Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 175043, 21 pages
doi:10.1155/2009/175043

Research Article

Trade-Off Exploration for Target Tracking Application in a Customized Multiprocessor Architecture

Jehangir Khan,[1] Smail Niar,[1] Mazen A. R. Saghir,[2] Yassin El-Hillali,[1] and Atika Rivenq-Menhaj[1]

[1] *Université de Valenciennes et du Hainaut-Cambrésis, ISTV2 - Le Mont Houy, 59313 Valenciennes Cedex 9, France*
[2] *Department of Electrical and Computer Engineering, Texas A&M University at Qatar, 23874 Doha, Qatar*

Correspondence should be addressed to Smail Niar, smail.niar@univ-valenciennes.fr

Received 16 March 2009; Revised 30 July 2009; Accepted 19 November 2009

Recommended by Markus Rupp

This paper presents the design of an FPGA-based multiprocessor-system-on-chip (MPSoC) architecture optimized for Multiple Target Tracking (MTT) in automotive applications. An MTT system uses an automotive radar to track the speed and relative position of all the vehicles (targets) within its field of view. As the number of targets increases, the computational needs of the MTT system also increase making it difficult for a single processor to handle it alone. Our implementation distributes the computational load among multiple soft processor cores optimized for executing specific computational tasks. The paper explains how we designed and profiled the MTT application to partition it among different processors. It also explains how we applied different optimizations to customize the individual processor cores to their assigned tasks and to assess their impact on performance and FPGA resource utilization. The result is a complete MTT application running on an optimized MPSoC architecture that fits in a contemporary medium-sized FPGA and that meets the application's real-time constraints.

1. Introduction

Technological progress has certainly influenced every aspect of our lives and the vehicles we drive today are no exception. Fuel economy, interior comfort, and entertainment features of these vehicles draw ample attention but the most important objective is to aid the driver in avoiding accidents.

Road accidents are primarily caused by misjudgment of a delicate situation by the driver. The main reason behind the driver's inability to judge a potentially dangerous situation correctly is the mental and physical fatigue due the stressful driving conditions. In cases where visibility is low due to poor weather or due to night-time driving, the stress on the driver increases even further.

An automatic early warning and collision avoidance system onboard a vehicle can greatly reduce the pressure on the driver. In the literature, such systems are called Driver Assistance Systems (DASs). DASs not only automatize safety mechanisms in a vehicle but also help drivers take correct and quick decisions in delicate situations. These systems provide the driver with a realistic assessment of the dynamic behavior of potential obstacles before it is too late to react and avoid a collision.

In the past few years, various types of DASs have been the subject of research studies [1–3]. Most of these works concentrate on visual aid to the driver by using a video camera. Cameras are usually used for recognizing road signs, lane departure warnings, parking assistance, and so forth. Identification of potential obstacles and taking corrective action are still left to the driver. Moreover, cameras have limitations in bad weather and low-visibility conditions.

Our system uses a radar installed in a host vehicle to scan its field of view (FOV) for potential targets, and partitions the scanned data into sets of observations, or tracks [4]. Potentially dangerous obstacles are then singled out and visual and audio alerts are generated for the driver so that a preventive action can be taken. The output signals generated by the system can also be routed to an automatic control system and safety mechanisms in case of the vehicles equipped for fully autonomous driving. We aim to use a low-cost automotive radar and complement it with an embedded tracking system for achieving higher

performance. The objective is to reduce the cost of the system without sacrificing its accuracy and precision.

The principle contributions of this work are as follows.

(i) design and development of a new MTT system specially adapted to the requirements of automotive safety applications,

(ii) feasible and scalable Implementation of the system in a low-cost configurable and flexible platform (FPGA),

(iii) optimization of the system to meet the real time performance requirements of the application and to reduce the hardware size to the minimum possible limit, this not only helps to reduce the energy consumption but also creates room for adding more functionality into the system using the same low-cost platform.

