Real Time Collision Detection and Avoidance: A Case Study For Design Space Exploration in HW-SW Codesign

Sandeep K. Lodha Shashank Gupta M. Balakrishnan
Subhashis Banerjee

Department of Computer Science and Engineering
Indian Institute of Technology
New Delhi 110016
email: *mbala@cse.iitd.ernet.in*

1 Abstract

In this paper we study a collision detection algorithm and partition it into hardware and software parts to enhance the performance of the system and achieve the real time goal (25 frames/sec). We explore the design space to identify various feasible implementations on on software, hardware, firmware and mixed platforms. The platforms considered are the *Intel's* processors and Sun ULTRA1 for software, MOTOROLA DSP56002 for firmware XILINX FPGAs for hardware. The mixed implementations include combinations of the above. A number of implementations discussed establish that factors such as time constraint, pin count and interface requirements strongly influence the design options.

2 Introduction and Motivation

This paper is based on an real time implementation of collision detection system based on Vision. This means that the system detects imminent collision and takes necessary action to avoid the collision. Collision detection system is the first step towards building applications in machine vision. Any machine vision system should have the capability to safeguard itself against accidental collision with surrounding objects. It is imperative that such decisions be taken in real time so that corrective actions may be taken in real time.Vision algorithms are inherently computation intensive as calculations have to be made at each image point. The purpose of partitioning the algorithm into hardware and software parts is to enhance the performance of the system and achieve the real time goal. It is also important to identify the *minimal* cost platform on which the application(with its specifications) is feasible.

The main motivation of this project was to develop a detailed benchmark for a hardware-software codesign system being developed in the department. The objective was to establish a software execution time estimation procedure, identify critical issues in partitioning and mapping onto DSP and/or FPGA targets.

The outline of the rest of the paper is as follows. Sections 3 and 4 present the theory and algorithm for collision detection respectively. In section 5, profiling and the software execution time of the algorithm has been given which forms the basis of design space exploration in section 6. In section 7, some implementation details are presented with conclusions drawn in section 8.

3 Collision Detection Theory

The relative motion of an observer with respect to an object produces a time varying image on the observer's retina [1]. This time-varying image contains valuable information about the three dimensional shape and motion of the object.

The time variation of an image can be represented by an image flow field (also known as image velocity field). An image flow field is a two dimensional velocity field defined over the eye's retina (or image plane in case of a camera). The velocity at any point is the instantaneous velocity of the image element at that point. Image flow is sometimes also referred to as optic flow [2].

The first order flow derivatives are sufficient to determine the bounds on the time to collision of the observer and the object [3]. Time to collision is the duration of time after which the observer and the object collide due to relative translational motion. It is given as the ratio of the distance of the object and the relative velocity of approach of the observer towards the object.

An interpretation of the above results is given in the image domain in terms of three differential invariants of the image flow field : divergence, curl and shear magnitude. The bound on time to collision is related to these invariants by simple algebraic relations. These three differential invariants can be expressed in terms of only the first order spatial derivatives of image flow. The detailed derivation is given in *Appendix A*.

4 Collision Detection Algorithm

In the presented algorithm, the derivatives with respect to x, y and t are referred to as E_x, E_y and E_t. H, X and Z are matrices. In this algorithm one pass over two images at a time is made. Two images are taken. Using these two images(one is old and the other new) the spatial derivatives as well as the derivative with respect to time is computed. Using these derivatives, valid pixels are found (MOVING pixels). Computation is done on these MOVING pixels to update two matrices H and Z which store the partial sum of the pixels for the image under consideration. When all such pixels are identified, LU Decomposition computes X where X follows HX=Z. Using X we find the *Time to Collision* [3].

Algorithm

1. **For each pixel**
   - (a) **Compute** E_x, E_y, E_t
   - (b) **Compute** $E_x^2 + E_y^2$
   - (c) **If** $E_x^2 + E_y^2 > T1$ **(static threshold)**
   - (d) **If** $E_t/\sqrt{E_x^2 + E_y^2} > T2$ **(moving threshold)**
     - i. $H = H + {H_i}^T H_i$
     - ii. $Z = Z + {H_i}^T Z_i$
2. **Solve** $HX = Z$ **by LU decomposition**
3. **Calculate lower bound on time-to-collision.**

The main objective here is to identify the critical sections of the code which are to be considered for mapping onto the hardware in a manner such that we are able to achieve a frequency of 25 frames/sec on the target platform. For the purpose of further division we divide the algorithm into 3 parts. Steps 1(a), 1(b), 1(c) and 1(c)(i) form the procedure *GradFlow*. This needs to be executed once every pixel. The updation to H and Z is done in the procedure *UpdateSums* which needs to be executed for all pixels which cross the threshold and could vary from few to all pixels (worst case). Steps 2 and 3 constitute the procedure *Signal*.

