# Design of Embedded Systems for Real-Time Vision

Vivek Haldar[1] Gokul Vardhan[1] Abhishek Saxena[2]

Subhashis Banerjee[1] M. Balakrishnan[1]

[1]Department of Computer Science and Engineering
[2]Mathematics Department
Indian Institute of Technology, New Delhi 110016, India
Email: {suban,mbala}@cse.iitd.ernet.in

## Abstract

In this paper we present a design methodology of real-time vision based embedded systems on PCs and other low-end platforms. We first develope vision algorithms on the Linux platform and port the application on to a real-time kerner with pluggable scheduling policies and real-time guarantees. The kernel has been ported to both X86 and Philips Trimedia platforms. The methodology has been validated by porting three vision applications on both thee platforms.

## 1 Introduction

In this paper we address the issues of design of embedded systems for real-time vision applications. Most real-time vision applications have a sensing-perception-action loop which involves i) video capture at frame rate (25 Hz) ii) vision processing which is mainly compute bound and iii) robot control. Such systems often have hard real-time deadlines to enable small latency reaction to real world events and ensure stability of control. Real time vision applications are typically developed in reasearch environments using high-end visual workstations which provide out-of-the-box solutions for frame capture and sophisticated tools for image processing, display, multithreading etc. The operating systems in such workstations are general purpose with large overheads and often do not provide any real-time guarantees. Consequently, a programmer has to mainly rely on the fast processors available in such systems to meet real-time requirements and can seldom enforce them in a systematic way.

However, the end use of most such vision applications lie in special purpose embedded systems requiring low cost solutions. Such embedded systems are often required to be built around low cost processors like the Intel X86 family or around special purpose DSP processors. They must necessarily support high speed frame capture and facilitate control of robotic and other special purpose equipment. Rather than being general purpose programmable boxes, embedded systems are special purpose application systems which require the vision algorithms to be integrated with the operating system to provide one complete application.

In this paper we present the basic design philosophy of a real-time operating system 'RTKER' developed at IIT Delhi specially for real-time vision based embedded systems. The main features of the operating system are:

1. It uses only light weight threads and provides for flexibility in the choice of the scheduling algorithm. The user can experiment and decide on a scheduling policy to suit his/her requirements.

2. It provides hard real-time guarantees and methods of graceful recovery in case the real-time specifications cannot be met.

3. Most of its API's are similar to those available in Linux. These include the API's for the threads package, API's for system calls, and device driver API's. Consequently any application or device driver code developed on Linux can be ported with minimal changes on to the real-time kernel.

4. It provides a standard C interface to the application programmer enabling him/her to integrate the vision application with the real-time kernel to produce a single executable binary image which can be burnt in to a boot PROM.

5. The kernel is portable. In fact it has been originally developed on Intel X86 and has been succes-

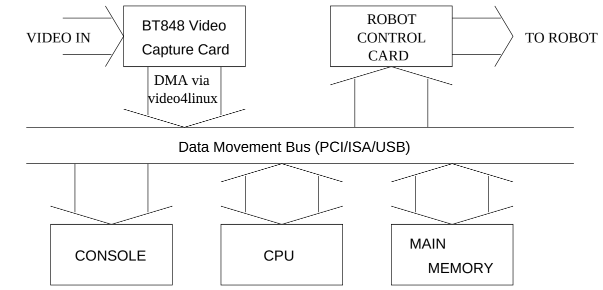


Figure 1: Hardware Architecture for reactive vision applications.

fully ported on to Philips Trimedia (a multimedia DSP processor).

We also present a methodology of design of vision applications wherein vision algorithms are first developed on the popular Linux operating system running on X86 processors and are subsequently ported, with minimal changes in code, on to RTKER with real-time guarantees.

We have chosen three prototypical real-time vision applications, developed earlier at IIT Delhi, to demonstrate our approach:

1. Motion segmentation
2. Collision detection [1]
3. Tracking of isolated objects using active robotic head [2]

We show the hardware system architecture in Figure 1.