We implement our system in FPGA using a multiprocessor architecture which is inherently flexible and adaptable. FPGAs are increasingly being used as the platforms of choice for implementing complex embedded systems due to their high performance, flexibility, and fast design times. Multiprocessor architectures have also become popular for several reasons. For example, monitoring processor properties over the last three decades shows that the performance of a single processor has leveled off in the last decade. Using multiple processors with a lower frequency, results in comparable performance in terms of instructions per second to a single highly clocked processor and reduces power consumption significantly [5]. Dedicated fully hardware implementation may be useful for high-speed processing but it does not offer the flexibility and programmability desired for system evolution. Fully hardware implementations also require longer design time and are inherently inflexible. Applications with low-power requirements and hardware size constraints, are increasingly resorting to MPSoC architectures. The move to MPSoC design elegantly addresses the power issues faced on the hardware side while ensuring the speed performance.

2. MTT Terminology and Building Blocks

2.1. Terminology. In the context of target tracking applications, a *target* represents an obstacle in the way of the host vehicle. Every obstacle has an associated *state* represented by a vector that contains the parameters defining the target's position and its dynamics in space (e.g., its distance, speed, azimuth or elevation, etc.).

A state vector with n elements is called *n-state* vector. A concatenation of target states defining the target trajectory or movement history at discrete moments in time is called a *track*.

The behavior of a target can ideally be represented by its *true state*. The *true state* of a target is what characterizes the target's dynamic behavior and its position in space in a 100% correct and exact manner. A tracking system attempts to estimate the state of a target as close to this ideal state as possible. The closer a tracking system gets to the *true state*, the more precise and accurate it is. For achieving this goal, a tracking system deals with three types of states

(i) *The Observed State or the Observation* corresponds to the measurement of a target's state by a sensor (radar in our application) at discrete moments in time. It is one of the two representations of the *true state* of the target. The *observed state* is obtained through a *measurement model* also termed as the *observation model* (refer to Section 4.2). The *measurement model* mathematically relates the *observed state* to the *true state*, taking into account the sensor inaccuracies and the transmission channel noises. The sensor inaccuracies and the transmission noises are collectively called *measurement noise*.

(ii) *The Predicted State or the Prediction* is the second representation of the target's *true state*. Prediction is done for the next cycle before the sensor sends the observations. It is a calculated "guess" of the target's true state before the observation arrives. The *predicted state* of a target is obtained through a *process model* (refer to Section 4.1). The *process model* mathematically relates the *predicted state* to the *true state* while taking into account the errors due to the approximations of the random variables involved in the prediction process. These errors are collectively termed as *process noise*.

(iii) *The Estimated State or the Estimate* is the corrected state of the target that depends on both the observation and the prediction. The correction is done after the observation is received from the sensor. The *estimated state* is calculated by taking into account the variances of the observation and the prediction. To get a state that is more accurate than both the observed and predicted states, the estimation process calculates a weighted average of the observed and predicted states favoring the one with lower variance more over the one with larger variance.

In this paper, the term *scan* refers to the periodic sweep of radar field of view (FOV) giving observations of all the detected targets. The FOV is usually a conical region in space inside which an obstacle can be detected by the radar. The area of this region depends upon the radar *range* (distance) and its view angle in azimuth.

The radar *Pulse Repetition Time* (PRT) is the time interval between two successive radar scans. The PRT for the radar unit we are using is 25 ms. This is the time window within which the tracking system must complete the processing of the information received during a scan. After this interval, new observations are available for processing. As we shall see latter, the PRT puts an upper limit on the latency of the slowest module in the application.

2.2. MTT Building Blocks. A generalized view of a Multiple Target Tracking (MTT) system is given in Figure 1. The system can broadly be divided into two main blocks, namely *Data Association* and *Filtering & Prediction*. The two blocks work in a closed loop. The *Data Association* block is

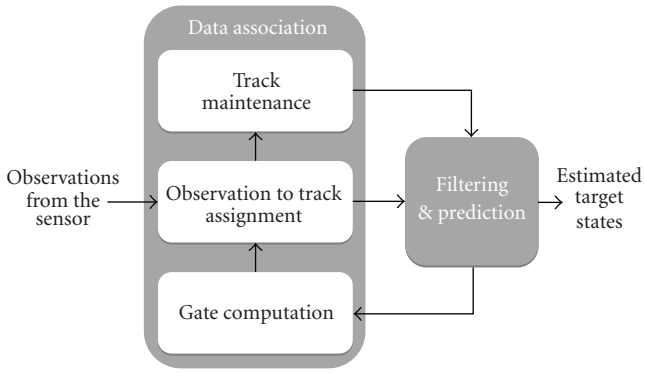

Figure 1: A simplified view of MTT.

further divided into three subblocks: *Track maintenance, Observation-to-Track Assignment* and *Gate Computation*.