Processor /Speed(MHz)	GradFlow (secs)	UpdateSums (secs)	Signal (secs)
486/66	2.848	5.058	0.010
Pentium/100	0.926	0.774	0.000
Pentium Pro/133	0.280	0.250	0.000
SUN Ultra1/170	0.390	0.300	0.000

Table 1: Profiling of Collision Detection Algorithm on Various Processors

5 Profiling and Estimation of Software Execution Time

Table 1 summarizes profiling results on various processors on a set of 25 images of size 128*128. The optimizations were done using *Intel's VTUNE* and the profiling was done using *Gprof* utility on UNIX (for SUN Ultra1) and LINUX (for INTEL processors).

After going through the profiling results, we identify that *GradFlow* and *UpdateSums* as being the possible candidates for hardware translation. The operations involved in the two procedures are given in table 2.

6 Design Space Exploration

The performance of software implementations can be directly computed from the profiling results. For hardware implementation, the target technology was chosen to be XILINX 4000 family of static RAM based FPGAs. In this implementation we assume that the system is feasible with the use of sufficient number of FPGAs. For Firmware implementation we chose DSP56002 Processor.[4] This processor runs at 40MHz and gives a maximum of 20 million instructions per second. On DSP, *GradFlow* takes around 1.5 microseconds and *UpdateSums* takes around 7 microseconds.

So we come up with the feasibility chart given in figure 1. The feasibility chart also gives a hint about feasible mixed implementations. They are given below.

- **Pentium and DSP :** With *GradFlow* mapped to DSP and *UpdateSums* onto Pentium, we get a feasible implementation. The only problem here is the inadequate interface bandwidth as for each valid pixel, the DSP has to transfer data to the Pentium. This problem can however be overcome by incurring some latency. The DSP calculates the valid points and stores them in a shared memory. Pentium regularly picks up this data and processes it. Since this involves block memory transfer, the

[2] b stands for bit

Procedure	Addition		Subtraction		Comparison		Multiplication	
	No.	Size	No.	Size	No.	Size	No.	Size
GradFlow	5	8-b [1]	5	8-b	2	20-b	3 1	10-b 20-b
UpdateSums	5 12 10	22-b 30-b 10-b					4 5 12 10	8*11-b, 11*11-b 11*19-b 19*19-b

Table 2: Operation Count for GradFlow and UpdateSums Procedures

Processor	GradFlow	UpdateSums	Both
FPGA	✓	✓	✓
DSP	✓	✗	✗
486	✗	✗	✗
Pentium	✓	✓	✗
Pentium Pro	✓	✓	✓
SUN Ultra1	✓	✓	✓

Figure 1: Feasibility Chart for Real Time implementation of collision detection(25 frames/sec and 128*128 pixels in each frame)

interface overheads incurred here are reduced and the implementation works out to be feasible.

- **Pentium and FPGA :** Mapping either *GradFlow* or *UpdateSums* to the Pentium(and the other onto FPGAs) would introduce the same interface problems as above. A solution similar to the one discussed above would result in a feasible configuration.
- **DSP and FPGA** Implementation of *UpdateSums* on FPGA and *GradFlow* on DSP, is a feasible option.

7 Implementations

7.1 Software Implementation

We describe two software implementations; SUN Ultra1 and Pentium Pro. The implementation on Pentium Pro is shown in figure 2. In case of SUN Ultra1 the Pentium Pro's work has been shifted to SUN Ultra1.

Frame grabbing on the Sun Ultra 1 machine uses the Sun Video card. A simple grab and display routine gives a throughput of 25 Hz, leaving no time for other operations like collision detection. However, a major part of the time involved in grabbing and displaying is

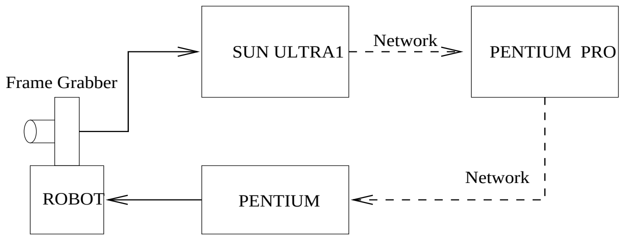

Figure 2: System Configuration

consumed in data transfer between the Sun Video card memory and the frame buffer. The CPU remains idle most of the time during this process.

