Profiling Accuracy-Latency Characteristics of Collaborative Object Tracking Applications

Soheil Ghiasi, Karlene Nguyen, Majid Sarrafzadeh
Computer Science Department
University of California, Los Angeles
{soheil, karlene, majid}@cs.ucla.edu

Abstract

Many parallel and collaborative signal processing systems utilize commercial off-the-shelf sensor nodes with constrained embedded processors. Applications running on such processors, e.g. object tracking, often demand real time performance and hence, another design metric such as accuracy has to be compromised to meet the performance constraint. Therefore, exact accuracy-latency characteristics of the application are required in order to implement it in a real time and sufficiently accurate fashion. This paper presents profiling techniques that are applicable to tracking applications including those implemented on a parallel system. The approach has been applied to a tracking application implemented on a collaborative system that has been built in our lab. Extensive profiling has been performed to study embedded vs. centralized and accuracy vs. latency tradeoffs. Experimental results verify the effectiveness of our profiling scheme, and support the fact that different computing schemes are appropriate for different accuracy and performance requirements. Experiments show that an appropriate choice of algorithms and computing schemes of our system, leads to 12 times speedup in feature tracking latency compared to its original version with a reasonable reduction in tracking accuracy.

1. Introduction

Today's advances in technology, has enabled the integration of processing resources, memory blocks and sophisticated I/O units into various electronic devices including commercial off-the-shelf (COTS) data acquisition units [1, 2]. Such 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 [3].
Many COTS sensor devices have been developed with serious cost, power, and size constraints. Therefore, the computation resources embedded in them are not as strong as systems that are designed without such limitations. In addition, these products have to be treated as black boxes in that designers cannot modify or upgrade the embedded functional units. Therefore, systems that are developed using such products often suffer from constrained and low performance processors.
On the other hand, many applications that utilize constrained sensor nodes demand real time performance. Examples include multimedia and tracking applications that use cameras with low power embedded processors. Such applications cannot tolerate slow processing and long latencies since they have to process the real time stream of incoming data. Therefore, designers have to compromise one of the design metrics, such as accuracy to obtain the desired performance for this class of applications.

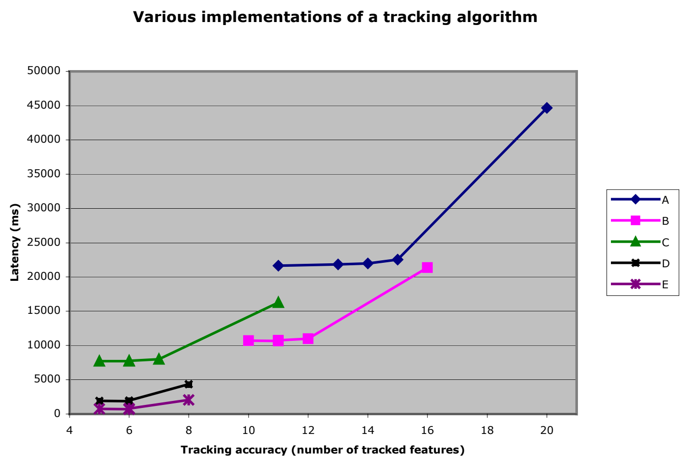


Figure 1. Accuracy (or other design metrics) can be compromised for performance in constrained systems.

Figure 1 demonstrates the idea of compromising accuracy to achieve application speedup. The chart shows accuracy-latency behavior of different 5 different implementations of a tracking algorithm. Implementations with lower accuracy execute faster and vice versa. Various accuracy-latency points provide the designer with a range of options to choose from based on design requirements. Specifically, design specifications often set an upper bound for the worst acceptable latency of the application and the objective is to find the most accurate implementation that satisfies the timing constraint. Note that accuracy is mentioned as a sample design metric used for tracking applications. The very same idea is applicable to all other designs using other conventional metrics such as power consumption, chip area, and cost.
Various design and optimization methodologies have been proposed to address the problem of selecting a proper implementation under timing constraints. Particularly, different *budgeting* techniques have been developed that essentially exploit the timing slack of application building blocks to improve its accuracy (or any other design metric) [14, 15, 16, 17]. These techniques assume that complete accuracy-latency information of application building blocks is available.

While the aforementioned assumption seems reasonable for many practical applications, elaborate profiling experiments have to be carried out in order to obtain the accuracy-delay information. Profiling techniques implement a given application in a number of different ways and measure its accuracy-latency relation (or another metric's relation with latency) for running on different system resources. Each implementation running on each particular system resource corresponds to a point in the accuracy-latency plane. Budgeting techniques are then applied to a number of points in the plane to select the proper set of implementations that optimizes some particular objective.

This paper focuses on the software profiling methodologies appropriate for parallel and collaborative signal processing applications. We present generic methodologies that are applicable to all KLT-based [5, 6, 7] tracking applications. The results have been experimented on a tracking system that has been implemented as part of this work.