Figure 1 represents a text book view of the MTT system as presented in [6, 9]. Practical implementation and internal details may vary depending on the end use and the implementation technology. For example, the *Filtering & Prediction* module may be implemented choosing from a variety of algorithms such as α-β filter [4, 6], *mean-shift algorithm* [7], *Kalman Filter* [6, 8, 9], and so forth. Similarly, the *Data Association* module is usually modeled as an *Assignment Problem*. The assignment problem itself may be solved in a variety of ways, for example, by using the *Auction algorithm* [10], or the *Hungarian/Munkres algorithm* [11, 12].

The choice of algorithms for the subblocks is driven by factors like the application environment, the amount of available processing resources, the hardware size of the end product, the track precision, and the system response time, and so forth.

3. Hardware Software Codesign Methodology

For designing our system, we followed the *y*-chart codesign methodology depicted in Figure 2.

On the right hand side, the software design considerations are taken into account. This includes the choice of the programming language, the software development tools, and so forth. On the left hand side, the hardware design tools, the choice of processors, the implementation platform, and the application programming interface (API) of the processors are defined. In the middle, the MPSoC hardware is generated and the software is mapped onto the processors.

After constructing the initial architecture, its performance is evaluated. If further performance improvement is needed, we track back to the initial steps and optimize various aspects of the software and/or the hardware to achieve the desired performance. The modalities of the "track back" step are mostly dependent on the experience and expertise of the designer. For this work, we used a manual track back approach based on the profiling statistics of the application. As a part of our ongoing work, we are formalizing the design approach to help the designer in choosing the right configuration parameters and mapping strategies.

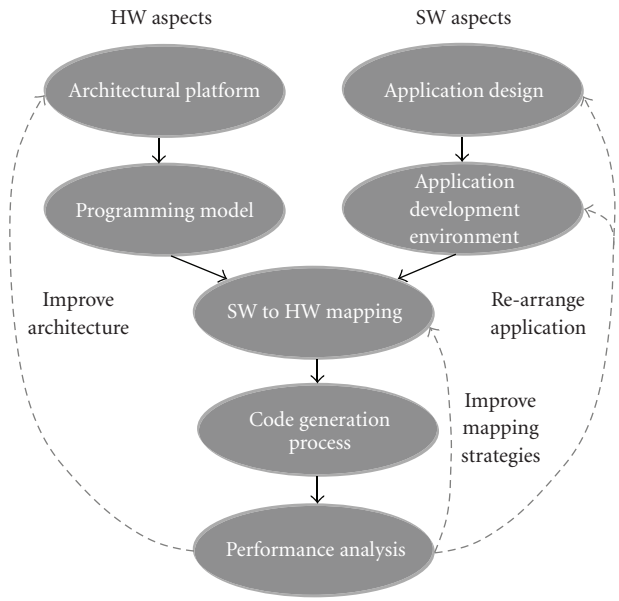

Figure 2: The Y-chart flow for codesign.

Following the codesign methodology, we first developed our application, details of which are described in the next section. After developing the application, we move on to the architectural aspects of the system which are detailed in Section 6.

4. Application Design and Development: Our Approach

As stated above, the choice of algorithms for the MTT system and the internal details are driven by various factors. We designed the application for mapping onto a multiprocessor system. A multiprocessor architecture can be exploited very efficiently if the underlying application is divided into simpler modules which can run in parallel. Moreover, simple multiple modules can be managed and improved independently of one another as long as the interfaces among them remain unchanged.

For the purpose of a modular implementation, we organized our MTT application into submodules as shown in Figure 3. The functioning of the system is explained as follows. Assuming a recursive processing as shown by the loop in Figure 1, tracks would have been formed on the previous radar scan. When new observations are received from the radar, the processing loop is executed.

