one XMOD could be interrupting at a time. If all the XMODs for an FPGA function have completed, the thread then informs the corresponding application service thread.

Once all the interrupts have been acknowledged by their respective application service threads, the interrupt handler enables further interrupts and loops back to wait till another interrupt occurs. For each graph node, an application developer needs to either implement an interrupt request circuit in the FPGA designs or let the host program wait for a pre-specified amount of time for the function to complete. The latter approach works only if the function completion time can be known in advance or can be determined in a well formulated way.

The *scheduler thread* which allocates XMODs either on-demand or speculatively normally sits idle until being "triggered" by one of three different types of events from an application service thread: (1) a request for demand loading, (2) the de-allocation of XMODs due to the release of a graph node, or (3) the receiving of a new flow graph. Depending on the type of event, its scheduling parameters and availability of resources, the scheduler either assigns an XMOD to the loader thread for loading or loops back to wait for another event to occur.

3.2. Modifications to the Resource Manager

Previously the RM was designed based on the assumption that each application contains one single thread [10, 11]. This assumption is not true for the IR ATR application because the ATR host program has two threads, a child thread that computes Round 0 and a parent thread that creates the child thread and computes Round 1 and the rest. As a result, the same socket is used for communication between each thread and the RM application service thread that services these two threads. To synchronize the inter-process communication, the socket communication is treated as a critical section and guarded by a semaphore. An alternative is to create one application service thread for each application thread instead of each application. In this case, two different sockets are used between an ATR host program and the RM and no synchronization problem exists. However, it was found that the overhead of creating the extra thread and socket was usually slightly higher than that of using semaphores. The final implementation uses the semaphores.

Another modification to the RM is to improve the efficiency of the inter-process communication. After the IR ATR application was ported to execute with the RM, the application slowed down drastically. The reason was because the application sent requests to the RM close to a hundred times via a RM library function. The socket communication between the application and the corresponding application service thread had the following communication pattern.

Application	**Application Service Thread**
1. write_A	
	2. read_A
3. write_B	
	4. read_B
	5. write_C
6. read_C	

As part of the stream socket protocol the socket sent a *hidden* acknowledgement from the application service thread to the application after the read_A operation. This acknowledgement was extremely slow, when there was no explicit writing to the socket after read_A on the application service thread side. To solve the problem, the first two messages (i.e., A and B) were combined into one single message as follows.

Application	**Application Service Thread**
1. write_A_B	
	2. read_A_B
	3. write_C
4. read_C	

In this way, the *hidden* acknowledgement from the application service thread after the read_A_B operation can be attached to (or "piggy-back") the write_C operation. Another solution is to disable the Nagle algorithm on the application side (see pages 267–273 in [12]). With either solution, the inter-process communication and the resulting RM library function become much faster. Note that this particular RM library function is used by Round 0 but not by Round 1.

3.3. IR ATR with the RM

In order for the IR ATR application to work with the RM, the host program was modified to incorporate several RM library function calls. There was no need to change the FPGA designs. The modification was successful and the IR ATR program was able to run concurrently with a Boolean satisfiability program that was described in [11]. (Different designs for the Boolean satisfiability problem can be found in [13, 14].) The capability was achieved at the expense of longer execution time. For example, when six XMODs are used for the four test images, Round 0 in average takes 0.641 seconds with the RM instead of 0.614 seconds without using the RM.

Figure 9 shows the execution times with the RM for the fourth test image. The horizontal axis of the figure is the number of XMODs used for Round 0. There are two observations of the figure. First, in most cases, the total execution time is smaller than the summation of Round 0 time and Round 1 time. This is due to the overlapping of Round 0 and Round 1 computation with the FPGA hardware and with each other. This is especially clear in the figure when one XMOD is used. Second, as the number of available XMODs decreases, the application gracefully degraded. This is because When a smaller number of XMODs is allocated to Round 0, all the other rounds get to share a higher PCI bus bandwidth. As a matter of fact, the figure indicates that allocating

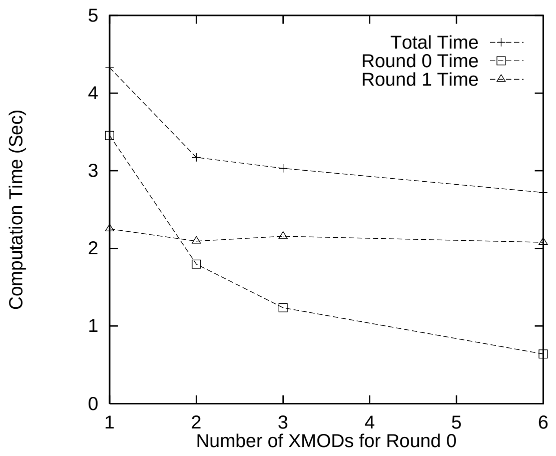


Figure 9. The execution times in seconds on top of the RM with different number of XMODs for Round 0

two XMODs to Round 0 may be a better strategy than allocating six when the G900 board is shared with other concurrent applications.