We proceed to discuss the framework of the implemented system in the next section. Section 3 presents the implemented target tracking application. The trade offs involved in embedded vs. traditional computing schemes are explained in section 4, followed by the trade offs between accuracy and latency of the implemented algorithms. Section 5 explains the image processing algorithms used in the system and the steps taken to adapt them to our framework and section 6 presents the experimental results.

2. Computing Trade offs Involved in a Generic Collaborative Tracking System

In this section, we present various trade offs involved in running a tracking application in a parallel and collaborative environment. Particularly we will discuss two possible computing schemes, namely, locally embedded at the sensing point and remote processor-based computing. Embedded execution of the computations is an essential task for performing the application in parallel and on distributed sensor nodes (cameras). Moreover, we discuss the trade offs involved in adapting the tracking algorithms ported to such a constrained platform. The algorithms have to be simplified in order to make them appropriate for execution on the constrained resources embedded in the sensors nodes, which in turn leads to parallel execution of the application.

2.1 Embedded vs. Processor-based Computing

Real time motion analysis is one of the most critical requirements of the tracking application. High performance is often difficult to achieve because many image-processing algorithms are computationally intensive and necessitate long latency calculations. Hardware realization of the algorithms is not often possible, since the embedded processors only allow software processing. However, it is still essential to attempt to maximize system performance.

One way of improving system performance is to collocate the input data acquisition and its processing on the vision sensors (cameras). This will lead to parallel execution of the computation on all of the vision sensors and might improve system performance. In the past, vision sensors were used primarily for grabbing frames and sending the raw or compressed image data to a graphics processing board located on a central processor. Such a computing scheme, limits the scalability of the system, adds additional data transfer latency and utilizes a significant amount of bandwidth by transferring entire images. Therefore, a networked camera with an embedded image sensor and proper computational resources can minimize network overhead by processing the data locally. Therefore, it needs to send a limited amount of data to other nodes to collaborate, which in turn enhances system scalability.

Unfortunately, one of the major drawbacks of using such camera with an embedded processor is that the processor tends to be slower than a general-purpose processor of a PC. Therefore, it is unclear whether computations should occur on or off the camera since there are drawbacks to each. The proper computing scheme depends on the particular system parameters and specification. Profiling techniques assist in determining such parameters and can help in determining the appropriate computing scheme for different objectives.

2.2 Accuracy-Performance Trade off

The processing power and memory modules embedded in many COTS products, such as cameras with embedded processors, are quite constrained compared to desktop PCs. Hence, it is important to consider this obstacle when attempting to perform complex computations in the cameras. In order to achieve real-time computations, complex algorithms must be simplified. While most of the required image-processing algorithms are computationally intensive and many prior research have tried to implement them on hardware for realtime performance [8, 9]; simplification of such algorithms can also lead to enhanced performance results (see Figure 1 for an example). As a result of simplifying algorithms, it is inevitable to sacrifice fine-grained accuracy for computational speed. A widely accepted methodology to improve accuracy is to employ effective budgeting schemes to select the proper simplification that maximizes accuracy while meeting the timing constraint [14, 15, 16, 17].

Common intruder detection and object tracking applications usually rely on a tracking scheme developed by Tomase and Kanade [6]. Their technique, usually referred to as KLT tracking scheme [5, 6, 7], locks onto particular points in the image, called *features*, and tries to track them in subsequent images of the same scene. KLT tracking uses two computationally intensive image processing algorithms for selection and tracking of feature points [5, 6, 7]. Since both of these algorithms are quite complex, the processing time is large and can cause performance bottlenecks. Before discussing the generics simplifications that can be made to these algorithms to increase their computational speed, we will briefly discuss the KLT feature selection and tracking algorithms and will explain how each algorithm works. Then, we will discuss the specific simplifications that can be made to feature selection and tracking to adapt them to a constrained parallel system. The simplifications are general and can be applied to any KLT-based tracking system.

3. KLT Tracking Algorithms

In order to construct a real-time system, it is essential to aim for high-performance by minimizing computational latency. Unfortunately, tracking objects is quite complex and requires a vast amount of computation. However, various techniques have been proposed in an effort to reduce computation. One such technique verifies and updates information about the positions of selected small windows, or features, of an object. By only "tracking" small windows in an image, an object can be tracked and the amount of computation can be decreased substantially. Selection of these windows is generally determined using an a priori basis for what is deemed an "interesting" feature. According to the aperture problem, not all points in an image are useful for tracking and so it is necessary to carefully select the points where motion information can be extracted. The basis for the selection of a window could be tracking of corners, windows with high spatial frequency content, or regions with particular brightness patterns. Tomasi and Kanade [6] derived a criterion for feature selection based on large contrasts of intensity. The KLT feature selection and tracking algorithms, commonly accepted within the vision community, was chosen for this application in order to maximize performance by minimizing computation while also attaining high-quality tracking results [5, 6, 7].

