The Journal of Supercomputing, 30, 213–238, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Collaborative and Reconfigurable Object Tracking

SOHEIL GHIASI soheil@cs.ucla.edu
HYUN J. MOON hjmoon@cs.ucla.edu
ANI NAHAPETIAN ani@cs.ucla.edu
MAJID SARRAFZADEH majid@cs.ucla.edu
Computer Science Department, University of California, Los Angeles

Abstract. Many Applications perceive visual information through networks of embedded sensors. Intensive image processing computations have to be performed in order to process the perceived information. Such computations usually demand hardware implementations in order to exhibit real time performance. Furthermore, many of such applications are hard to be characterized a priori, since they take different paths according to events happening in the scene at runtime. Hence, reconfigurable hardware devices are the only viable platform for implementing such applications, providing both real time performance and dynamic adaptability for the system.

In this paper, we present a collaborative and dynamically adaptive object tracking system that has been built in our lab. We exploit reconfigurable hardware devices embedded in a number of networked cameras in order to achieve our goal. We justify the need for dynamic adaptation of the system through scenarios and applications. Experimental results on a set of scenes advocate the fact that our system works effectively for different scenario of events through reconfiguration. Comparing results with non-adaptive implementations verify the fact that our approach improves system's robustness to scene variations and outperforms the traditional implementations.

Keywords: reconfigurable computing, embedded systems, dynamic adaptation, object tracking, feature selection

1. Introduction

Today's advances in technology have enabled the integration of processing resources, memory blocks and sophisticated I/O units into various electronic devices including data acquisition units [23, 26]. Such *networked embedded systems* provide the opportunity of processing the perceived information locally at the sensor nodes as opposed to more traditional approach of transferring the data to a remote processing station, having the station perform the computation and reading the result back. These two approaches to data processing, namely locally embedded at the sensor nodes and traditional communication-based computing schemes, introduce many trade offs into the design space [7].

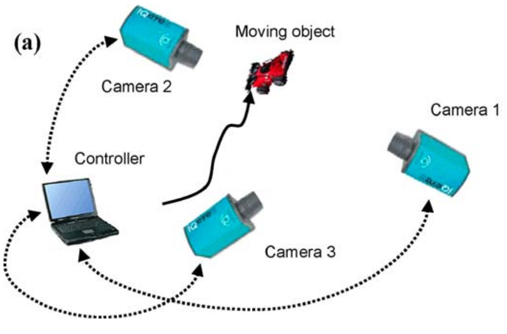
An example of the aforementioned systems is a network of vision sensors deployed to autonomously detect specific events. The application of such a system is generally referred to as "unsupervised detection of events", which is widely used in many different high-level applications [6]. Such applications require the system to automatically detect the events happening in its surrounding area and take proper actions according to these events. Moreover, real time response to external events is usually another requirement, due to the nature of the applications; examples of which include traffic management and intelligent intruder detection. Various image-processing algorithms have been developed for this class of applications. These algorithms usually perform intensive computations and hence, require powerful computational resources in order to comply with the real time performance requirement.

Image processing algorithms generally perform very intensive computations. Therefore, many constrained embedded processors dedicated to image data collection and/or processing cannot meet the real time performance constraint. However, most of image-processing algorithms perform similar local computations for all of the pixels of an image. Therefore, they are considered as *intrinsically parallel* computations that exhibit substantial speedup when implemented on a dedicated hardware unit. Hardware implementation is the only viable solution for most of the real time image-processing systems. Researchers have reported many efficient hardware implementations of such algorithms with significant speedups over pure software implementations [2, 3, 5, 8, 12, 16, 20].

Moreover, the image-processing algorithms implemented in a system work based on some assumptions. Examples include the number of moving objects, their shape and their motion type. Based on the events happening in scene, these assumptions might become invalid. For example, KLT tracking scheme [14, 21, 24] assumes that the moving object moves across the camera and its size doesn't change from the camera point of view. However, if the object moves towards the camera, this assumption is no longer valid and KLT tracking will not be effective anymore. Therefore, the system has to be able to adapt to external events. The external events are hard, and in some cases impossible, to be pre-characterized. Hence, it is practically impossible to determine the required algorithms a priori [10].

The aforementioned arguments, introduce the reconfigurable fabrics as the only viable solution for implementing such applications. Reconfigurable hardware units not only demonstrate real time performance by exploiting the intrinsic parallelism of image processing algorithms, but also provide the required flexibility and adaptability for the system. This cannot be achieved by traditional pure software or hardware implementations.

Figure 1(a) depicts an intruder detection and object tracking system that has been built in our lab as part of this work. The system consists of multiple IQeye3 cameras [11]. The cameras are connected to the local area network and communicate with each other or the control unit in order to collaborate and share their information. An outline of the architecture of one



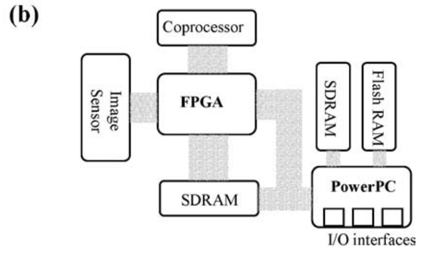


Figure 1. (a) The implemented target tracking system using IQeye3 cameras with embedded processors and FPGAs. (b) IQeye3 camera architecture. Courtesy of IQinVision Inc.

of IQeye3 cameras is demonstrated in Figure 1(b). A Xilinx Virtex1000E[1] [27] FPGA and a general-purpose processor (IBM PowerPC) are embedded in all of the cameras. The FPGA and the processor can both be used to implement the comprising blocks of an application.

Traditional implementations of similar systems use “dumb” cameras as vision sensors. Therefore, scene data is transferred to a powerful computation station that performs image manipulation. This approach has a number of negative implications. Firstly, communication overhead can be significant for large image sizes and slow networks and can harm the real time performance requirement. Secondly, the centralized computation entity must be capable of performing many intensive computations in a short amount of time; hence, it imposes significant hardware costs. Moreover, such a system is vulnerable to scalability and robustness issues. Therefore, the system can handle a limited number of cameras and will fail with degradation in server performance.

On the other hand, our system employs “smart” cameras that have computation resources locally embedded at the sensing location. Therefore, perceived scene data is processed locally at the sensor node and hence, only a limited amount of data is communicated with the controller. This reduces the overall system cost, improves its performance and facilitates its scalability compared to the traditional implementations. Moreover, we utilize the reconfigurable device embedded in each camera to dynamically adapt the system to the external events. To the best of our knowledge, this idea has not been practiced in the domain of collaborative detection of events before. The two aforementioned facts, i.e., distributed local computations and dynamic system adaptation form the major contributions of this paper.

This project took about two years as two people have been working on development and implementation of the system, algorithms and mechanisms. In this paper, we focus on the crucial role of the embedded reconfigurable devices (i.e. Xilinx FPGAs) in our system. We exploit these devices to achieve both real-time performance and dynamic adaptability of the system to the external events. We compare our system’s performance with a traditional non-adaptive one and verify that our system outperforms its competitor in terms of tracking quality.

We proceed to describe our system framework and its application in the next section. In Section 3, we present the image-processing algorithms that are required for the implemented tracking application. In addition, the effect of environment changes on these algorithms and hence, the need for system adaptability is explained in this section. Experimental results including algorithms implementation and their performance for some scenes, has been presented in Section 4. Finally, Section 5 outlines the conclusions and future directions of this paper.