We run the three applications in three separate threads and introduce a separate control thread to switch/schedule the individual applications depending on real world events. The system normally runs only the segmentation and the collision detection threads and the robotic device is made to take evasive action when an impending collision is detected. In case the motion segmentation thread detects a moving object in the visual field, then the control is passed on to the tracking thread after suitable initialization. The tracking thread runs till we either lose track or the object disappears, in which case the control is passed on to the segementation/collision detection thread.

In Section 2 we briefly describe the computational requirement of the three vision algorithms and present the overall control structure. In Section 3 we discuss the operating systems requirements to support such applications. In Section 4, we describe our real-time kernel and porting of the vision applications. In Section 5, we present some performance results.

**for** each pixel $p$ in image **do**
  Compute $E_x(p)$, $E_y(p)$ and $E_t(p)$
  **if** $E_x^2 + E_y^2 > T$ **then**
    $v_\perp = -E_t / \sqrt{(E_x^2 + E_y^2)}$

Figure 2: Computation of optic flow parameters. Common code for segmentation and collision detection.

Figure 3: Segmentation in Progress.

## 2 Vision Algorithms

In this section, we briefly present the computation invovlved in each of the three vision functionalities. The collision detection and segmentation algorithms are based on optic flow [7]. They require the computation of normal component of the optic flow (Here $E_x$, $E_y$ and $E_t$ are the image gradients in $x$, $y$ and $t$ (time) directions respectively)

$$v_\perp = -E_t / \sqrt{(E_x^2 + E_y^2)}$$

at dominent edge pixels where

$$\sqrt{E_x^2 + E_y^2} > T \text{ a pre-determined threshold}$$

The psuedo code for the common part is given Figure 2.

Once the above common computation is finished, segmentation and collision detection algorithms run as two separate threads.

### 2.1 Segmentation

In Figure 3, we give typical instances of segmentation. The algorithm proceeds as follows:

1. First, we mark individual pixels as "moving" or not by thresholding on the normal component of the optic flow field.
2. Subsequently, we consider blocks of $8 \times 8$ pixels, and mark each block as moving or not moving by thresholding on the fraction of pixels moving

within the block. We finally mark a block as moving only if 5 out of the 8 blocks around it are moving.

3. Once the moving blocks in the image have been located, we fit the moving pixels into the simple translation model

$$E_t = -(E_x \; E_y)\begin{pmatrix} u \\ v \end{pmatrix}.$$

For a least square minimisation, this equation transforms to

$$z_i = H_i \mathbf{X} + w_i$$

with

$X = (u, v)$,
$H_i = -(E_x E_y)$,
$z_i = E_{t_i}$, and
$w_i$ are *iid* Gaussian noise with zero mean and covariance $R$.

4. The least square solution for $\bar{X}$ minimizes $J(\bar{X}) = \sum_i (Z_i - H_i\bar{X})^t (Z_i - H_i\bar{X})$ Differentiating this equation, the stationary values of $\bar{X}$ satisfy

$$H\hat{X} = Z$$

where, $H = \sum_i (H_i^t H_i)$ and $Z = \sum_i (H_i^t Z_i)$. We solve this equation using LU decomposition.

The centroid and the bounding rectangle of the moving region are passed passed as initialization parameters for the tracker.

### 2.2 Collision Detection

In Figure 4, we show a typical instance of a collision detection. The collision detection algorithm follows [1] :

1. Pixels are classified as moving or not, by thresholding on normal component of the optic flow. This step is identical to step [1] of segmentation and its results can be shared by the two.

2. We use the affine model for the image velocity field, i.e.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} + \begin{bmatrix} u_x & u_y \\ v_x & v_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + O(x^2, xy, y^2)$$

This can be put into a standard measurement equation form for least square solution as:

Figure 4: Collision Detection in Progress.

$$Z_i = H_i \bar{X}$$

where $\bar{X} = (u_0 \; v_0 \; u_x \; u_y \; v_x \; v_y)^t$,
$H_i = (E_{x_i} \; E_{y_i} \; x_i E_{x_i} \; y_i E_{x_i} \; x_i E_{y_i} \; y_i E_{y_i})$, and
$Z_i = -E_{t_i}$.