In the following section, two image-processing algorithms that enable a system to track a moving object will be explained. These algorithms include feature selection and feature tracking.

3.1 Feature Selection

The feature selection algorithm consists of carefully choosing the points in the image, which can be easily tracked throughout a series of images. In order to select good features to track, the following steps are taken:

1. Smooth the entire image.
2. Calculate g_x and g_y, the intensity gradients in the x and y directions by computing the Gaussian and Gaussian derivative kernel as well as convolving these kernels in the horizontal and vertical directions.
3. For each pixel:
a) Sum the gradients in the surrounding window in order to compute the Z matrix, where

$$Z = \int\int_W \begin{bmatrix} g_x^2 & g_x g_y \\ g_x g_y & g_y^2 \end{bmatrix} d\mathbf{x}$$

b) 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.
c) Store the trackability value associated with the x and y coordinates in a list, called `FeatureList`.
4. Sort this list in ascending order.
5. While ascending the list, enforce a minimum distance between features and add all acceptable features to another list. In this manner, features with the largest trackability values that have a minimum distance will be inserted in the second list until no more features can be added. Figure 2 demonstrates the output of the feature selection on a selected region of sample images.

3.2 Feature Tracking

After feature selection, features must be tracked to the next successive image to determine the displacement of the feature that minimizes dissimilarity. Dissimilarity caused by motion between two successive images can be represented by the following equation:

$$J(\mathbf{x}+\mathbf{d}) = I(\mathbf{x}) \quad , \quad \mathbf{x} \in W$$

where I(**x**) denotes the intensity of point **x** = [x y] in the image for feature window **W** and J(**x**+**d**) denotes the intensity of another successive frame with constant displacement **d** = [dx dy]. Consequently, to find displacement d that minimizes dissimilarity **Є**:

$$\epsilon = \int\int_W [J(\mathbf{x}+\mathbf{d}) - I(\mathbf{x})]^2 d\mathbf{x}$$

Using the iterative gradient descent method, we can minimize dissimilarity **d** by solving the following linear system:

$$Z\mathbf{d} = \mathbf{e}$$

where:

$$Z = \int\int_W \begin{bmatrix} g_x^2 & g_x g_y \\ g_x g_y & g_y^2 \end{bmatrix} d\mathbf{x} \quad e = \int\int_W [I(\mathbf{x}) - J(\mathbf{x})] \begin{bmatrix} g_x \\ g_y \end{bmatrix} d\mathbf{x}$$

and where $\boldsymbol{g_x}$ and $\boldsymbol{g_y}$ are intensity gradients in the *x* and *y* directions. The following steps are used in the KLT feature tracker:

1. Process first image by smoothing, computing resolution pyramid, and computing *x* and *y* gradient for each pyramid
2. Process second image by smoothing, computing resolution pyramid, and computing *x* and *y* gradients for each pyramid
3. For each feature in the `FeatureList` that has a positive trackability value (i.e., only track features that are not lost):
   a. Transform feature location to coarsest resolution.
   b. Beginning with coarsest resolution
      i. Compute gradient sum and intensity difference windows.
      ii. Use these windows to construct a 2x2 gradient matrix **Z** and 2x1 error vector **e.**

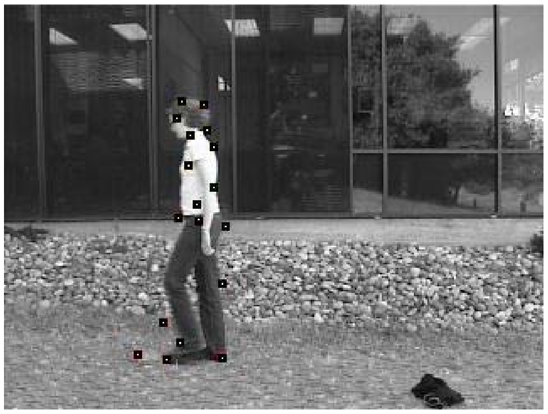
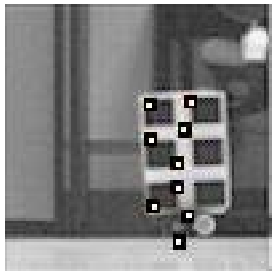



Figure 2. Sample outputs of feature selection algorithm executed on the camera. Features are denoted by black squares with white centers.

iii. Use matrices to solve equation for new displacement.
iv. Iteratively, update the window position.

c. Ensure new window is not out of bounds; residue is not too large; etc.
d. Record new feature window coordinates.

Figure 3 demonstrates feature tracking on a series of images. Throughout the series of images, the features are updated and tracked as the person walks across the scene.
In short, the KLT feature tracker uses an iterative gradient descent method to minimize the displacement of a feature caused by motion. The feature tracker can only work successfully if there are small amount of motion between two successive frames.