In the first cycle of the loop, at most 20 of the incoming observations would simply pass through the *Gate Checker*, the *Cost Matrix Generator,* and the *Assignment Solver* on to the filters' inputs. A filter takes an observation as an inaccurate representation of the *true state* of the target, and the amount of inaccuracy of the observation depends on the measurement variance of the radar. The filter estimates the current state of the target and predicts its next state before

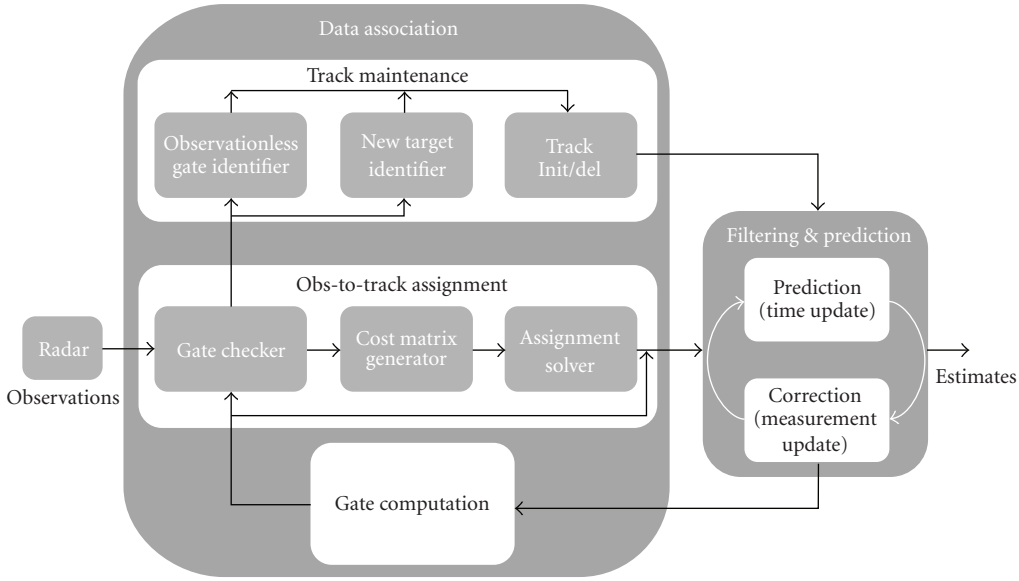

Figure 3: The Proposed MTT implementation.

the next observation is available. The estimation process and the MTT application as a whole rely on mathematical models. The mathematical models we used in our approach are detailed below.

4.1. Process Model. The process model mathematically projects the current state of a target to the future. This can be presented in a linear stochastic difference equation as

$$Y_k = AY_{k-1} + BU_k + W_{k-1}. \quad (1)$$

In (1), Y_{k-1} and Y_k are n-dimensional state vectors that include the n quantities to be estimated. Vector Y_{k-1} represents the state at scan $k-1$, while Y_k represents the state at scan k.

The $n \times n$ matrix A in the difference equation (1) relates the state at scan $k-1$ to the state at scan k, in the absence of either a driving function or process noise. Matrix A is the assumed known *state transition matrix* which may be viewed as the coefficient of state transformation from scan $k-1$ to scan k, in the absence of any driving signal and process noise. The $n \times l$ matrix B relates the optional control input $U_k \in \Re^l$ to the state Y_k, whereas W_{k-1} is zero-mean additive white Gaussian process noise (AWGN) with assumed known covariance Q. Matrix B is the assumed known control matrix, and U_k is the deterministic input, such as the relative position change associated with the host-vehicle motion.

4.2. Measurement Model. To express the relationship between the *true state* and the *observed state* (measured state), a measurement model is formulated. It is described as a linear expression:

$$Z_k = HY_k + V_k. \quad (2)$$

Here Z_k is the measurement or observation vector containing two elements, distance d and azimuth angle θ. The $2 \times n$ observation matrix H in the measurement equation (2) relates the current state to the measurement (observation) vector Z_k. The term V_k in (2) is a random variable representing the measurement noise.

For implementation, we chose the example case given in [8]. In the rest of the paper, the numerical values of all the matrix and vector elements are borrowed from this example. In this example, the matrices and vectors in equations (1) and (2) have the forms shown below:

$$Y_k = \begin{bmatrix} y_{11} \\ y_{21} \\ y_{31} \\ y_{31} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad Z_k = \begin{bmatrix} d \\ \theta \end{bmatrix}. \quad (3)$$

Here y_{11} is the target range or distance; y_{21} is range rate or speed; y_{31} is the azimuth angle; y_{41} is angle rate or angular speed. In vector Z_k, the element d is the distance measurement and θ is the azimuth angle measurement. Matrix B and control input U_k are ignored here because they are not necessary in our application.