3. The least square solution for $\bar{X}$ minimizes $J(\bar{X}) = \sum_i (Z_i - H_i\bar{X})^t (Z_i - H_i\bar{X})$ Differentiating this equation, the stationary values of $\bar{X}$ satisfy

$$H\hat{X} = Z$$

where, $H = \sum_i (H_i^t H_i)$ and $Z = \sum_i (H_i^t Z_i)$. We solve this equation using LU decomposition.

4. $v_x + v_y$ gives us the image divergence which is inversely proportional to the time to collision [1]. An interrupt is raised if this number drop below a certain threshold.

### 2.3 Tracking

In Figure 5, we show two typical frames corresponding to tracking an object. The tracker combines the traditional Kalman filter based prediction with additional structure related information and uses this to more accurately predict the position of a corner. Its execution is briefly described below. See [2] for details.

1. Detect corners in the image. Again, this step makes a pass over the entire image, this time on a high resolution (384 × 288) image.

2. Predict the position of each corner in the next frame using the Kalman filter prediction equations (a constant image velocity model) and find the best match by correlation. Compute affine basis for the matched corners.

Figure 5: Snapshots showing a hand being tracked.

3. If basis computation is successful, compute affine structure.

4. Reject outliers using the affine structure and the basis computed in the previous steps.

5. If recomputation of basis is successful, then compute the affine strcuture and force matches using it; locate gaze point using affine structure. If the recomputation was unsuccessful, then set gaze point to the centroid of the matched corners.

6. Move the robot head to point to the gaze point.

### 2.4 Overall Control Structure

The segmenter computes the centroid of the major moving segment and the bounding rectangle in the image. We use this information to initialise the tracker. The two threads work exclusively in time - as soon as the segmenter finds a moving region in the image, it wakes up the tracking thread and suspends itself and the collision detection thread. The control code for accomlishing this is given in Figure 6.

Figure 7 shows the control flow diagram for this scheme.

```
do forever {
    start segmentation and collision detection threads
    if collision detected then {
        raise interrupt (evasive action of robot)
        sleep for while
        restart segmentation and collision detection
    }
    if segment detected then {
        suspend segmentation and collision detection
        threads
        initialize tracker with output of segmentation
        start tracker
        if lost track then
            restart segmentation and collision detection
    }
}
```

Figure 6: Pseudo-code for overall control

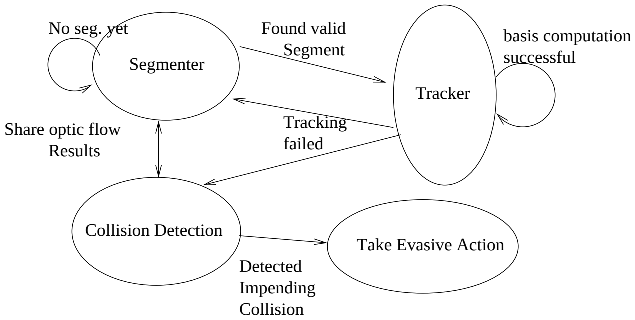


Figure 7: Control Flow Diagram for the Application.

## 3 Operating System Requirements

### 3.1 System Architecture

The hardware of our system consists of a Pentium III based PC running Linux (Kernel version 2.2.12 and above with video4linux enabled) along with a frame grabber card that supports a capture rate of at least 25 frames per second.

For capturing video, we use the *video4linux* API. The API gives us a comprehensive selection of functions in the form of *ioctls* to accomplish video capture. The API has evolved and has now become a standard and device drivers are available for almost all Frame Capture Cards and several of the newer USB cameras. This gives us the added ability to work with any hardware (provided it supports the *video4linux* API). Thus, the testbed to a large extent is hardware independent.