A virtual two stage pipeline has been built consisting of a image grabber and a collision detector using threads (figure 3). Each stage of the pipeline is implemented by a separate thread of control and all the threads share data among themselves.

In order for the pipeline to function properly, the threads need to be synchronized. The grabber after grabbing Image1 sends a signal to the Collision Detection , which starts operating on Image1. The collision detector then sends a signal to the grabber indicating that it can start grabbing Image1. This synchronization is done using semaphores.

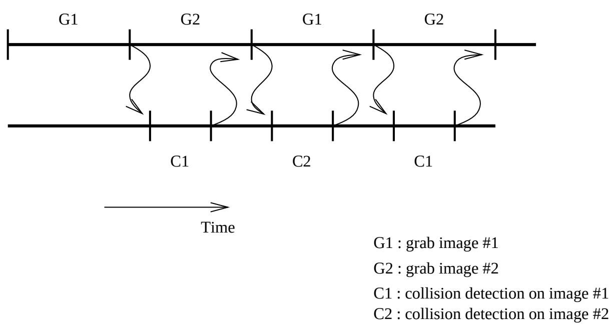

Figure 3: Pipeline Timing Diagram

7.2 DSP-FPGA Implementation

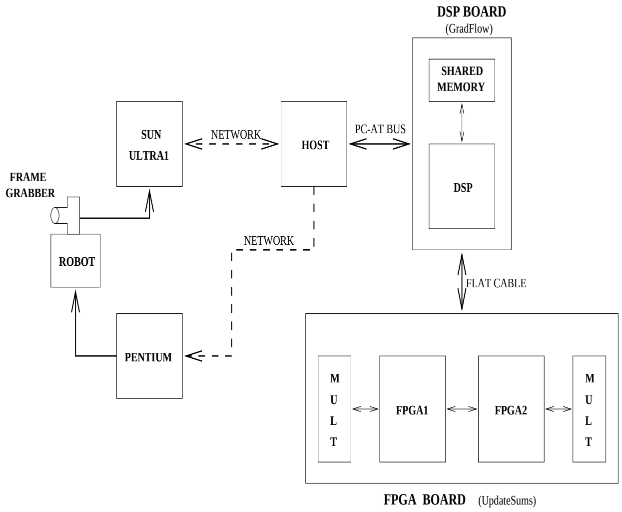

Figure 4: DSP-FPGA System Configuration

The block diagram of the system is shown in figure 4. The work of the various components of the system is explained here :

- **SUN Ultra1** : This grabs the frames and sends them to the host machine over the network. This has been implemented using threads. The first thread does the grabbing of the images while the second thread picks up these images and sends them to the HOST over the network(10 Mbps ethernet).
- **HOST** : The host machine is a PC (either a 486 or a Pentium). This picks up the images from the network and downloads them on the shared memory present on the DSP board. Later, when the whole image has been processed, it picks up the results from the memory shared with the DSP and performs the final computations (*Signal*) to determine the *Time to Collision (TTC)*. If there is an impending collision, it sends a signal on the network to the Pentium machine interfaced with the ROBOT.
- **DSP Board** : The shared memory is used to store the images and the results calculated by the FPGAs. The DSP performs *GradFlow* and passes on the data for the valid points to the FPGAs. It also fetches data from the FPGAs once a whole image has been processed and stores it in the shared memory.
- **FPGA Board** : The two FPGAs together do the computations required in *UpdateSums*. For this they accept data from the DSP. After, the whole image has been processed, they send their results to the DSP. The external multipliers present on this board are used to perform the multiplications required during *UpdateSums*.
- **Pentium** : This acts as an interface to the ROBOT and is connected to the HOST over the network.

7.2.1 Gradflow on DSP :

GradFlow has been implemented on DSP Processor. The Host machine after proper sychronization writes onto the DSP memory and then DSP takes over the memory and does the computations. For each valid pixel DSP has to communicate with *UpdateSums* on FPGAs.

The DSP provides an on board RAM of 32K words of 24 bits each. The HOST has to access that shared memory and download images onto this shared memory. When the DSP + FPGA's complete their processing, the HOST has to again pick the results from the DSP's memory.