4. Profiling by Applying Various Optimizations to KLT Tracking Algorithms

We have performed a thorough analysis of the computational bottlenecks in the standard KLT feature selection and feature tracking algorithms. The analysis has been done using standard academic software implementations. Several possible algorithmic modifications and simplifications are proposed. These simplifications successfully compromise computation latency with tracking quality by maintaining algorithm correctness. Note that careless modification of the algorithm might not preserve its functional correctness. These simplifications include:

1. Elimination of pyramidal structure. This structure is used for performing computations on different down-sampled versions of the image.
2. Elimination of smoothing. Smoothing is required for high quality tracking and can be neglected is performance is an issue.
3. Elimination of the original complex gradient calculation algorithm and introduction of a simplified gradient algorithm. This has to be done with complete understanding of the algorithm, since it might not preserve its correctness.
4. Elimination of whole image gradient calculation and introduction of small window gradient calculation.

In the rest of this section, we explain various simplifications that we made to the standard KLT tracking algorithms. The simplifications produced 5 different (including the unsimplified version, which we call it version ‘a’) implementations of the same algorithm, which correspond to different points in the delay-accuracy plane and can assist in profiling a given parallel and collaborative tracking system.
Many of the image processing algorithms perform computations on a number of down sampled versions of the original image to improve their quality. A pyramidal structure is used for implementing the down sampled versions of an image, where the original image is assumed to be at the lower level of the pyramid. Each upper level of the pyramid down samples its immediately preceding level by a constant factor. While the pyramidal structure allows tracking to occur on several resolutions of an image, it also adds to the complexity of the computation by requiring additional memory allocation and computational overhead (particularly of gradients). Hence, the

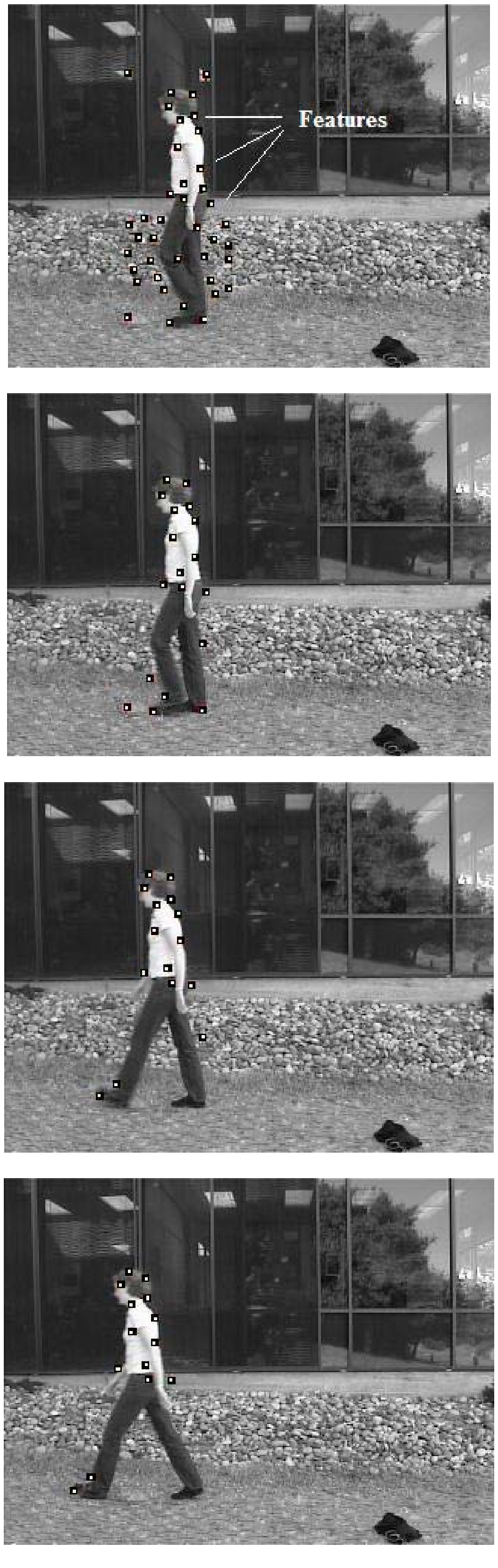


Figure 3. Output of feature tracking algorithm on a consecutive series of images. Features are denoted by black squares with white dots in center.

pyramidal structure can be eliminated for the feature selection and feature tracking algorithms to improve their runtime. Hence, version ‘b’ of the feature selection and tracking only occur on the original image, not on coarse-grained images.

Another additional computation is created by smoothing an image. Smoothing enhances the quality of the image before running the actual computation. Therefore, it can be safely removed if the original image has a reasonable quality. Smoothing code was removed to produce version ‘c’, which further simplifies the feature selection and feature tracking algorithms.