The radar Pulse Repetition Time (PRT) is denoted by T and it is 0.025 seconds for the specific radar unit we are using in our project.

Having devised the process and measurement models, we need an estimator which would use these models to estimate the true state. We use the Kalman filter which is a recursive Least Square Estimator (LSE) considered to be the optimal estimator for linear systems with Additive White Gaussian Noise (AWGN) [9, 13].

4.3. Kalman Filter. The *Filtering & Prediction* block in Figure 3 is particularly important as the number of filters employed in this block is the same as the maximum number of targets to be tracked. In our work, we fixed this number at

20 as the radar we are using can measure the coordinates of a maximum of 20 targets. Hence this block uses 20 similar filters running in parallel. If the number of the detected targets is less than 20, the idle filters are switched off to conserve energy.

Given the process and the measurement models in (1) and (2), the Kalman filter equations are

$$\hat{Y}_k^- = A\hat{Y}_{k-1} + BU_k, \tag{4}$$

$$P_k^- = AP_{k-1}A^T + Q, \tag{5}$$

$$K = P_k^- H^T \left(HP_k^- H^T + R\right)^{-1}, \tag{6}$$

$$\hat{Y}_k = \hat{Y}_k^- + K\left(Z_k - H\hat{Y}_k^-\right), \tag{7}$$

$$P_k = (I - KH)P_k^-. \tag{8}$$

Here $\hat{Y}_k^-$ is the state prediction vector; $\hat{Y}_{k-1}$ is the state estimation vector, K is the Kalman gain matrix, P_k^- is the prediction error covariance matrix, P_k is the estimation error covariance matrix and I is an identity matrix of the same dimensions as P_k. Matrix R represents the measurement noise covariance and it depends on the characteristics of the radar.

The newly introduced vectors and matrices in (4) to (8) have the following forms:

$$\hat{Y}_k = \begin{bmatrix} \hat{y}_{11} \\ \hat{y}_{21} \\ \hat{y}_{31} \\ \hat{y}_{41} \end{bmatrix}, \qquad \hat{Y}_k^- = \begin{bmatrix} \hat{y}_{11}^- \\ \hat{y}_{21}^- \\ \hat{y}_{31}^- \\ \hat{y}_{41}^- \end{bmatrix}, \qquad H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

$$R = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} 10^6 & 0 \\ 0 & 2.9 * 10^{-4} \end{bmatrix}, \tag{9}$$

$$Q = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ q_{41} & q_{42} & q_{43} & q_{44} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 330 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.3 * 10^{-8} \end{bmatrix}.$$

Here $\hat{Y}_{11}^-$ is the range prediction, $\hat{Y}_{21}^-$ is the speed prediction, $\hat{Y}_{31}^-$ is the azimuth angle prediction, $\hat{Y}_{41}^-$ is the angular speed prediction, $\hat{Y}_{11}$ is the range estimate, $\hat{Y}_{21}$ the speed estimate, $\hat{Y}_{31}$ is the angle estimate and lastly $\hat{Y}_{41}$ is the angular speed estimate, all for instant k.

Matrices K and P_k^- have the following forms:

$$K = \begin{bmatrix} k_{11} & k_{11} \\ k_{21} & k_{122} \\ k_{31} & k_{32} \\ k_{41} & k_{42} \end{bmatrix}, \qquad P_k^- = \begin{bmatrix} p_{11}^- & p_{21}^- & p_{31}^- & p_{41}^- \\ p_{21}^- & p_{22}^- & p_{23}^- & p_{24}^- \\ p_{31}^- & p_{32}^- & p_{33}^- & p_{34}^- \\ p_{41}^- & p_{42}^- & p_{43}^- & p_{44}^- \end{bmatrix}. \tag{10}$$

Matrix P_k is similar in form to P_k^- except for the superscript "−".The scan index k has been ignored in the elements of these matrices and vectors for the sake of notational simplicity. The Kalman filter cycles through the *prediction-correction* loop shown pictorially in Figure 4. In the prediction step (also called time update), the filter predicts the next state and the error covariance associated with the state prediction using (4) and (5), respectively. In the correction step (also called measurement update), it calculates the filter gain, estimates the current state and the error covariance of the estimation using (6) through (8), respectively.