Our Video Capture Card is a BrookTree BT848 card with no on-board memory. The card maps a segment of main memory where it dumps the captured frames. It supports capture of two frames and transmits these frames via Direct Memory Access by downloading the DMA microprogram from the device driver and executing it.

### 3.2 Threading and Pipelining

The first thing to note here is that the video capture card uses DMA in order to transfer the captured frame to the computer's memory. This means that during the execution of a capture the CPU is actually idle. We can use this CPU time to work on a previously captured frame. Thus, we can run two threads here - one for capture, and another computation thread that uses the CPU time while the DMA goes on for the capture.

Two problems arise while using threads -

1. **Memory sharing** - How can two threads work using the same memory for frame buffer? The answer lies in the fact that the threads do not use the same memory for frame buffers! Our video capture card (and most of them) provide support for capturing two frames.

   Even then, the problem is of making the threads share the buffers in a mutually exclusive way. This is a simple producer consumer problem, with two buffers and the capture thread acting as the producer and the computation thread acting as the consumer. The synchronisation problem is solved in a standard way using semaphores.

2. **Scheduling and Preserving the pipeline** - There is an inherent pipelining in the way applications of this class work; the capture thread first fills up a frame, and then the computation thread uses this frame for whatever its purpose is. Hence, there is a need to ensure that this order is preserved at all times.

So, the scheme of things can be expressed as below -

- The capture proceeds into one frame buffer.
- The computation procedure works with the other frame in the meantime.

```
do forever {
    if (odd_ frame) then {
        wait(capture_semaphore_odd);
        compute_function using odd buffer;
        signal(compute_semaphore_odd);
    }
    else {
        wait(capture_semaphore_even);
        compute_function using even buffer;
        signal(compute_semaphore_even);
    }
}
```

(a) Computation thread

```
do forever {
    if (odd_frame) then {
        wait(compute_semaphore_odd);
        capture_function using odd buffer;
        signal(capture_semaphore_odd);
    }
    else {
        wait(compute_semaphore_odd);
        compute_function using even buffer;
        signal(capture_semaphore_odd);
    }
}
```

(b) Capture thread

```
signal(compute_semaphore_odd);
signal(compute_semaphore_even);
```

(c) Initialiaztion

Figure 8: Pseudo-code for synchronization

- The two threads swap buffers when they are both done.

The pseudo-code in Figure 8 gives the synchronization of the two threads.

The scheme has been constructed to ensures two things -

- The two threads do not clash in the memory space.
- The pipelining is preserved, i.e. the capture thread precedes the compute thread.

In Figure 9, we give a typical timing diagram of the various threads in the implementation.