Moreover, another accuracy-delay trade off can be achieved by taking into account the gradient calculation mechanism. The computation used for the calculation of the gradient in the KLT is quite complex. In order to simplify the computation for calculating the gradient, we replaced it with convolution of a simple kernel. Using this method, the gradient is calculated by convolving I(**x**) with the following kernels:

$$k_x = \frac{1}{2}\begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \qquad k_y = \frac{1}{2}\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

While this modification degrades the quality of the feature selection and tracking algorithms, it greatly improves their latency. Furthermore, the quality of the simplified tracking is reasonable for usual working conditions. This version of the algorithm is called version ‘d’.

There exists another computational bottleneck in the computation of the gradient of the entire image within the feature tracking algorithm. An excessive amount of computation is required for calculating the x and y gradients of the entire pixels of an image. However, only the gradient computations around small feature windows within the image are necessary. Hence, in version ‘e’ of our implementation, the feature tracking algorithm was further modified to calculate the gradients around small feature windows as needed instead of the gradient of the entire image.

5. Experiments

In this section, we present a collaborative object tracking system that is built as the experimental platform of this work. The aforementioned profiling techniques have been applied to a KLT tracking application implemented on this platform. Different versions of the algorithms have been tested on our platform to profile their delay-accuracy characteristics.

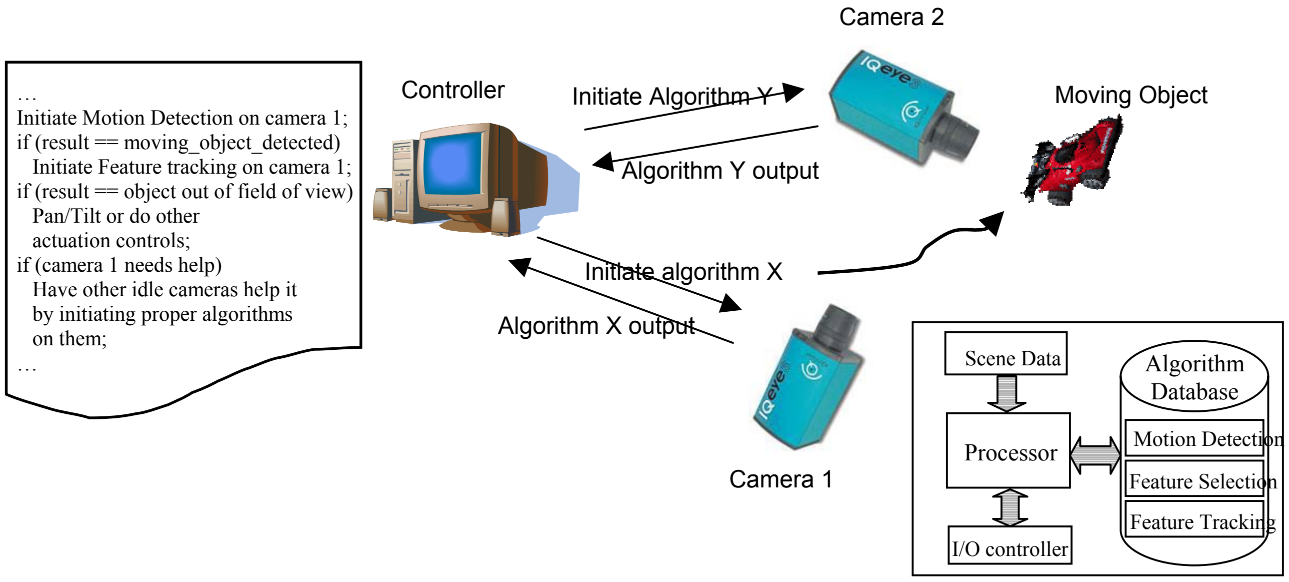
5.1 Experimental Platform Framework

The framework for our system is comprised of several components including: two IQeye3 cameras provided by IQinVision [4], pan-tilt units to enable the actuation of the cameras, a main controller residing on a PC, and a network for communication (Figure 4).

There is a 250 MIPS PowerPC CPU, 4 MB of Flash RAM and 16 MB of SDRAM embedded in each camera. Each IQeye3 camera gives full access to raw real-time image data streams and the 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 D serial port. In our system, each IQeye3 camera is mounted on a pan-tilt unit and is able to directly communicate with the pan-tilt unit via a 9-pin D serial port. Each pan-tilt actuation unit can be sent simple messaging commands to specify the pan angle, pan speed, pan acceleration, tilt angle, tilt speed, and tilt acceleration.

The main supervisory controller resides on a PC 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



```
…
Initiate Motion Detection on camera 1;
if (result == moving_object_detected)
  Initiate Feature 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;
…
```

Figure 4. An overview of the tracking system architecture: Each camera has a set of the required algorithms available. The controller communicates with the cameras via an implemented message passing scheme. It can initiate the proper algorithm on each camera, and organize the collaboration among cameras.

appropriate actions that should be taken by each camera (e.g., swapping of a different algorithm). This is performed by sending a message to the designated camera.