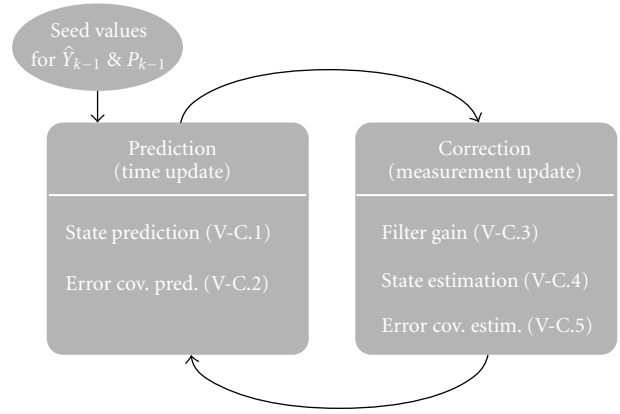

Figure 4: The Kalman filter.

Figure 5 shows the position of a target estimated by the Kalman filter against the true position and the observed position (measured by the radar). The efficacy of the filter can be appreciated by fact that the estimated position follows the true position very closely as compared with the observed position after the 20 transitional iterations.

In the case of a system dedicated to tracking a single target, the estimated state given by the filter would be used to null the offset between the current pointing angle of the radar and the angle at which the target is currently situated. This operation would need a control loop and an actuator to correct the pointing angle of the radar. But since we are dealing with multiple targets at the same time, we have to identify which of the incoming observed states to associate with the predicted states to get the estimation for each target. This is the job of data association function. The data association submodules are explained one by one in the following sections.

4.4. Gate Computation. The first step in data association is the gate computation. The *Gate Computation* block receives the predicted states $\hat{Y}_k^-$ and the predicted error covariance P_k^- from the Kalman Filters for all the currently known targets. Using these two quantities the *Gate Computation* block defines the probability gates which are used to verify whether an incoming observation can be associated with an existing target. The predicted states $\hat{Y}_k^-$ are located at the center of the gates. The dimensions of the gates are proportional to the prediction error covariance P_k^-. If the *innovation* "$Z_k - H\hat{Y}_k^-$" (also called the *residual*) for an observation, is greater than the gate dimensions, the observation fails the gate and hence

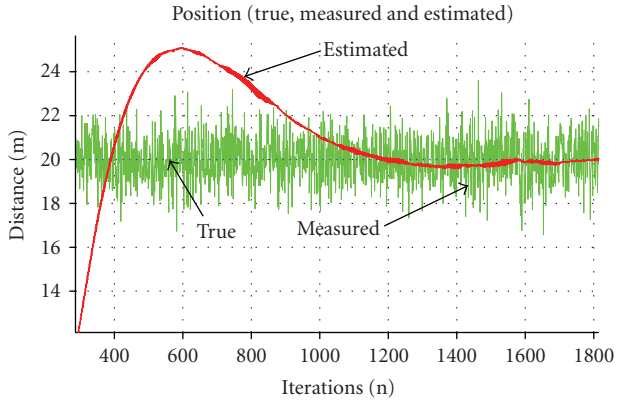

Figure 5: Estimated target position.

it cannot be associated with the concerned prediction. If an observation passes a gate, then it may be associated with the prediction at the center of that gate. In fact, observations for more than one targets may pass a particular gate. In such cases all these observations are associated with the single prediction. The *Gating* process may be viewed as the first level of "screening out" the unlikely prediction-observation associations. In the second level of screening, namely the assignment solver (discussed latter in Section 4.7), a strictly one-to-one coupling is established between observations and predictions.

The gate computation model is summarized as follows.