2. Networked reconfigurable tracking system

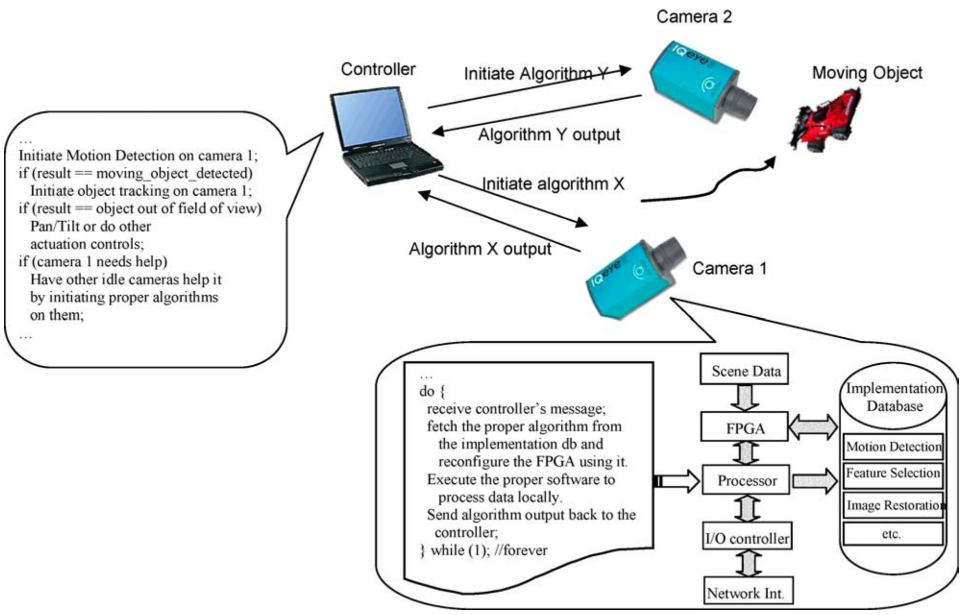
In this section, we present the reconfigurable tracking system that has been built as part of this work. First, we present the framework of our system along with its application. In next section, we discuss the algorithms that are needed for implementing the system application. We describe how these algorithms have to be tuned based on the changes in the scene and hence, highlight the “*reconfigurability*” feature of our tracking system.

2.1. *System framework*

The hardware framework for our system is comprised of several components including: IQeye3 cameras provided by IQinVision [4], pan-tilt units to enable the actuation of the cameras, a PC serving as the main controller, and a network that connects the cameras and the controller and allows them to communicate and collaborate. Figure 2 illustrates a simple view of the system framework.

An IQeye3 camera, as a “smart” vision sensor with embedded computation resources, allows input image data acquisition and processing to be collocated in the camera, which minimizes network communication overhead and facilitates scalability. The processing resources embedded in each camera include a Xilinx Virtex 1000E FPGA and a 250 MIPS PowerPC CPU (Figure 1(b)). In addition, there is 4 MB of Flash RAM and 16 MB of SDRAM on each camera. Each IQeye3 camera gives full access to raw real-time image data streams and the general-purpose processor can be used for customization since a large “C” development library is available to application developers. Full networking functionality is provided by each IQeye3 camera through an Ethernet connection. It can communicate using TCP, UDP, and IP.

In addition, the IQeye3 camera can send and receive 230 Kbps over a 9-pin RS232C serial port. By supporting such communication standards, the IQeye3 cameras can be placed in various environments; while the raw and/or processed captured images can be accessed



```
...
Initiate Motion Detection on camera 1;
if (result == moving_object_detected)
  Initiate object tracking on camera 1;
if (result == object out of field of view)
  Pan/Tilt or do other
  actuation controls;
if (camera 1 needs help)
  Have other idle cameras help it
  by initiating proper algorithms
  on them;
...
```

```
...
do {
  receive controller's message;
  fetch the proper algorithm from
    the implementation db and
    reconfigure the FPGA using it.
  Execute the proper software to
    process data locally.
  Send algorithm output back to the
    controller;
} while (1); //forever
```

Figure 2. An overview of the tracking system architecture: Each camera has a set of the required configurations available. The controller communicates with the cameras via an implemented message passing scheme and can initiate the proper algorithm on each camera, organizing the collaboration among cameras.

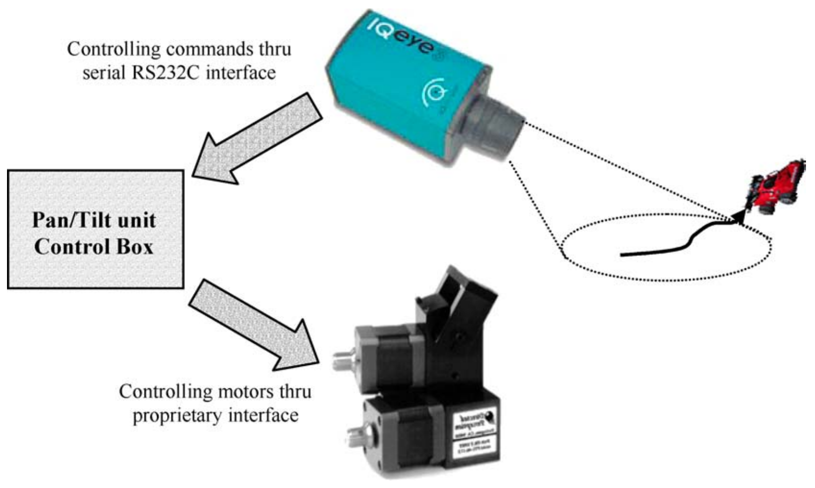

Controlling commands thru serial RS232C interface
Pan/Tilt unit Control Box
Controlling motors thru proprietary interface
IQeye

Figure 3. Each camera is mounted on a pan/tilt unit (PTU). When the object moves out of the field of view of the camera, it sends controlling commands to the PTU thru its serial interface. Consequently, PTU moves the camera in order to be able to track the object.

remotely. In our system, each IQeye3 camera is mounted on a pan-tilt unit, which is directly controlled by the corresponding camera via its RS232C serial interface. A pan-tilt actuation unit can be controlled using simple commands that specify the pan/tilt angle/speed/acceleration. Figure 3 illustrates the need for actuation control when an object moves out of the field of view on camera. The flow of commands from a camera to its corresponding pan-tilt unit is demonstrated.

Figure 2 demonstrates our system with two cameras and the main controller. The main supervisory controller resides on an ordinary small computer and acts as the centralized governing unit of the system by maintaining the current state, processing internal and external triggers, and coordinating the collaboration among the cameras. When the main controller receives data from one of the IQeye3 camera clients over the network, it deterministically selects the appropriate actions that should be taken by each camera (e.g. reconfiguring an embedded FPGA by swapping in a different algorithm from the database). This is performed by sending a message to the designated camera. The two blocks close to the main controller and the lower IQeye3 camera in Figure 2 outline the functionality of the main controller along with the idea of "implemented algorithms database" and reconfiguration at the sensor node.

2.2. *System application*

The sample application implemented on the framework is to continuously detect and track a moving object that is within the field of view of a camera (Figure 2). We assume that the object is always moving across the camera and hence, KLT tracking scheme [14, 21, 24] can effectively track the motions. However, various parameterization and dynamic adaptations have to be performed in order to make the system robust to variations in light, objects' shape and location, etc.

If the object leaves the field of view of one camera, the camera should pan or tilt to maintain the object within its field of view or it should hand off control to another camera. Depending on the light, focus and other parameters, different algorithms are used to maximize the tracking performance.

When the entire system initializes, cameras establish a connection with the main supervisory controller on the PC. First camera assumes control initially and continuously runs feature selection algorithm on its embedded FPGA. Feature selection algorithm selects points in the scene that are appropriate for tracking. Sharp corners and local intensity variations in an image usually form good features. The selected features are passed to the KLT tracking algorithm to track their motion in consequent images. The tracking algorithm has to meet the real time performance constraint.

Feature tracking has to perform some computations for each selected feature and hence, the algorithm latency increases with the number of selected features. If the number of selected features is more than a certain upper bound, the algorithm will be so slow that it cannot meet the real time performance constraint. Furthermore, accuracy will be compromised if the number of selected features is not large enough. Therefore, it is desired that the number of selected features be within a certain range.

However, as the objects in the scene, distance of the object to the camera, light conditions, lens focus and other parameters change, the number of selected features varies. For example, two runs of the algorithm on a scene with two different lighting conditions will lead to selecting less number of features for the darker scene. Our implementation can detect such conditions and can adapt itself in order to compensate the effect of variations in the scene and environment. Therefore, it is ensured that the number of selected features, and hence both latency and tracking accuracy, are kept within a certain range. This is accomplished through reconfiguration and parameterization of the algorithms running on the embedded FPGA.

Furthermore, when a moving object moves close to the edge of the image, the camera detects this situation and sends a message to the pan-tilt unit to take the appropriate action to keep the moving object within its field of view. At a certain point, the pan-tilt unit will no longer be able to pan or tilt further and the moving object will move completely out of the field of view of the camera. The camera has to surrender complete control of the scene and another camera will be forced to monitor the scene. In this situation, the camera that can no longer monitor the scene notifies the main controller by sending a message indicating the position where the moving object is located. The main controller then decides which camera should gain control and sends the proper camera a message indicating where the object is. As a result, the camera issues commands to move the pan-tilt unit so that the moving object is in the field of view of the camera. Figure 2 outlines the architecture and application of the system. A sample pseudo code running on the controller and a high-level block diagram of each camera have been demonstrated.