The access to the shared memory has to take place in a mutually exclusive manner. To achieve this, DSP provides two signals BREQ and BGRANT. When BREQ is asserted DSP grants all the buses and then asserts BGRANT. The memory to be shared has to be mapped to the address space of both the DSP and the HOST. This memory was mapped to the Upper Memory Blocks of a PC. Both BREQ and BGRANT are implemented using the IO ports.

All synchronization between the DSP and the HOST has been done through the shared memory. To illustrate this the initial synchronization is explained here. DSP when started has to wait till two images have been downloaded by the HOST onto the DSP memory. This has been done through a signal, which has been mapped to the shared memory. So when the HOST finishes downloading the images, it signals the DSP to start. DSP was busy polling this signal and as soon as the DSP detects that the HOST has completed its work, it starts its processing.

7.2.2 UpdateSums on FPGAs :

The behavior of the *UpdateSums* procedure is summarized in the box.

UpdateSums

1. Accepts 5 data values from *Gradflow*.
2. Performs 4 (9 * 7 bit) multiplications to obtain some intermediate results.
3. Performs 17 (9 * 16 bit) and 10 (16 * 16 bit) multiplications on the intermediate results and accumulates the results obtained with the results from the previous executions.
4. When a pass over the whole image has been made, it transfers these results to the *Host* through the *Shared Memory* resident on the *DSP* board. For this it passes over the values to the *DSP*.

The implementation requires the following resources.

- **Multipliers** : The time constraints are such that the multiplications cannot be performed by a shift and add synchronous multiplier. Also, a purely combinational multiplier of this size would not fit onto the FPGAs available. [1] So we decided to use 16-bit *external multiplier* chips to perform the multiplications in a single clock cycle.
- **SRAM** : The results of the multiplications have to be added to the results from the previous calculations. After the whole image has been processed, these accumulations have to be supplied to the *DSP*. For this we need to store 27 values, each 48 bits in size. This has been done by configuring part of the FPGA as an SRAM of the required size.
- **Adder** : For the accumulations, a 48 bit wide adder is required. This is configured on the FPGA itself.
- **Others** : Besides these, all the other components like *Registers, Counters, Muxes* etc. are configured on the FPGA itself.

Due to the timing constraint, we have to issue more than one multiplication at one time. This would require the use of two *external multipliers*. But, due to pin constraints, two multipliers cannot be attached on the same FPGA. So, it was decided to partition the design onto two FPGAs each connected with its own dedicated *external multiplier*.

[1] *Largest device available XC4010E - 400 CLBs*

Some of the important issues while partitioning *UpdateSums* onto two FPGAs are given below:

- The execution time of both the partitions should roughly be the same. This is because *GradFlow* gives the five data values which are picked up by both the FPGAs to begin operation. The supply of the next set of values is constrained by the completion signal from the slower FPGA.
- Since the multiplications are of different sizes, it would benefit to have one FPGA do only the smaller multiplications. This way, a smaller multiplier can be used with this FPGA, if possible.
- When the whole image has been processed, the results have to be supplied by the FPGAs to the DSP. The two FPGAs have to sequentialize their access to the common data bus for this purpose, because it should not happen that both of them are trying to write on the bus at the same time. For this there should be some *synchronization* mechanism.

7.2.3 DSP-FPGA Interface:

DSP56002 processor provides with a few general purpose ports, each line of which is user configurable. The port B which is a 15-bit port has been used in our case. The implementation has been explained below.

- DSP $\Longrightarrow$ FPGA: The communication from DSP to FPGA is implemented as a nonblocking transfer. For this we have used an SRAM buffer. Whenever the DSP is ready to send data to the FPGA, it checks the status buffer (BUFFER). A zero indicates that the buffer is empty and ready to receive data. The DSP waits till BUFFER is zero and then transmits data. After all the five data values are received, BUFFER is asserted. When *UpdateSums* is ready to receive new data (i.e. it has finished the calculations with the previous set of data), it checks if BUFFER is high. If this is not the case, it waits till BUFFER becomes high. Then it reads these values into its registers and clears BUFFER.
- FPGA $\Longrightarrow$ DSP: Once a pass over the whole image has been made, the DSP asserts a signal IM_DONE. When the FPGA finishes all the calculations on the last data set it received from the DSP and it finds IM_DONE high, it configures itself into the output mode and asserts READY (DSP_DATA8). Meanwhile, the DSP reconfigures its data lines as input. Once it receives the

READY signal from the FPGA, it starts giving the DSP_CLK and reading byte wide data.