Define $\tilde{Y}$ to be the *innovation* or the *residual* vector $(Z_k - H\hat{Y}_k^-)$. In general, for a track i, the residual vector is

$$\tilde{Y}_i = Z_k - H\tilde{Y}_i^-. \tag{11}$$

Now define a rectangular region such that an observation vector Z_k (with elements z_{kl}) is said to satisfy the gate of a given track if all elements $\tilde{y}_{il}$ of residual vector $\tilde{Y}_i$ satisfy the relationship

$$\left|Z_{kl} - H\hat{Y}_{il}^-\right| = \left|\tilde{Y}_{il}\right| \le K_{Gl}\sigma_r. \tag{12}$$

In (11) and (12), i is an index for track i, G is gate and l is replaced either by d or by θ, whichever is appropriate (see (17) and (18)). The term σ_r is the *residual standard deviation* and is defined in terms of the *measurement variance* σ_z^2 and *prediction variance* $\sigma_{\hat{y}_k^-}^2$. A typical choice for K_{Gl} is $[K_{Gl} \ge 3.0]$. This large choice of gating coefficient is typically made in order to compensate for the approximations involved in modeling the target dynamics through the Kalman filter covariance matrix [4]. This concept comes from the famous *3 sigma rule* in statistics.

In its matrix form for scan k and track i, (11) can be simplified down to

$$\tilde{Y}_{ik} = \begin{bmatrix} \tilde{y}_{ik11} \\ \tilde{y}_{ik21} \end{bmatrix} = \begin{bmatrix} d_i - \hat{y}_{k11}^- \\ \theta_i - \hat{y}_{k31}^- \end{bmatrix}. \tag{13}$$

Consequently (12) gives

$$\begin{vmatrix} \tilde{y}_{ik11} \\ \tilde{y}_{ik21} \end{vmatrix} \le K_{Gl}\sigma_r. \tag{14}$$

The residual standard deviations for the two state vector elements are defined as follows

$$\sigma_{rd} = \sqrt{r_{11} + p_{22}^-}, \tag{15}$$

$$\sigma_{r\theta} = \sqrt{r_{22} + p_{44}^-}. \tag{16}$$

From (14), (15), and (16), we get

$$\left|\tilde{y}_{ik11}\right| = \left|\tilde{y}_{ikd}\right| \le 3.0\sqrt{r_{11} + p_{22}^-} \tag{17}$$

$$\left|\tilde{y}_{ik21}\right| = \left|\tilde{y}_{ik\theta}\right| \le 3.0\sqrt{r_{22} + p_{44}^-} \tag{18}$$

Equations (17) and (18) together put the limits on the residuals $\tilde{y}_{ikd}$ and $\tilde{y}_{ik\theta}$. In other words, the difference between an incoming observation and prediction for track i must comply with (17) and (18) for the observation to be assigned to track i. The Gate Checker subfunction, explained next, tests all the incoming observations for this compliance.

4.5. Gate Checker. The *Gate Checker* tests whether an incoming observation fulfills the conditions set in (17) and (18). Incoming observations are first considered by the *Gate Checker* for updating the states of the known targets. Gate checking determines which *observation-to-prediction* pairings are probable. At this stage the pairing between the predictions and the observations are not done in a strictly one-to-one fashion. A single observation may be paired with several predictions and vice versa, if (17) and (18) are complied with. In effect, the *Gate Checker* sets or resets the binary elements of an $N \times N$ matrix termed as the *Gate Mask* matrix M where N is the maximum number of targets to be tracked,

$$M = \overbrace{\begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1N} \\ m_{21} & m_{22} & \cdots & m_{2N} \\ \vdots & \vdots & \cdots & \vdots \\ m_{N1} & m_{N2} & \cdots & m_{NN} \end{bmatrix}}^{\text{Predictions}} \Bigg\} \text{observations},$$

$$m_{ij} = \begin{cases} 1 & \text{if obs } i \text{ obey (V-D.7) \& (V6D.8) for track } j, \\ 0 & \text{otherwise.} \end{cases} \tag{19}$$

If an observation i fulfills both the conditions of (17) and (18) for a prediction j, the corresponding element m_{ij} of matrix M is set to 1 otherwise it is reset to 0. Matrix M would typically have more than one $1's$ in a column or a row. The ultimate goal for estimating the states of the targets is to have only one '1' in a row or a column for a one-to-one coupling of observations and predictions. To achieve this goal, the first step is to attach a cost to every possible coupling. This is done by the *Cost Generator* block explained next.