In such a manner, the moving object is vigilantly tracked using multiple cameras. The use of reconfigurability in our system leads to the proper tradeoff between tracking quality and latency. Moreover, it improves the system robustness to changes in the scene such as lighting and moving objects variations. Note that by use of the “hands off” approach, the cameras can collaborate in tracking an object. The object will be continuously tracked as long as the object is within the field of view of a camera.

3. Vision algorithms overview

In this section, we present two algorithms that are required for enhancing the image quality and tracking the motions, i.e. image restoration and feature selection. First, we outline the algorithms’ underlying idea and functionality and then, we describe their sensitivity to the changes in the scene. Finally, details of the FPGA implementation in our system will be discussed.

3.1. Feature selection

In this work, we assume that the object is moving across the camera. Therefore, from camera point of view, the object in each frame is moved by a constant displacement compared to its immediately preceding frame. KLT tracking scheme [14, 21, 24], has been developed to track the objects that comply with the aforementioned motion. Note that this scheme cannot track rotations or size variations (when the object moves towards or away from the camera and its size changes from camera point of view).

KLT tracking scheme is carried out in two stages. In the first stage, called feature selection, a number of *trackable* points in the images are selected. These points, called features, show significant intensity changes compared to their neighboring pixels. Feature points are passed on to the second stage, feature tracking, in order to find their location in the consequent images. In our system, we have implemented the feature selection stage on the FPGA[2] and feature tracking is currently performed on the PowerPC embedded in the IQeye3 cameras.

Feature selection algorithm consists of carefully choosing the points in the image, which can be easily tracked throughout a series of images. Corner points of an object, where intensity changes noticeably, are considered as good feature points. The tracking stage looks in a small patch around the location of a feature in the preceding frame, in order to find its new location after possible motion. This process is repeated for all selected features. Therefore, the latency of tracking phase linearly grows with the number of selected features. On the other hand, due to various factors including variations in the intensity of two consecutive frames and noise, some features might be lost during tracking. Therefore, a minimum number of features are required to guarantee an accurate tracking. Hence, despite ever-changing parameters of the scene, controlling the number of selected features is required.

In summary, the feature selection algorithm performs the following operations for all of image pixels [3]:

1. Calculate g_x and g_y, the intensity gradients in the x and y directions for all pixels of the image. This is done by computing the Gaussian and Gaussian derivative kernel as well as convolving these kernels in the horizontal and vertical directions.

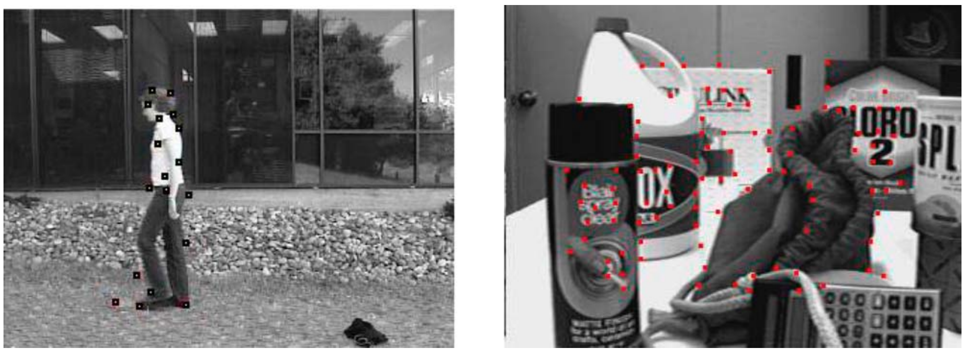


Figure 4. Sample outputs of feature selection algorithm run on a selected portion of the images. Features are denoted by black squares with white centers in the left image, and by filled dark squares in the right image.

2. Sum the gradients in the surrounding window of each pixel in order to compute the Z matrix, where

$$Z = \iint_W \begin{bmatrix} g_x^2 & g_x g_y \\ g_x g_y & g_y^2 \end{bmatrix} dx$$

3. Compute λ_1 and λ_2, the eigenvalues of the Z matrix. Let $\lambda_1 = \min(\lambda_1, \lambda_2)$. λ_1 represents the trackability of the pixel.
4. Given λ as the threshold value, If $\lambda_1 > \lambda$ then declare the pixel as a feature.

Figure 4 demonstrates the output of feature selection algorithm executed on a selected region of sample images. For example in the left image, a rectangular region around the walking girl has been chosen for selecting features. Note that the choice of two different threshold values has lead to selecting different number of features in two images. Features are denoted by black squares with white centers in the left image, and by red squares in the right image.

The number of selected features reduces with the increase of λ and vice versa. Therefore, points that are selected with higher values of λ are considered *better* features. Note that such features are also selected with small values of λ. These points are usually easier to track in consequent images. They exhibit significant intensity variation compared to their neighboring pixels.

Based on the main steps of the algorithm, it is easy to observe the effect of the changes in the scene on the number of selected features. Intuitively, increasing/decreasing the intensity value of the image pixels should increase/decrease the number of selected features with a constant λ. In reality, brighter/darker lighting can create such a case. Therefore, different number of features will be selected for a particular scene under different lighting conditions. Furthermore, the number of selected features heavily depends on the objects in the scene. A particular threshold value will select less number of features on a round object with a few sharp corners compared to a complex object with many sharp corners and intensity variations. In addition, other parameters such as lens focus and the number of objects in the scene can affect the number of selected features.

The feature tracking stage of the KLT tracking method locks onto the selected features and strives to locate them in the next upcoming frame. Note that this is performed with the assumption that the two consecutive images differ only by a small displacement factor. The tracked features will be tracked again in the future upcoming frames. Therefore, the displacement, motion direction, velocity and other information about the motion can be inferred.

3.2. Image restoration

Image restoration is a commonly used algorithm in image acquisition or processing for recovery of degraded images. Atmospheric turbulence, defocusing or motion of objects can be reasons of degradation. Restoration process recovers lost information of images by such degradation [4, 13, 25]. The following degradation model holds in a large number of applications [18]:

$$y(i, j) = d(i, j)^{**}x(i, j)$$

where $x(i, j)$ and $y(i, j)$ denote the original and observed degraded image respectively. $d(i, j)$ represents the impulse response of the degradation system, and ** stands for two-dimensional (2D) discrete linear convolution. The goal of image restoration is to estimate $x(i, j)$ given $y(i, j)$ and $d(i, j)$, however one of the main difficulties in performing an ideal image restoration is that the degradation model is not completely known. In other words, $d(i, j)$ is not exactly defined/known at the receiver. Therefore, it might not be able to completely reconstruct the image.

Noise signal injected into the image usually exhibits quick variations and hence, is considered high frequency signal. Therefore, common realizations of noise-removal filters implement a low-pass filter, which allows the image signal to pass and filters out the high-frequency noise. A low-pass filter has no effect on low frequency image data (pixels with small variations compared to neighboring pixels) and removes the high frequency elements of the signal. As a result, the sharp edges of an image passed through a low-pass filter become blurred while the solid textures remain intact. On the other hand, blurred and defocused images have to be passed through a high pass filter in order to be restored. The high pass filter restores such images by sharpening and/or preserving their edges. Figure 5 demonstrates a simple image and the result after applying a low pass and a high pass filter on it. Note that the high pass filter preserves the sharp edges, while the low pass filter blurs them out [19].