5.2 Experimental Platform 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 4). 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 speed of the object, different timing constraints exist for performing computations.

When the entire system initializes, cameras establish a connection with the main supervisory controller on the PC. Camera 1 assumes control initially and continuously waits for a moving object to enter its field of view. This is accomplished using the motion detection algorithm, a fast and simple high-level computation that can be used for future low-level image-processing algorithms. The motion detection algorithm returns the amount of motion detected (in pixels) given a particular threshold as well as the rectangular coordinates of the region where the motion is located. These rectangular coordinates can be utilized as input to other algorithms to avoid unnecessary computations on still sections of the field of view and hence, allow for additional computational speedup.

In our system, the rectangular coordinates are utilized by the feature selection algorithm on the current streamed image in order to determine features in the rectangular region where a moving object is detected. The feature selection algorithm returns a feature list, which contains x-y coordinate pairs representing small windows called features that are deemed "interesting" to track. Each feature also has an associated integer value, which represents the trackability of the feature.

On the next consecutive streamed images, the feature tracking algorithm is run in order to track the same features generated by the feature selection algorithm. Those features that are successfully tracked modify the feature list with their new x-y coordinate pairs. Since our application is targeted at tracking moving objects, we eliminate all features that have not moved substantially beyond a given threshold. Feature tracking can extract directional and speed information about a moving object, therefore, it is possible to run different versions of feature tracking depending on the speed of the moving object. The ability to execute different versions of feature tracking enables the algorithm that maximizes the tracking quality to be used.

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 and the current version of the feature selection/tracking algorithm. The main controller then decides which camera should gain control and sends the camera a message indicating where the object is and which version of the feature selection/tracking algorithm to use. 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 4 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 has been demonstrated. In such a manner, the moving object is continuously and vigilantly tracked using multiple cameras with the most efficient feature selection and tracking algorithms. The use of the right implementation of the feature selection/tracking algorithms leads to the highest quality tracking results. Note that by use of the "hands off" approach, the object will be continuously tracked as long as the object is within the field of view of a camera.

5.3 Experimental Results

In this section, we present the result of experiments that have been carried out using the system depicted in figure 4. In order to determine the extent of performance improvements after a series of simplifications to the feature selection and feature tracking code, we determined the runtimes of the simplified versions of each algorithm on a static series of images.

Figure 5 presents the results of the aforementioned experiment, i.e. the effect of the simplifications on latency of each algorithm. Runtimes are reported for running the algorithm on the controller (a PC). While we expect the runtimes to scale almost linearly when the algorithms are ported to the camera, we have performed the experiments on the camera as well. The results of running the algorithms on the camera are presented later in this section. Various versions of cumulative simplifications include:

a) original implementation, b) elimination of the pyramidal structure, c) elimination of smoothing of the image, d) substitution of a simple gradient calculation, e) elimination of whole gradient calculation and introduction of small window gradient calculation.

The cumulative simplifications made to the feature tracking code reduced the runtime 3.6 times, namely from 365 ms to 101 ms. As expected, the simplifications did not reduce the runtimes significantly for the feature selection algorithm. However, this does not seem to be a major problem, because feature selection is executed only once for selecting features, while feature tracking is iteratively executed on each frame to track the moving objects.

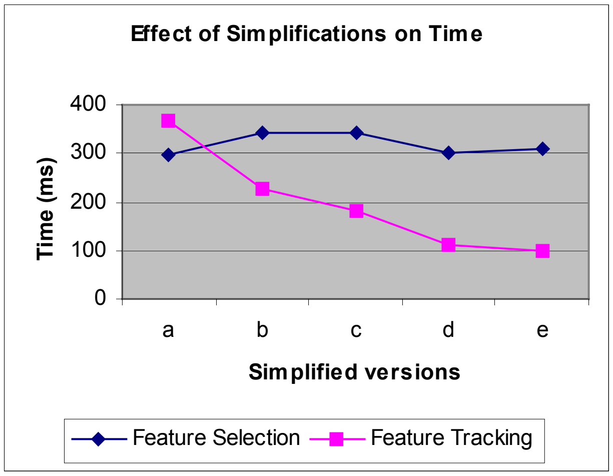


Figure 5. Effect of algorithm simplifications on runtimes on a PC.

It is important to note that there was not a significant decrease in algorithm latency between simplified version d and e. A more noticeable decline in runtime would be noticeable with fewer features to track (e.g., less than 100 features), because as the number of "features to track" increases, the small window gradient calculation covers more pixels of the entire image (refer to the 4th simplification for feature tracking algorithm described in section 4). Therefore, less improvement in runtime of version e over d would be expected with increasing the number of features.

A significant improvement in runtime of feature tracking resulted from the simplifications. While it is difficult to quantify how much effect these optimizations had on the quality of feature tracking, we can quantify how these optimizations affected the number of features that are tracked. Number of tracked features correlates with the quality of the tracking algorithms, because a less accurate tracking algorithm is more likely to '*lose*' a feature during the tracking process.