- Initialization: Initially, at start up and then every time before starting computation on a new image, the DSP has to initialize the FPGA. During this phase, the FPGA clears the SRAM and initializes its control registers. For this, the DSP asserts the INIT signal. The FPGA on sensing this asserts the BUFFER line. After finishing the initialization, the FPGA again sets this line to low. When the DSP finds that the BUFFER has become low, it sets INIT to low and begins computation.

8 Conclusion

In this paper we have described a HW-SW codesign of an important vision algorithm namely *Real Time Collision Detection and Avoidance.* One of the primary objectives was to explore the design space and identify *feasible* mappings for a given specification i.e frame rate and frame size. For the firmware and hardware implementation, we have explored Motorola DSP56002 and XILINX 400 series FPGAs as targets respectively. The design space explored includes not only homogeneous but also mixed targets (e.g DSP + FPGAs). It is established that factors such as time constraint, pin count and interface communication requirements strongly influence the hardware design. For automating the mapping process one interesting observation is that variable bitwidth data operations form a significant aspect of computation in this application domain which traditional high-level synthesis approaches do not handle the same.

A Collision Detection Theory

The image velocities (u, v) in the x and y directions are related to the image brightness gradient by the classical optic flow equation [2].

$$uE_x + vE_y + E_t = 0 \tag{1}$$

where E_x, E_y, E_t represent partial derivatives of the image brightness along x, y and t (time) directions respectively. This equation represents a straight line in the (u, v) space and any point on this line is a possible solution. Since we have one equation and two variables(u and v), these cannot be determined. So writing (u, v) in functional form by neglecting the higher order terms (for a small field of view) we get [3]:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} + \begin{bmatrix} u_x & u_y \\ v_x & v_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Here $(u_0 v_0)$ corresponds to the velocity at origin (center of image) and u_x, u_y, v_x, v_y are spatial derivatives at relative coordinates (x, y). So the optic flow equation which is valid at all pixels now takes the form

$$E_x(u_0 + xu_x + yu_y) + E_y(v_0 + xv_x + yv_y) + E_t = 0$$

where at the i^{th} pixel $E_x = E_{xi}, x = xi, y = yi, E_y = E_{yi}, E_t = E_{ti}$.

In the above equation $(u_0, v_0, u_x, u_y, v_x, v_y)$ are same for all pixels of an image and thus the above equation reduces to

$$\begin{aligned} &H_i X = Z_i, \text{ where} \\ &X = (u_0, v_0, u_x, u_y, v_x, v_y) \\ &H_i = (E_{xi}, E_{yi}, xiE_{xi}, yiE_{xi}, xiE_{yi}, yiE_{yi}) \\ &Z_i = (-E_{ti}) \end{aligned}$$

We have one such equation for each MOVING point and hence we perform standard least square fit to estimate X by minimizing $J(X) = \sum_i (Z_i - H_i X)^t (Z_i - H_i X)$. Taking derivative of $J(X)$ *wrt* X and equating it to 0, we get

$$\begin{aligned} &\textstyle\sum_i(-2H_i^t(Z_i - H_i X)) = 0 \\ &\Rightarrow \textstyle\sum_i (H_i^t Z_i) = \sum_i (H_i^t H_i X) \\ &\Rightarrow HX = Z, \\ &\text{where } H = \textstyle\sum_i (H_i^t H_i) \text{ and } Z = \sum_i (H_i^t Z_i) \end{aligned}$$

This can be solved by LU decomposition to obtain X. Further, we can derive the expression for upper and lower bounds of the approach velocity V_z as

$$V_z = (u_x + v_y)/2 \pm \sqrt{(u_y + v_x)^2 + (u_x - v_y)^2}/2$$

The time to collision is computed as the inverse of the approach velocity.

References

[1] J. J. Koenderink and A. J. van Doorn, "Invariant properties of the motion parallax field due to the movement of rigid bodies relative to an observer", *Optice Acta, Jan'1975, Vol.22, No.9, pp. 773-79.*

[2] B. K. P. Horn, Robot Vision, *McGraw-Hill 1975.*

[3] Alok Mittal, Aditya Vailaya, Subhashis Banerjee M. Balakrishnan, "Real Time Vision System for Collision Detection", *CSI Journal of Computer Science and Informatics, March 1995, pp. 13-29..*

[4] DSP56002 Digital Signal Processor User's Manual. *Motorola.*