A common implementation of image filters places an imaginary 3×3 window with filter coefficients, over a pixel in the original image and calculates the new value of this pixel in the filtered image using its old intensity value and those of the neighboring pixels. Figure 6 demonstrates the idea of such an implementation. The coefficients used in the window, specify the type of filtering operation that the filter performs. Intuitively, positive coefficients take average of close pixels to calculate the new value of a pixel, and therefore blur sharp

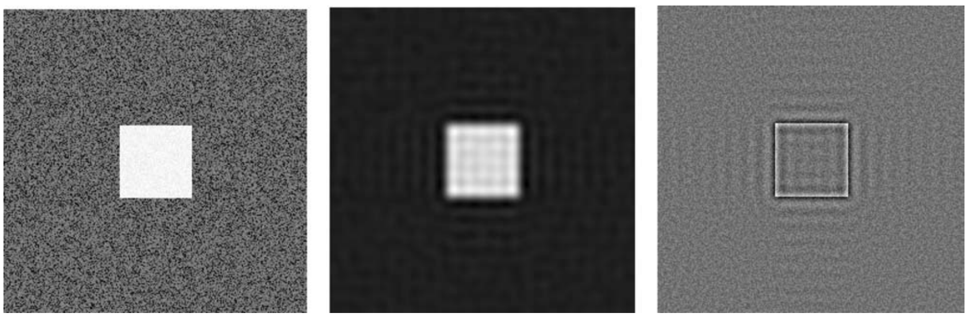
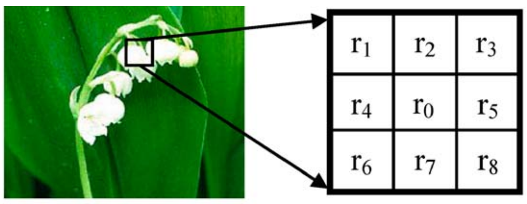


Figure 5. A sample image, its low pass, and high pass filtered versions are shown, respectively.[3] Note that the low pass filter removes quick variations in intensity and blurs out sharp edges, while the high pass filter preserves these elements.



$$P_0' = \sum_{i=0}^{8} r_i . P_i$$

1	1	1
1	8	1
1	1	1

A sample low pass filter

-1	0	-1
0	8	0
-1	0	-1

A sample high pass filter

Figure 6. A filter is applied on a pixel by replacing its value with a weighted combination of its old value and its neighboring pixels. A low pass filter typically has positive coefficients, while a high pass filter has negative coefficients for neighboring pixels. Coefficients can be normalized to keep the total intensity of the image intact.

edges. Therefore, they make low-pass filters while negative coefficients for neighboring pixels highlight the difference of the center pixel with its adjacent pixels and create a high pass filter (Figure 6). Usually, the total value of all nine coefficients is one, in order to keep the total intensity of the image intact.

The process of applying a filter on a pixel is repeated for all of the pixels in the image. Moreover, for some applications, the image is filtered many times until the residual value (the normalized amount of change between two consecutive images) is less than a given threshold. Experiments have shown that a certain number of iterations on the image, exhibit satisfactory quality for most of the scenes [18].

In our system, applying image restoration (or any other proper filter) before feature selection can enhance the image quality by sharpening the edges, and improve the quality of the selected features. Iterative application of the filter on the image requires the entire image to be accessible throughout the process. Conventional hardware implementations constantly retrieve the image from an attached memory unit and store the result back, however this is not possible in our constraint platform. In our system, the entire image is not available to the restoration module due to real time incoming stream of the scene data, which is not flow controllable. Therefore, we had to adapt the functionality of image restoration to our constrained platform. This will be thoroughly discussed in the next section.

4. Hardware implementations

In this section, we describe our system constraints and the modifications we had to make to the original algorithms in order to fit them to our platform. Moreover, we discuss the system adaptability issue and discuss its implications on hardware implementation. Throughout the paper, we assume that IQeye3 cameras, as discussed in Section 2, are the experimental platform of our system.

4.1. Platform constraints

As described in Section 2, IQeye3 camera is the vision sensor used in our platform. Three major components of IQeye3 are the imager, embedded FPGA chip and PowerPC. The imager continuously captures scenes and injects a real-time stream of image pixels into FPGA. The incoming stream of information is not flow controllable and runs at 24 MHz. The design residing on the FPGA (called the image processing pipeline in Figure 8(a)) performs several operations on the incoming stream such as image correction, windowing and down sampling. Finally, a DMA unit residing on the FPGA stores the processed scene data in the main memory. Any program running on the PowerPC can access the memory and scene data through regular software function calls. For example, a sample application running on the processor embedded in the camera implements an embedded web server that compresses the image data into jpeg format and exports the jpeg file through HTTP connection. Figure 8(a) visualizes the path that each pixel goes through in order to become available to software programs running on the processor.

Within this environment and platform, applications implemented on the FPGA need to meet a number of constraints. The most important issue is the timing constraint of the design, because the imager continuously generates real-time stream of image pixels and injects the flow into the FPGA. The applications implemented on the FPGA have to process the input stream and generate the corresponding output at the same rate to avoid congestion. This forces many designs to perform their intended computations with the small on-chip memory, because using the off-chip memory units will impose additional latency, which might not be tolerable for some designs. Consequently, we have implemented a modified version of the required algorithms that work with the limited available on-chip memory.

Furthermore, there is a basic design running on the FPGA at all times. This design performs basic necessary image manipulation functions such as windowing and packetizing. Any application being mapped onto the FPGA has to integrate with this design and has to cope with its communication standards and data formats. Therefore, the algorithms cannot be used in their original form and have to be adapted to our constrained platform.

For example, the aforementioned basic FPGA design processes the image stream in Bayer pattern [9]. Therefore, any other application has to comply with this constraint and perform its computation using Bayer pattern; or convert the Bayer pattern to any other desired format, perform the computation and convert the stream back to Bayer pattern. These two major constraints, namely limited amount of on-chip memory and complying with system existing format/standard conventions, impose significant overhead in implementing new designs on the system.

4.2. *Implementations*

In this subsection, we discuss the issues involved in implementing the required algorithms, i.e., feature selection and image restoration, on our constrained platform. In general, implementing an application on the IQeye3 camera is composed of hardware and software development. Each of these two portions of the design, require a particular development style and tool chain in order to be able to run the application on the camera. Figure 7

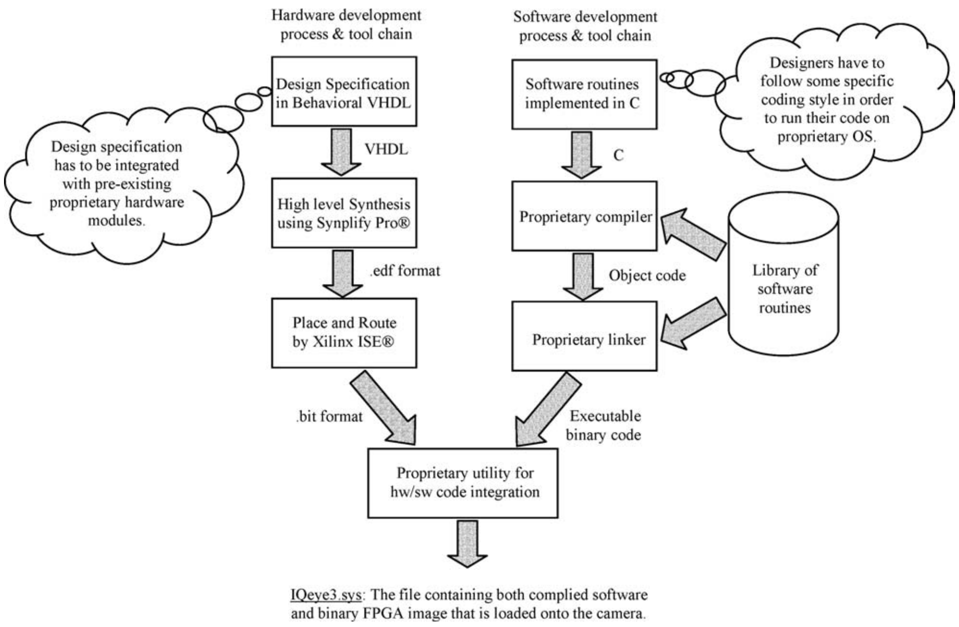

Hardware development process & tool chain
Software development process & tool chain
Design Specification in Behavioral VHDL
Design specification has to be integrated with pre-existing proprietary hardware modules.
VHDL
High level Synthesis using Synplify Pro®
.edf format
Place and Route by Xilinx ISE®
.bit format
Software routines implemented in C
Designers have to follow some specific coding style in order to run their code on proprietary OS.
C
Proprietary compiler
Object code
Proprietary linker
Library of software routines
Executable binary code
Proprietary utility for hw/sw code integration
IQeye3.sys: The file containing both complied software and binary FPGA image that is loaded onto the camera.

Figure 7. The process of developing a software and/or a hardware application for executing on IQeye3 camera, and the corresponding tool chain are illustrated.

illustrates the block diagram and the required tool chain for developing an executable application for the camera. Software development process, which is shown on the right column of the Figure 7, is similar to an ordinary software development flow except that the compiler and linker are tailored to the particular camera platform. Similarly, for hardware design development the process shown on the left column of Figure 7 has to be followed.

For all of our hardware implementations, design specifications have been done using a combination of RTL and behavioral VHDL. ModelSim VHDL simulator [17] has been used for simulation and debugging of designs. Architectural Synthesis has been carried out using Synplify Pro from Synplicity [22], which is one of the popular FPGA synthesis tools. The result of synthesis has been saved as an EDF file.