4. Conclusions

An infrared automatic target recognition (IR ATR) application is accelerated by implementing its bottleneck sections on a co-processor board that contains multiple field programmable gate array (FPGA) chips. The FPGA design explores parallelism at the template and the pixel levels and achieves a speedup of 21 compared to host processor execution of Round 0, the most computationally demanding routine in the application. A preliminary FPGA design was implemented for Round 1, the next most demanding routine. While this preliminary design does not provide a performance improvement, it gives us a baseline CLB count and provides performance information that can be used to improve the design. The paper then describes an FPGA resource manager (RM) developed to support concurrent applications sharing the FPGA board. We demonstrated that the IR ATR application could run concurrently with other FPGA applications when managed by the RM. The resulting ATR design was able to adapt to the amount of reconfigurable hardware that was available at the time the application was executed, with a gradual performance degradation as resources were reduced.

Acknowledgments

This research was supported by DARPA under Air Force contract number F33615-97-1-1148, an Ohio State investment fund, and an Ohio State research challenge grant. Xilinx Inc. donated an FPGA design tool and FPGA chips on XMODs. The authors would like to thank Mr. Jim Hilger with the US Army Night Vision & Electronics Sensors Directorate for his introduction to the IR ATR application and his help and comments on the design.

References

1. J. Villasenor, B. Schoner, K. Chia, C. Zapata, H. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, “Configurable Computing Solutions for Automatic Target Recognition,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 70–79, 1996.
2. M. Alderight, E.L. Gummati, V. Piuri, and G.R. Sechi, “A FPGA-based Implementation of a Fault-Tolerant Neural Architecture for Photon Identification,” in Proc. of ACM/SIGDA International Symposium on FPGAs, pp. 166-172, 1997.
3. J.G. Eldrege and B.L. Hutchings, “Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-based FPGAs,” in Journal of VLSI Signal Processing, Volume 12, pp. 67-86, 1996.
4. M.J. Wirthlin and B. L. Hutchings, “Sequencing Run-Time Reconfigured Hardware with Software,” in ACM/SIGDA International Symposium on FPGAs, pp. 122-128, 1996.
5. S. Singh and R. Slous, “Accelerating Adobe Photoshop with the Reconfigurable Logic,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1998.
6. M. Shand and L. Moll, “Hardware/Software Integration in Solar Polarimetry,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1998.
7. W.E. King, T.H. Drayer, R.W. Conners, and P. Araman, “Using MORRPH in an Industrial Machine Vision System,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1996.
8. J.D. Hadley and B. L. Hutchings, “Designing A Partially Reconfigured System,” in FPGAs for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607, pp. 210-220, 1995.
9. J. Burns, A. Donlin, J. Hogg, S. Singh, and M. Wit, “A Dynamic Reconfiguration Run-Time System,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 66-75, 1997.
10. V. Yavagal, “A Resource Manager for Configurable Computing Systems”, Master’s thesis, Wright State University, July 1998.
11. J.S.N. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook, “Dynamic Reconfiguration to Support Concurrent Applications,” in IEEE Transactions on Computers, Special Issue on Configurable Computing, Vol. 48, No. 6, pp. 591–602, June 1999.
12. W.R. Stevens, *TCP/IP Illustrated*, Volumn 1, Addison-Wesley, Reading, MA, USA, 1994.
13. A. Rashid, J. Leonard, and W.H. Mangione-Smith, “Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability,” in IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 196–204, April 1998.
14. P. Zhong, M. Martonosi, P. Ashar, and S. Malik, “Accelerating Boolean Satisfiability with Configurable Hardware,” IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 186–195, April 1998.

Authors' Vitae

Jack Shiann-Ning Jean
received the B. S. and M. S. degrees from the National Taiwan University, Taiwan, in 1981 and 1983, respectively, and the Ph.D. degree from the University of Southern California, Los Angeles, C.A., in 1988, all in electrical engineering. In 1989, he received a research initiation award from the National Science Foundation. Currently he is an Associate Professor in the Computer Science and Engineering Department of Wright State University, Dayton, Ohio. His research interests include parallel processing, reconfigurable computing, and machine learning.

Xuejun Liang
received his M. S. degree in Mathematics from Beijing Normal University, Beijing, China, in 1985. From 1985 to 1997, he taught Mathematics at Beijing Normal University. He was a visiting scholar at Wright State University in 1996. He is currently working towards his Ph.D. degree in Computer Science and Engineering at Wright State University. His current research interests include adaptive computing system and FPGA application.

Brian Drozd
is a graduate research assistant at Wright State University, Ohio, USA. His research interests include artificial intelligence and software architecture. He received the MS (1998) from Wright State University in Computer Science.

Karen Tomko
has been an assistant professor at Wright State University since January of 1996. She earned her doctorate degree at the University of Michigan in 1995. She received the M.S.E. and B.S.E. degrees from the University of Michigan in 1992 and 1986 respectively. From 1986-1992 she developed image processing and system software for Synthetic Vision Systems and the Environmental Research Institute of Michigan. Her research interests are adaptive computing systems, parallel computing, application optimization, and graph partitioning.

Yan Wang
received a B. S. degree in Applied Physics and a M. S. degree in Computer Science from JILIN university, P.R.China, in 1993 and in 1996, respectively. Since 1998 she has been a Ph.D student of the Department of Computer Science and Engineering at Wright State Uni-

versity. Her primary research interests are Reconfigurable Computing and Distributed and Parallel Computing.