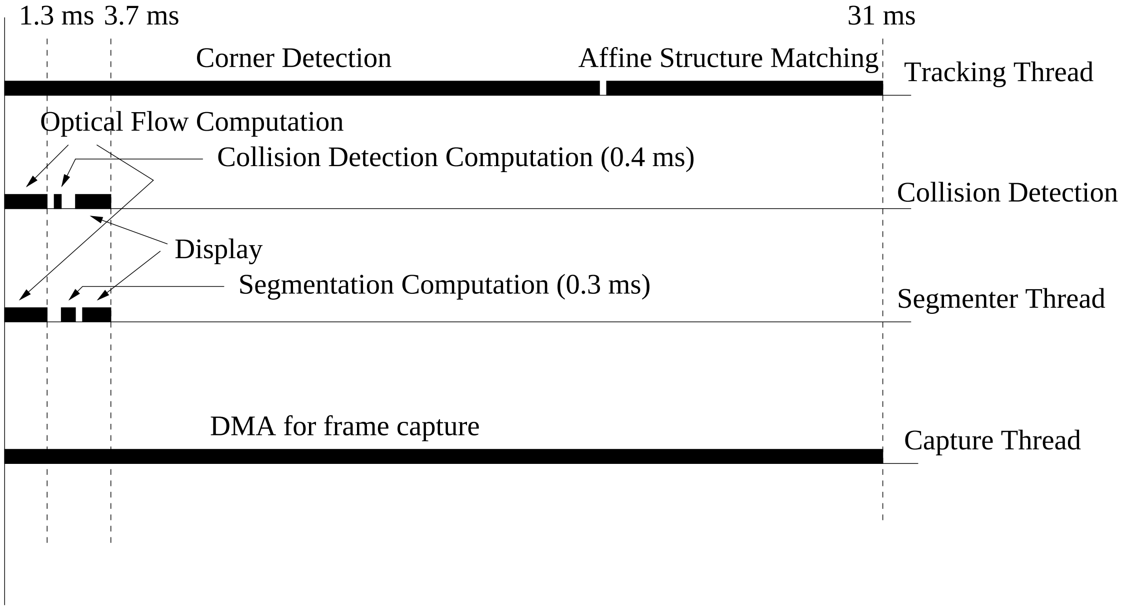


Figure 9: Profile diagram for our application.

### 3.3 Need for Real Time Kernel

Real time reactive vision applications typically have various timing constraints. For example, in a collision detection application, one time constraint would be to process the frames in real-time (25 frames/sec), and another would be to react quickly when a collision is detected. Thus, it is very important to incorporate support for specifying and attempting to meet various timing constraints.

To be truly reliable, such time-constrained computation cannot be done on a general purpose operating system (like Linux) with no real-time scheduling or time guarantees whatsoever. Also, in case a deadline is missed (which again cannot be checked for in a general purpose OS), we need to handle issues related to graceful recovery, such as using partially computed results or quickly substituting old results. To deal with such issues, we need a low-overhead real-time kernel which can do real-time scheduling, check deadlines and take measures for graceful recovery in case deadlines are missed.

Consider the following scenario. Supposing we have a MPEG decoder application that takes in frames and decodes them. This application is basically a periodic application - it takes frames say every 40 ms and has to decode a frame before it receives the next frame. Its deadline then is 40 ms. What happens if suppose the application misses its deadline ? We want to have some mechanism of graceful recovery whereby the application simply takes the previous decoded frame and passes this as the result.

## 4 RTKER - A Real Time Kernel

We have implemented a small real-time uniprocessor operating system kernel. This kernel runs a number of light weight processes or threads, each of which can spawn other threads. These threads can specify their timing constraints using a task description which has the following attributes:

- **Type of task** : periodic or not
- **Periodicity** : time period (if a task is periodic)
- **Ready Time** : time before which task cannot run.
- **Deadline** : time before which task has to be completed
- **Recovery Task** : the task to run if this task misses its deadline.
- **Execution time** : how long the task will take to complete (we can use either worst case or average case execution times, depending on the application). This is not always required.

Note that a general pupose operating system does not allow us to specify various constraints that we are handling, such as periodicity, ready times, deadlines, and recovery tasks. The recovery task can be used to do some sort of graceful recovery when a deadline is missed.

The kernel provides semaphores, which can be used both for synchronisation between threads, and communication between them.

We have ported our kernel to two platforms - Intel X86 (386+), and Philips TriMedia. Our kernel has essentially been implemented as a layer of threading and synchronisation over a standard C library. Thus we make use of a "C interface" to the chip.

### 4.1 Real Time Scheduling

The timing constraints are met using various scheduling policies, which schedule threads depending on the timing constraints specified for them, in a way so as to attempt to meet these constraints.

We have designed our kernel such that the scheduling algorithms are **"pluggable"**, i.e., the kernel itself is independent of the scheduling policy used. This permits us to experiment with various scheduling policies, which provide different types of timing guarantees. For example, some policy may attempt to minimize number of tasks which miss their deadlines, another could try to minimize lateness (time by which a deadline is missed, more precisely, completion time - deadline).

Among the policies that we have tried out is the earliest deadline first policy(EDF) [5, 6], which has the desirable property of being optimal. A policy is said to be optimal if it is able to find a schedule meeting all deadlines if one exists.