The generated EDF format has been passed to physical synthesis stage. The physical synthesis stage, including clustering, mapping, placement, and routing has been done using Xilinx ISE package. All tools have been targeted for our embedded FPGA devices (Xilinx Virtex1000E). Finally, The FPGA chips embedded in the cameras have been programmed using the generated configuration files. Therefore, all of the designs are physically implemented on our platform and experiments are performed with actual scenes to verify the designs functionality and performance in action.

4.2.1. Feature selection. Feature selection algorithm has been implemented on the same platform in a previous work [3, 15]. This implementation only needs to store two rows of the image data on-chip before deciding whether a pixel is a feature or not. The algorithm performs local computations in a 3×3 window around a pixel and compares the result with a *fixed* threshold for determining features. The value of threshold used in this implementation has to be specified at design time. Then, the design undergoes conventional architectural and physical synthesis phases and the resulting FPGA configuration stream is mapped onto the FPGA embedded in the camera.

While this implementation works well in practice, it does not have any control on the number of selected features. Moreover, the value of threshold cannot be altered easily. The feature selection's threshold has been implemented as a constant, which should be specified at design time. Therefore, altering the threshold forces the designer to repeat the entire design flow, which can take up to 30 minutes and is not tolerable for real time applications.

Various parameters such as objects' shape, scene light and lens focus can affect the number of selected features. As mentioned before, the selected features are passed to the tracking phase. The latency of the tracking grows, while its accuracy drops, with the increase of feature count. Therefore, the number of selected features has to be controlled in order to maintain a proper tradeoff between tracking latency and its accuracy.

We have started from the implementation in [15] and have modified the original design such that the threshold value can be controlled by a program running on camera PowerPC at runtime. Specifically we have developed registers that can be read/written by a software program running on the PowerPC. The hardware design has also been modified to read its threshold value from the register, without losing its synchronous operation with other parts of the basic design. Note that, the software program can alter the register contents at any time during processing of a frame and therefore, the design has to be able to handle asynchronous incoming events.

Our implementation can dynamically tune the feature selection algorithm running on the FPGA. According to the algorithm, if the threshold used in feature selection is too low for a particular scene, we get too many features and if the threshold is too high, we get too few features. Therefore, given a target number of features desired, we increase the threshold if we get features more than the target and decrease if we get less.

Note that the actual feature selection performs its computations on the FPGA and exhibits real time performance. The threshold controlling entity is a small program running on the camera PowerPC, which counts the number of selected features and controls the threshold value accordingly.

4.2.2. Image restoration. Image restoration has a variety of implementations and iterative method is a widely used one. The purpose is to estimate the original image given the degraded image. Common restoration methods perform operations on the entire image iteratively. Following each iteration, the normalized difference between current and immediately preceding image, called residual value, is calculated. Iterations are stopped when the restored image converges with insignificant residual ε [18].

As discussed in Section 4.1, our constrained platform does not allow the entire image to be stored on the FPGA. On the other hand, accessing the off-chip memory iteratively will impose additional latency on the algorithm, which is not affordable because of the real time performance constraint of system applications.

We have made several modifications to adapt the original method to our environment. Firstly, instead of globally iterating over the entire image, we iterate over local windows, where the size of window can be from 3×3 to the entire image. As the window gets smaller, the restoration quality drops since the center pixel does not have any information about pixels out of the restoration window. However, this enables processing of image stream using a small-sized storage.

Figure 8(a) illustrates the path that each image pixel goes through to be processed in our system cameras. Image sensor converts the scene into a non flow-controllable stream of pixels flowing into the FPGA. The proprietary image processing pipeline implemented on

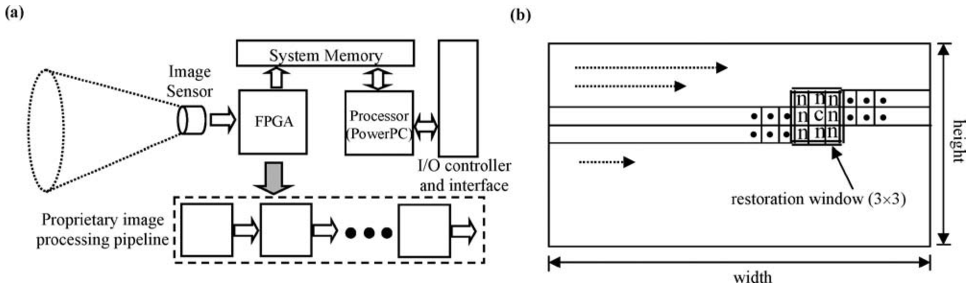


Figure 8. (a) Block diagram of the camera illustrating the path each image pixel goes through in order to be processed. The image processing pipeline residing on the FPGA is not disclosed due to copyright issues. (b) Restoration window implemented as one of the blocks in image processing pipeline. Pixels stream in starting from the upper left corner of the image.

the FPGA performs various computations on the incoming flow of pixels and finally stores the result in the system memory, where the software applications running on the camera processor (PowerPC) can access it.

The image restoration algorithm has been implemented as one of the stages in the pipeline (Figure 6(b)). Therefore, it does not have access to the entire image pixels at any point of time (assuming no off-chip data communication). Note that image pixels are revealed to the system starting from upper left corner of the image flowing to the right. When a row is finished, the flow of pixels moves down a row and again start from left to right. As Figure 6(b) visualizes, the amount of memory required for implementing a 3×3 restoration window is a bit larger than two rows of the image. The FPGA devices embedded in the system cameras have enough BlockRAMs available to store two rows of the image on-chip. Therefore, the restoration algorithm can be performed without any off-chip communication.

In general, for a restoration window of size $n \times n$, $((n - 1).\text{rows} + n)$ pixels need to be stored on the chip. Each FPGA device contains a certain number of logic blocks and BlockRAMs. Hence, the window size cannot grow beyond physical limitations of the target FPGA. For example, our system's embedded FPGA (Xilinx Virtex1000E) allows the window size to grow up to 15 for processing the widest images. The maximum width of images in our system is 1280 pixels.

In addition, we have unrolled local iterations of the algorithm on a 3×3 window a priori, and therefore, the current implementation performs an equivalent but more efficient computation for restoration of each pixel. Current implementation performs a single step evaluation of each window in order to calculate the new value of the center pixel, as opposed to iterating over the window. Table 1 summarizes the area improvement of the unrolled implementation compared to the original implementation, which iterates 40 times over each pixel.

Researchers in [18] have studied tradeoffs of restoration performance and quality with changes in restoration window size. According to their work, 3×3 restoration windows reflect reasonable restoration quality for many applications. The restoration algorithm used in this work, implements a high pass filter with 2 and -0.125 as coefficients for center and neighboring pixels, respectively.

Varying the restoration window size, leads to accuracy-memory requirement tradeoff. Small restoration windows need smaller on-chip storage, however their quality is not as good as larger restoration windows. On the other hand, larger windows improve the restoration quality at the price of higher memory requirement. Note that memory requirement inversely correlates with the system performance.

Table 1. The breakdown of hardware resources used by different portions of the designs. Note that the unrolled version of image restoration frees up 2% of block RAMS and 11% of CLBs for Xilinx1000E device

Implemented design	BlockRAM	CLBs
Basic design + Feature selection	51 out of 96 (53%)	9170 out of 24576 (37%)
Basic design + Feature selection + Original image restoration	66 out of 96 (68%)	12278 out of 24576 (49%)
Basic design + Feature selection + Unrolled image restoration	64 out of 96 (66%)	9454 out of 24576 (38%)

5. Experiments

In this section, we present the framework and results of our experiments. First, we describe the platform and designs used in conducting the experiments. We address the issue of dynamic system adaptation to the environment variations in this section. Then, we present the results of our approach for a number of scenes and compare them with a traditional non-adaptive system results.

5.1. Experimental setup

We have implemented the feature selection and image restoration algorithms (discussed in Sections 3 and 4) on IQeye3 cameras. The threshold value in the feature selection algorithm can be dynamically adjusted through a software program running on the PowerPC of the camera.

Furthermore, the implemented image restoration algorithm can be dynamically disabled or enabled through system reconfiguration. If the quality of the image is not good enough, then the FPGA will be reconfigured to enable the image restoration before feature selection. The quality of images can be determined by examining the value of the threshold in feature selection for selecting a certain number of features. Lower threshold values correspond to lower quality features and blurred corners. On the other hand, image restoration can alter the original image if it is not degraded to some degree. Therefore, we need to disable it for cases that the image quality is reasonable.