Figure 6 demonstrates the effect of algorithm simplifications on their quality. This graph is made by selection of 100 features on a static image and running different versions of feature tracking algorithm to track those features in the next frame. Each algorithm loses some of the original features during the tracking process. The number of successfully tracked features is reported as a measure of accuracy.

As it can be inferred from Figure 6, simplifications minimally compromised the number of features that were tracked after the feature selection algorithm selected 100 features to track. In particular, the original version was able to track 83 features while the most simplified version (e) tracked 64 features. Although the number of features tracked decreased slightly with the simplified versions, the object can continue to be tracked precise enough, with the remaining number of features.

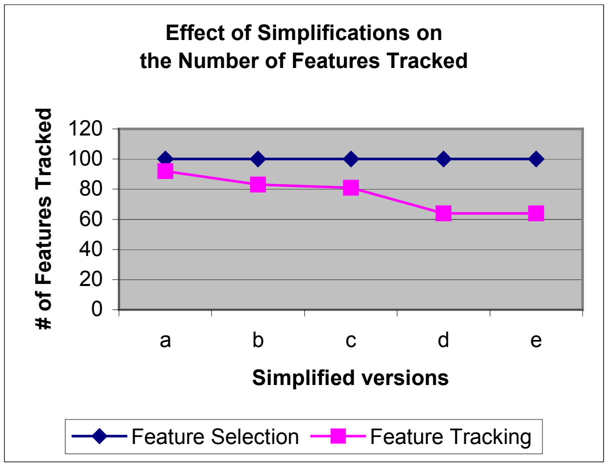


Figure 6. The effect of algorithm simplifications on the quality of tracking.

While several optimizations appeared quite promising on the PC, we were interested in how effective the optimizations would be when these algorithms were ported to the IQeye3 camera platform. Figure 7 shows the results for the feature selection algorithm. The original KLT code performs significantly better when the image is sent over the network and processed on the PC compared to simply processing the image on the embedded processor on the camera.

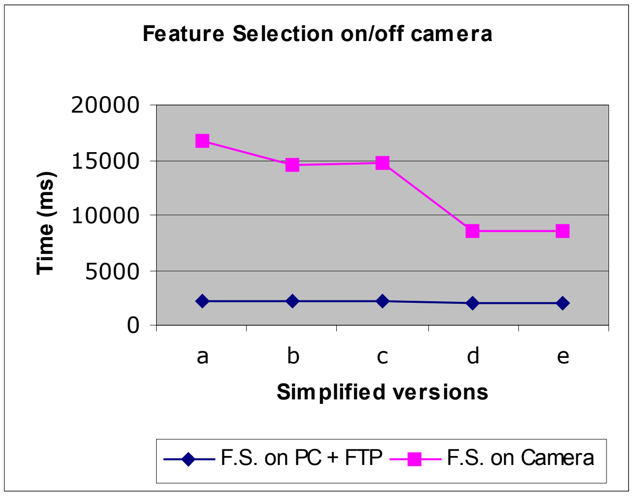


Figure 7. Runtimes of different versions of feature selection in two embedded and centralized modes of operation.

After several simplifications to the KLT code, the feature selection algorithm code runs locally on the camera in less time than the original implementation. It is evident that the simplifications have made the feature selection to run more than 2 times faster on the camera. However, it remains more efficient to FTP the image over the network and run the feature selection algorithm on the main controller. These results are not surprising since most of the simplifications affected the feature tracking algorithm, not the feature selection algorithm.

An interesting point to notice is that the latency of running computations in centralized scheme (FTP + execution on the controller) is dominated by the FTP latency. Therefore, different versions of the algorithm exhibit similar latencies compared to the large latency of executing the algorithms on the cameras. The controller has powerful computational resources (we used a PC with Intel Pentium III running at 750 MHz and 512 MB of main memory in our experiments), therefore, the latency of running the computations on controller is almost equal to the latency of sending images back and forth.

Since more simplifications had an impact on the feature tracking algorithm, it is interesting to note how the optimizations affect feature tracking on images that were processed locally on the camera. Figure 8 shows the latency of different versions of feature tracking algorithm in two embedded and centralized modes. The optimizations made on the feature tracking algorithm allow it to perform better on the local embedded processor of the IQeye3 camera. Though the camera processes computations much slower compared to the PC, the camera does not have to incur the extra overhead of sending the image through the network.

Particularly, it is interesting to note that the original implementation of the KLT feature tracking algorithm takes about 22000 milliseconds on the camera to run. It takes about 2100 milliseconds to transmit an image to the controller and execute the very same implementation of the KLT feature tracking algorithm on it. The aforementioned simplifications reduce the algorithm latency on the camera about 12 times, i.e. from original 22000 ms to 735 ms, while the runtime on the controller improves only 5%.