Another policy called least laxity(deadline - completion time) first (LLF) [5, 6], provides a different type of guarantee - it minimizes maximum lateness. However it suffers from the drawback of requiring an estimate of completion time of all threads, and poor estimates may lead to skewed scheduling. Compared to LLF, EDF is preferred because it does not need completion time estimates.

### 4.2 Use of real-time kernel for vision applications

All our vision applications need a frame-grabber device which grabs video frames from a camera and makes them available to the application. On the Intel platform, we have ported the Linux device driver for the BT848 framegrabber card to our kernel.

While the frame capture is taking place within the device, computation (on previous frames) can take place at the processor in parallel during this time. When the frame capture is complete, it is passed back to the processor via DMA.

On the TriMedia platform, we do not need a separate framegrabber card as the TriMedia chip itself has video in/out units which can be connected to a camera and used for grabbing frames. Again, grabbing happens in parallel with computation.

We have ported the vision applications to our kernel.

### 4.3 Application Development Flow

We now have two platforms for deploying a vision application - Linux and our own real-time kernel (RTKER). The development flow is as follows :

- Implement the application on Linux on X86, using the pthreads and video4linux API. We maintain a clean separation between frame grabbing and processing.
- Port this implementation to RTKER. The threads and semaphores API of RTKER on X86 or Trimedia is very similar to pthreads, and thus the application can be easily ported.

The frame grabbing API of RTKER differs from Linux, but since we maintain a clean separation between grabbing and processing code, porting is easy. We have used this methodology to successfully port the above mentioned vision applications. Linux, being easy to develop and debug on, is ideal as a first implementation platform. Once the application is robust, we port it to RTKER, in order to impose timing guarantees. The porting needs minimal changes to threading and grabbing code, and no changes to the processing code.

## 5 Results

### 5.1 Profiling for collision detection on TriMedia

For the RealTime Kernel

| Frame Size | time(ms) |
|---|---|
| 300x200 | 25.6 |
| 128x128 | 9.6 |
| 200x200 | 19.0 |
| 384x244 | 36.0 |

### 5.2 Profiling on Intel

All frame sizes are 128 x 96 except tracker which is 384 x 288

| Application | Time (RTKER/ms) | Time (Linux/ms) |
|---|---|---|
| collision detection | 1.8 | 1.9 |
| segmentation | 1.8 | 1.9 |
| tracker | 29 | 33 |

## 6 Conclusion

We have presented a methodology for implementing real-time vision algorithms on PCs and other low-end platforms. A significant contribution has been the development of a real-time kerner with pluggable scheduling policies and real-time guarantees. The kernel has been ported to both X86 and Philips Trimedia platforms. The methodology has been validated by porting three vision applications on both thee platforms.

## References


[1] Alok Mittal, Aditya Valilaya. *Real Time Vision System for Collision Detection*, Journal of Computer Science and Informatics, Special Issue on Robotics and Automation, 25(1), pp. 174-208, March 1995.

[2] Gurmeet Singh Manku, Pankaj Jain, Amit Aggarwal, Lalit Kumar and Subhashis Banerjee. *Object Tracking using Affine Structure for Point Correspondences*, IEEE CVPR'97, June 19-21, San Juan, Puerto Rico.

[3] Michal Irani, Benny Rousso. *Computing Occluding and Transparent Motions.* Int. J. Computer Vision, Vol 12 No. 1, January 1994, pp. 5-16.

[4] Moshe Ben-Ezra, Shmuel Peleg, Benny Rousso. *Motion Segmentation Using Convergence Properties.* ARPA Image Understanding Workshop, November 1994, pp. 1233-1235.

[5] Clifford W. Mercer. *An Introduction to Real Time Operating Systems: Scheduling Theory.* Technical Report. (http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/sur1.review.ps)

[6] J.A. Stankovic et al. *Implications of Classical Scheduling Results for Real Time Systems.* IEEE Computer, Vol. 28, No. 6, pp. 16-25, 1995.

[7] B.K.P.Horn. *Robot Vision.* McGraw Hill, NY, 1986.