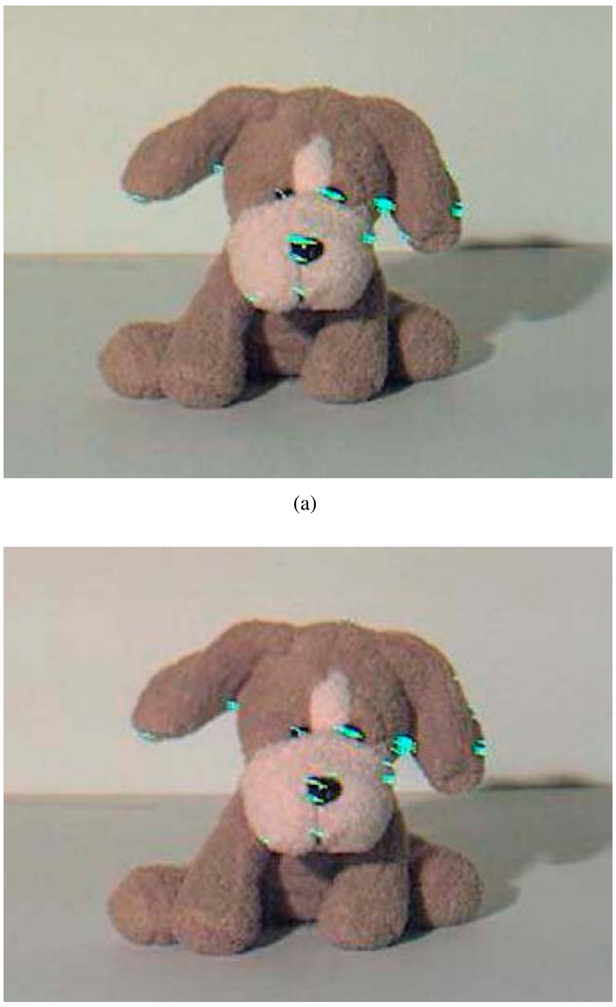
5.2. Experimental results

In the following sets of experiments, we examine the effect of our proposed techniques. The first two sets of experiments demonstrate the quality of automatically adjusted threshold compared to the original fixed threshold feature selection. The third experiment shows how image restoration can affect the performance of feature selection. In all experiments, automatically adjusted threshold targets for 150 features with 10% tolerance range, i.e. the number of selected features should be in the (135–165) range.

One example, where dynamically adaptive feature selection finds its use, is in the environments with variations in lighting. This applies to outdoor places where the natural lighting changes throughout the time. Another example is indoor scenes under various lighting conditions. For the first set of experiments, we varied the lighting condition in the laboratory and observed the results of the feature selection application.

Figures 9(a), 10(a) and 11(a) show the result of feature selection with fixed threshold, called FS-FIX, for an object under three different lighting conditions. Figures 9(b), 10(b) and 11(b) show the results of feature selection with automatically adjusted threshold, called FS-AUTO, for the same object and lighting conditions.

Figure 9(a) and (b) show the result of both FS-FIX and FS-AUTO under normal lighting. Both implementations select about 150 features (with 10% tolerance). Figure 10(a) and (b) illustrate the same object under similar lighting, which is brighter than the previous settings used in Figure 9. Extra brightness causes edges and corners to have greater intensity



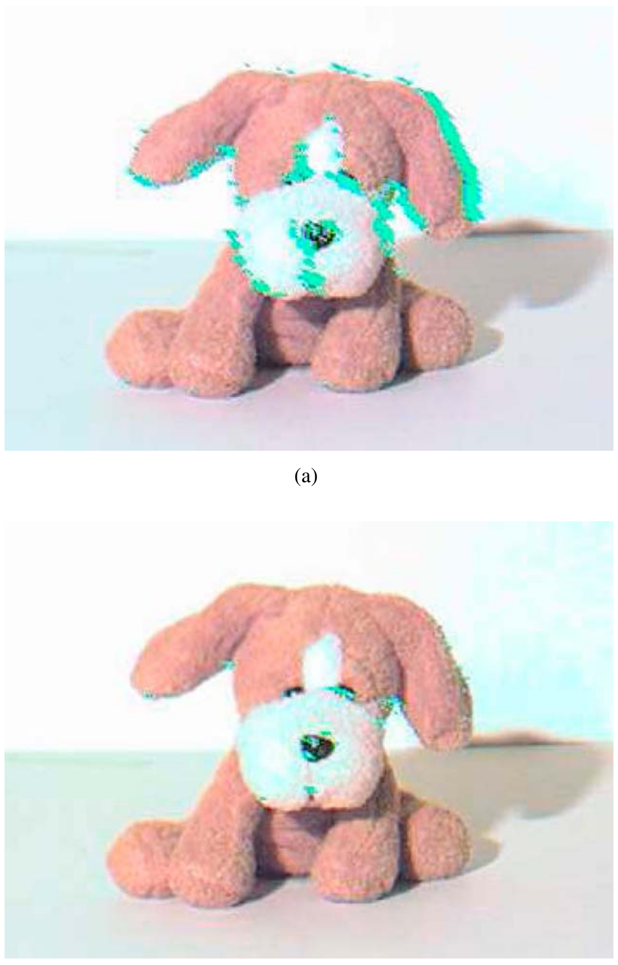
(a)

(b)

Figure 9. (a) 152 features are selected by fixed threshold (FS-FIX) under default lighting conditions. (b) 148 features are selected by automatic threshold adjustment (FS-AUTO) under normal lighting.

difference from their adjacent pixels, therefore a larger number of points are chosen as features. Figure 10(a) shows many unnecessary features chosen whose count is 1150. This is too many compared to the target feature count, 150. FS-AUTO increases the threshold value from 512 to 1552 and chooses 150 features in Figure 10(b). It selects features at almost same locations as in Figure 9(b) even after the significant change in brightness.

Figure 11(a) and (b) are taken under dark lighting. The object is observable by eyes, but FS-FIX is unable to find any features, since the intensity variations are not large enough for the fixed threshold value. However, FS-AUTO successfully decreases the threshold value


(a)

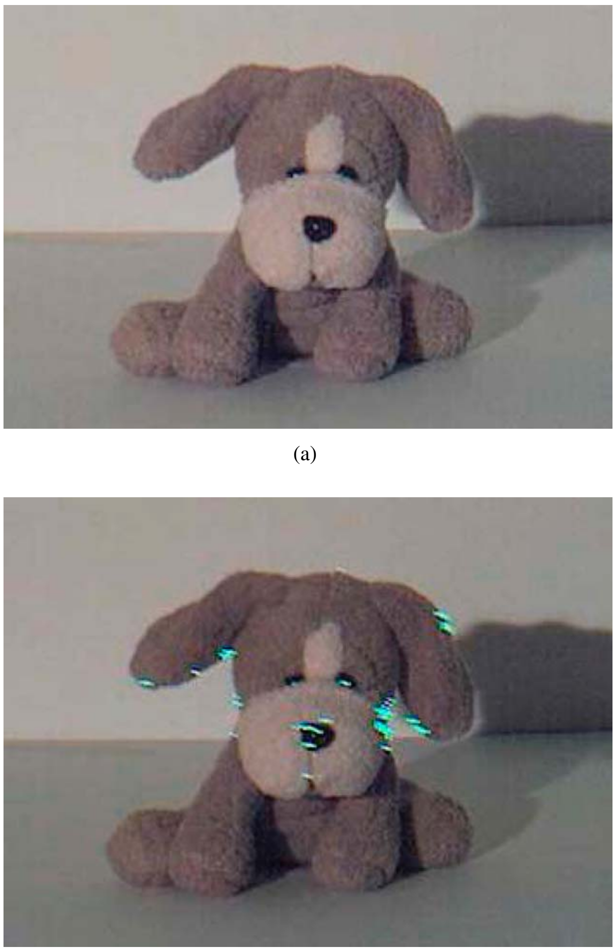
(b)

Figure 10. (a) 1150 features are selected by fixed threshold (FS-FIX) under bright lighting. (b) 150 features are selected by automatic threshold adjustment (FS-AUTO) under bright lighting.

from 512 to 160 and finds 156 features. Locations of features are almost same as Figures 9(b) and 10(b).

The aforementioned set of experiment verifies the efficiency of our approach in implementing a system robust to lighting variations through dynamic adaptation of the system. However, the advantage of our implementation is not limited to handling lighting variations. We have carried out another set of experiments to show that this technique can assist in handling other realistic scenarios, such as object's shape variations and multiple object cases.