The most simplified version runs 2.7 times faster in the embedded mode compared to the centralized scheme. Therefore, it is a good candidate for implementing the application in distributed fashion.

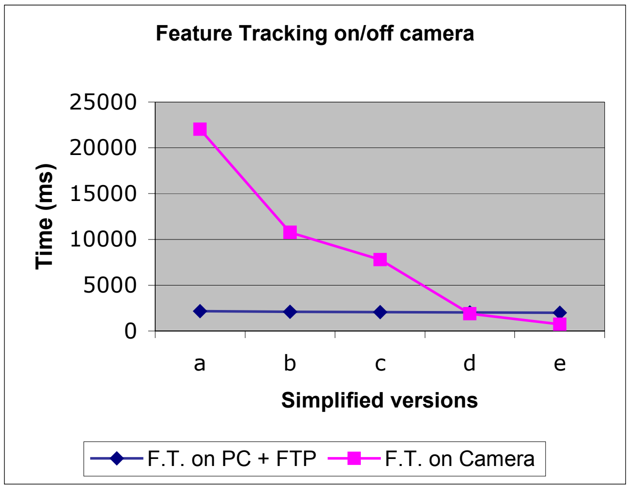


Figure 8. The Latency of embedded versus centralized computing schemes for feature tracking algorithm.

6. Conclusions and Future Directions

We presented an accuracy-delay profiling methodology that is applicable to tracking applications. General guidelines for compromising the tracking accuracy with computation latency have been proposed. Applying these techniques produces different versions of the tracking algorithm with different accuracy-latency behavior.

A collaborative signal processing system has been developed to experiment our approach. The system consists of multiple cameras with embedded processors and a control unit connected through the local area network. A target tracking application utilizing KLT feature selection and tracking algorithms has been implemented on this framework. Since these algorithms are computationally intensive, they do not show satisfactory performance when run on the constrained embedded processors of the camera. Therefore, various simplifications have been made to these algorithms to adapt them to our framework and allow adaptation of the application to parallel execution scheme. Optimizations improved algorithm runtimes up to 12 times with reasonable degradation in tracking accuracy.

Future works include porting the algorithm on reconfigurable hardware resources of the camera and applying the runtime hardware reconfiguration to various applications handled by our framework. Moreover, we would like to investigate the issue of profiling for generic classes of applications.

7. References

[1] D. Tennenhouse, "Proactive Computing," *Communications of the ACM*, May 2000, vol. 43, no. 5, pp. 59–66.

[2] M. Weiser, "The Computer for the 21st Century", *Scientific American*, Sept. 1991, vol. 265, no. 3, pp. 94–104.

[3] 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.

[4] IQinVision Online Documentations, IQinVision Inc., http://www.iqinvision.com.

[5] B. Lucas, T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision", *International Joint Conference on Artificial Intelligence*, pp. 674-679, 1981

[6] C. Tomasi, T. Kanade, "Detection and Tracking of Point Features", *Carnegie Mellon University Technical Report CMU-CS-91-132*, April 1991.

[7] J. Shi, C. Tomasi, "Good Features to Track", *IEEE Conference on Computer Vision and Pattern Recognition*, pages 593-600, 1994

[8] A. Benedetti, 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.

[9] 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.

[10] A. Bissacco, A. Chiuso, Y. Ma and S. Soatto, "Recognition of human gaits", *In Proc. of the IEEE Intl. Conf. on Computer Vision and Pattern Recognition*, vol II, pages 52-58, 2001.

[11] X. Feng, P. Perona, "Real Time Motion Detection System and Scene Segmentation", *CDS TR CDS98-004*, Caltech, 1998

[12] H. Jin, P. Favaro and S. Soatto, "Real-time Feature Tracking and Outlier Rejection with Changes in Illumination", *Proc. of the Intl. Conf. on Computer Vision*, July 2001.

[13] S. Smith and J. Brady, "Asset-2: Real-time Motion Segmentation and Shape Tracking", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 8, no. 17, pp. 814-820, 1995.

[14] E. Bozorgzadeh, S. Ghiasi, A. Takahashi, M. Sarrafzadeh, "Optimal Integer Delay Budgeting on Directed Acyclic Graphs", *Design Automation Conference (DAC)*, June 2003.

[15] S. Ghiasi, K. Nguyen, E. Bozorgzadeh, M. Sarrafzadeh, "On Computation and Resource Management in an FPGA-based Computing Environment", *A Poster in International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2003

[16] C. Chen, E. Bozorgzadeh, A. Srivastava, M. Sarrafzadeh. "Budget Management with Applications". *Algorithmica*, pages 261-275, July 2002.

[17] M.Sarrafzadeh, D.A. Knol and G.E. Tellez. "A Delay Budgeting Algorithm Ensuring Maximum Flexibility in Placement". *In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1332-1341, 1997.