Figures 12(a) and 13(a) show two different objects that have been processed by FS-FIX to select some features on them. As expected, FS-FIX has no control over the number of

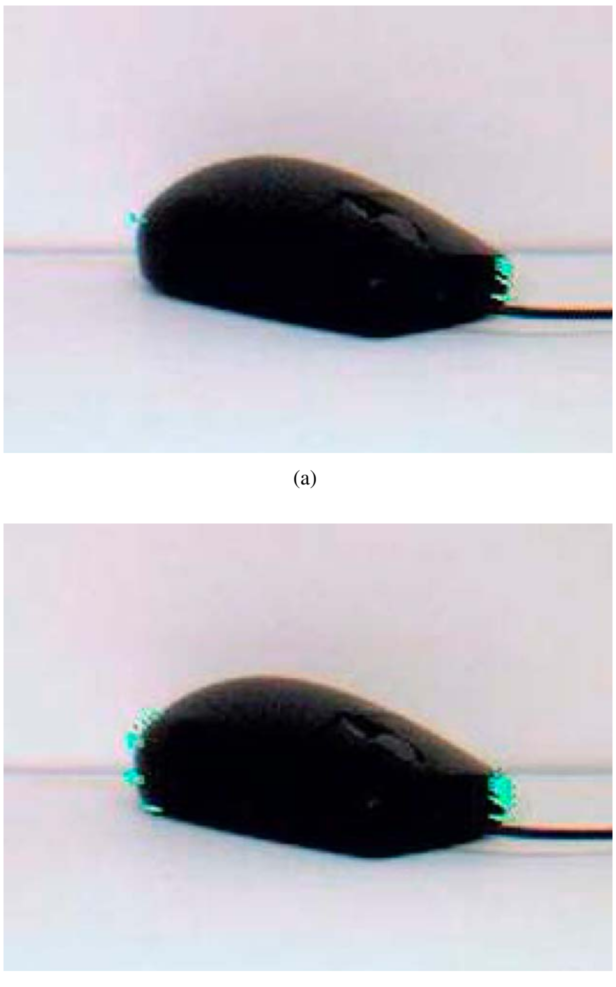

(a)

(b)

Figure 11. (a) No feature is selected by fixed threshold (FS-FIX) under dark lighting. (b) 156 features are selected by automatic threshold adjustment (FS-AUTO) under dark lighting.

selected features. Therefore, the number of selected features on a round object, such as a computer mouse shown in Figure 12(a), is not large enough, while this number on a complex object with many sharp corners is too large. In fact, FS-FIX chooses 42 features in Figure 12(a), which is far less than our target, 150. Similarly, it selects 572 features in Figure 13(a), which is almost 4 times more the desired number of features.

Figures 12(b) and 13(b) illustrate the same objects shown in Figures 12(a) and 13(a), however these objects are processed by FS-AUTO. The object in Figure 12(b) is round and does not have enough sharp corners, however, FS-AUTO successfully decreases the threshold value until it selects 154 features with a new threshold value of 300. Extra features

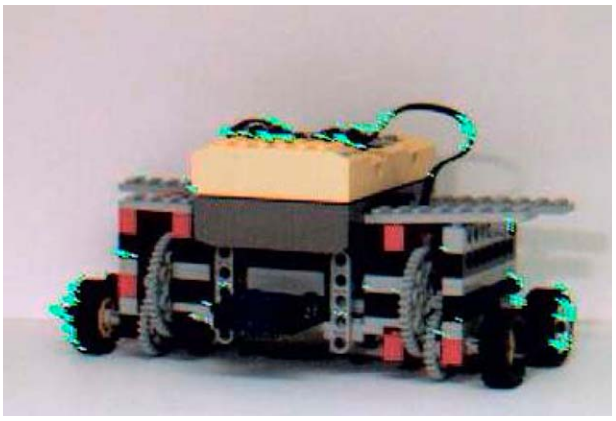

(a)

(b)

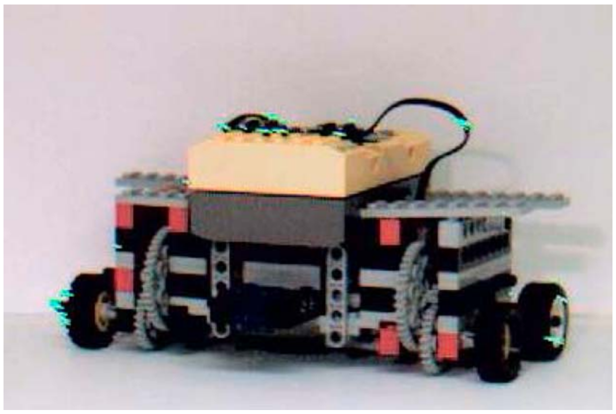
Figure 12. (a) 42 features are selected on a simple object with FS-FIX. (b) FS-AUTO selects 152 features on the object shown in Figure 10(a).

are observed at the left end of the object. Feature tracking algorithm can utilize this additional information for better tracking. The object in Figure 13(b) is a toy car that has many colorful parts and sharp edges, which are potentially good candidates for features. As presented earlier, FS-FIX uses a fixed threshold value for selecting features and it selects 572 features. Unnecessarily many features are observed around the wheel and wire part of the object in Figure 13(a). FS-AUTO adjusts the threshold value to select fewer features. It selects 152 features with a new threshold value of 912 (Figure 13(b)).

As discussed above, FS-AUTO is able to select proper number of features for any type or number of objects. It certainly is a better solution than FS-FIX, which works only for limited



(a)

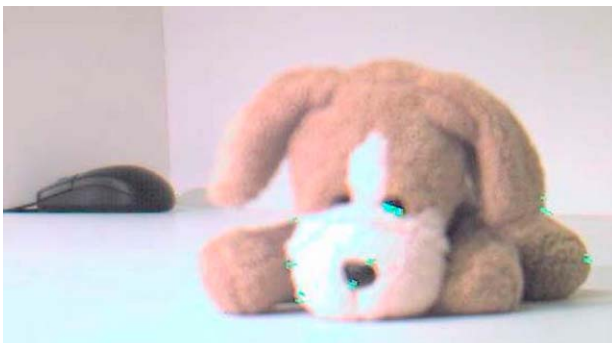


(b)

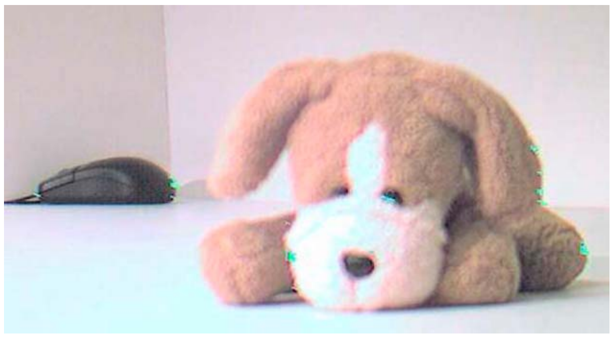
Figure 13. (a) 572 features are selected on an object with sharp edges, using FS-FIX. (b) FS-AUTO selects 152 features on the object shown in Figure 11(a).

type or number of objects, but it cannot solely handle all possible cases. One example is where multiple objects are present in a single scene. Therefore, the camera lens can be focused on only one of them. Under this situation, most of the features will be placed on one well-focused object and the rest of the objects will not be tracked.

Figure 14(a) demonstrates such a situation where the puppy doll that is close to the camera is better focused than the mouse located farther from the camera. FS-AUTO cannot select any features on the mouse. This is generally a hard problem to solve. However, by employing image restoration, the problem is alleviated to some degree. In Figure 14(b), FS-AUTO selects features on the same scene as Figure 14(a), however the image is first restored using the implemented image restoration algorithm. Restoration enhances the clarity of the edges and corners of both objects. After applying the image restoration algorithm, features are



(a)



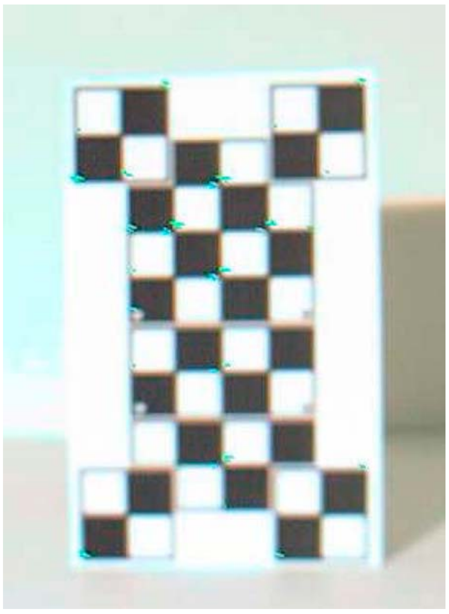
(b)

Figure 14. (a) FS-AUTO selects all of features on one of the objects (puppy in this example). (b) Applying image restoration before feature selection enhances the image quality by sharpening the edges and distributes the selected features on both objects.

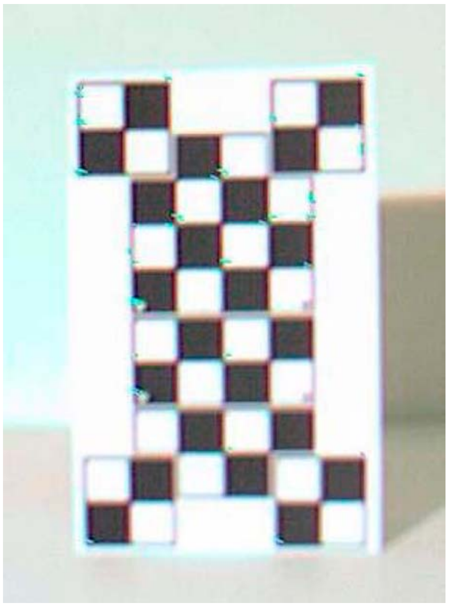
selected on the mouse as well as the puppy. Moreover, the number of features is balanced on the two objects. Note that the choice of enabling or disabling the image restoration algorithm is made on the fly and the system dynamically adapts itself to environment changes.

Figure 15(a) and (b) clearly demonstrate the effect of image restoration on feature selection results. In Figure 15(a), the lens is not well focused on the object. Although FS-AUTO can adjust its threshold to select the required number of features, features do not show satisfactory quality. The low threshold value used for selecting the features highlights this fact. Figure 15(b) shows the result of the same algorithm after dynamically enabling the image restoration before selecting the features in the image. Image restoration enhances the image quality by sharpening the edges. Therefore, the threshold value for selecting the same number of features on the restored image is larger. Hence, the features' quality has been enhanced and features with larger intensity difference compared to their adjacent pixels have been selected.

Note that sharp and clear images do not need to be restored before being passed to feature selection algorithm. Failure to do so might degrade the image quality by adding



(a)



(b)

Figure 15. (a) Selected features using automatically threshold adjustment (FS-AUTO) without image restoration. (b) Image restoration sharpens the edges and corners. Therefore, FS-AUTO selects better (selected with larger threshold) features.

Table 2. Feature selection threshold value and feature count for figures presented in experimental results section

Figure number	Threshold	Feature count
9(a)	512 (Fixed)	152
9(b)	465	148
10(a)	512 (Fixed)	1150
10(b)	1552	150
11(a)	512 (Fixed)	0
11(b)	160	156
12(a)	512 (Fixed)	42
12(b)	300	154
13(a)	512 (Fixed)	572
13(b)	912	152
14(a)	290	164
14(b)	664	162
15(a)	1279	146
15(b)	2083	148

noise to the image and can create fake features in the image. Therefore, the system should be reconfigured to enable or disable image restoration based on the requirements. In our system, we can dynamically enable or disable this module before selecting the features.

Table 2 summarizes the number of selected features and the utilized threshold value for selecting those features for images presented in this section. The enhanced performance of FS-AUTO compared to FS-FIXED in terms of number of selected features is evident. Furthermore, the effect of image restoration on the threshold value used in FS-AUTO can be observed. Note that applying image restoration on the blurry image shown in Figure 15(a) sharpens its edges and corners (see Figure 15(b)) and increases the required threshold in FS-AUTO. This in turn corresponds to features that are easier to track in the feature tracking stage. This has been highlighted in the last two rows of Table 2.

6. Conclusions and future directions

In this paper, we presented the idea of dynamic system reconfiguration in order to be able to adapt to the external events. A collaborative tracking system has been built and presented as the experimental framework for verifying the idea. Experimental results show that the idea is effective in practice and the system can function in a wide range of working conditions.

Particularly, we have implemented automatic adjustment of threshold value in feature selection algorithm, and dynamic enabling of image restoration for enhancing the image quality. These techniques have been integrated into our system framework. It has been shown that our approach is effective for dynamically adapting to various lighting and lens focus conditions in practice.

Future works include the integration of tracking phase of the KLT feature-tracking method into our current system, enhancing the collaboration schemes and applying the system reconfiguration idea to other applications or application domains.

Notes

1. The camera was originally shipped with an embedded Xilinx Virtex200E FPGA whose CLBs are used for proprietary logic implementation. Therefore, we had to replace the FPGA with a larger device in order to be able to implement the required image-processing algorithms.
2. Our implementation is based on [15].
3. The images are copied from http://astronomy.swin.edu.au/∼pbourke/analysis/imagefilter/

References

1. H. C. Andrews and B. R. Hunt. *Digital Image Restoration*. Prentice Hall, 1977.
2. P. Athanas and L. Abbott. Addressing the computational requirements of image processing with a custom computing machine: an overview. In *Proceedings of the 2nd Workshop on Reconfigurable Architectures*, April 1995, Santa Barbara, CA.
3. A. Benedetti and P. Perona. Real-time 2-D feature detection on a reconfigurable computer. *IEEE Conference on Computer Vision and Pattern Recognition*, June 1998, Santa Barbara, CA.
4. J. Biemond, J. Rieske, and J. J. Gerbrands. A fast kalman filter for images degraded by both blur and noise. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1983.
5. G. Bilardi and M. Sarrafzadeh. Optimal VLSI circuits for discrete fourier transform. *Advances in Computing Research*, 4:87–101, 1987.
6. F. Cuzzolin, A. Bissacco, R. Frezza, and S. Soatto. Towards unsupervised detection of actions in clutter. *Proc. of the Asilomar Conference on Signals, Systems and Computers*, 2002.
7. D. Estrin et al. Embedded, everywhere: a research agenda for networked systems of embedded computers. Committee on Networked Systems of Embedded Computers, Computer Science and Telecommunications Board, National Research Council, Washington, DC, 2001.
8. X. Feng and P. Perona. Real time motion detection system and scene segmentation. *CDS TR CDS98-004*, Caltech, 1998.
9. B. Fortner, T. E. Meyer, and T. Meyer. *Number by Colors: A Guide to Using Color to Understand Technical Data*. Springer Verlag, 1997.
10. S. Ghiasi, H. J. Moon, and M. Sarrafzadeh. Collaborative and reconfigurable object tracking. *Engineering of Reconfigurable Systems and Algorithms*, 2003.
11. IQinVision Online Documentations, IQinVision Inc., http://www.iqinvision.com.
12. D. J. Li, L. Jiang, T. Isshiki, and H. Kunieda. New VLSI array processor design for image window operations. *IEEE Transactions on Circuits and Systems*, 46(5):635–640, 1999.
13. A. K. Katsaggelos. Iterative image restoration algorithms. *Optical Eng.*, 28:735–748, 1989.
14. B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. *International Joint Conference on Artificial Intelligence*, 674–679, 1981.
15. M. Maire, Design and implementation of a realtime visual feature tracking system on a programmable video camera. Technical Report, California Institute of Technology, 2002.
16. K. Melhorn and F. Preparata. Area-time optimal vlsi integer multiplier with minimum computation time. *Information and Control*, 58:137–156, 1983.
17. ModelSim product manual, Model Technology Inc., http://www.model.com.
18. S. Ogrenci Memik, A. K. Katsaggelos, and M. Sarrafzadeh. FPGA implementation and analysis of an iterative image restoration algorithm. *IEEE Transactions on Computers*, 52(3), 2003.
19. J. C. Russ. the image processing handbook. CRC Press, 1999.
20. M. Sarrafzadeh, A. K. Katsaggelos, and S. P. Kumar. In Parallel architectures for iterative image restoration. Kluwer Academic, M. Bayoumi editor, 1991.
21. J. Shi and C. Tomasi. Good features to track. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 593–600, 1994.
22. Synplify Pro product manual, Synplicity Inc., http://www.sinplicity.com.
23. D. Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):59–66, 2000.

24. C. Tomasi and T. Kanade. Detection and tracking of point features. *Carnegie Mellon University Technical Report CMU-CS-91-132*, April 1991.
25. H. J. Trussel and B. R. Hunt. Improved methods of maximum a posteriori restoration. *IEEE Transactions On Computers*, 28, 1979.
26. M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.
27. Xilinx Online Documentations, Xilinx Inc., http://www.xilinx.com.