4.6. Cost Matrix Generator. The *Mask matrix* is passed on to the *Cost Matrix Generator* which attributes a cost to each pairing. The costs associated with all the pairings are put together in a matrix called the *Cost Matrix C*.

The cost c_{ij} for associating an observation i with a prediction j is the statistical distance d_{ij}^2 between the observation and the prediction when m_{ij} is 1. The cost is an arbitrarily large number when m_{ij} is 0. The statistical distance d_{ij}^2 is calculated as follows.

Define

$$S_{ij} = HP_k^- H^T + R. \tag{20}$$

Here i is an index for observation i and j is the index for prediction j in a scan, S_{ij} is the residual covariance matrix. The statistical distance d_{ij}^2 is the norm of the residual vector,

$$d_{ij}^2 = \tilde{Y}_{ij}^T S_{ij}^{-1} \tilde{Y}_{ij} \tag{21}$$

$$C = \overbrace{\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1N} \\ c_{21} & c_{22} & \cdots & c_{2N} \\ \vdots & \vdots & \cdots & \vdots \\ c_{N1} & c_{N2} & \cdots & c_{NN} \end{bmatrix}}^{\text{Predictions}} \left.\vphantom{\begin{bmatrix} a \\ a \\ a \\ a \end{bmatrix}}\right\} \text{Observations}$$

$$c_{ij} = \begin{cases} \text{Arbitrary large number} & \text{if } m_{ij} \text{ is } 0 \\ d_{ij}^2 & \text{if } m_{ij} \text{ is } 1 \end{cases} \tag{22}$$

Equation (20) can be written in its matrix form and simplified down to

$$S_{ij} = \begin{bmatrix} p_{\bar{1}\bar{1}}^- + r_{11} & p_{\bar{1}\bar{3}}^- \\ p_{\bar{3}\bar{1}}^- & p_{\bar{3}\bar{3}}^- + r_{22} \end{bmatrix}. \tag{23}$$

Using (13), (21), and (23), d_{ij}^2 is calculated as follows:

$$d_{ij}^2 = \frac{\begin{bmatrix} \tilde{y}_{ik11} & \tilde{y}_{ik21} \end{bmatrix} \begin{bmatrix} p_{\bar{3}\bar{3}}^- + r_{22} & -p_{\bar{1}\bar{3}}^- \\ -p_{\bar{3}\bar{1}}^- & p_{\bar{1}\bar{1}}^- + r_{11} \end{bmatrix} \begin{bmatrix} \tilde{y}_{ik11} \\ \tilde{y}_{ik21} \end{bmatrix}}{((p_{\bar{1}\bar{1}}^- + r_{11}) * (p_{\bar{3}\bar{3}}^- + r_{22}) - p_{\bar{1}\bar{1}}^- * p_{\bar{3}\bar{3}}^-)}. \tag{24}$$

Recall here that $\tilde{y}_{ik11} = \tilde{y}_{ikd}$ and $\tilde{y}_{ik21} = \tilde{y}_{ik\theta}$.

The cost matrix demonstrates a conflict situation where several observations are candidates to be associated with a particular prediction and vice versa. A conflict situation is illustrated in Figure 6.

The three rectangles represent the gates constructed by the Gate Computation module. The predicted states are situated at the center of the gates. Certain parts of the three gates overlap one another. Some of the incoming observations would fall into these overlapping regions of the gates. In such cases all the predictions at the center of the concerned gates are eligible candidates for association with the observations falling in the overlapping regions. The

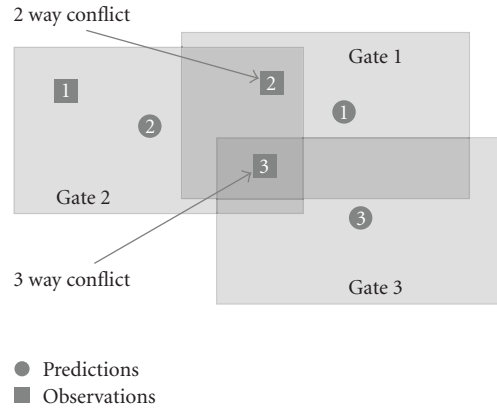

Figure 6: Conflict situation in data association.

mask matrix M and the cost matrix C corresponding to this situation are shown below,

$$M = \overbrace{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}}^{\text{Predictions}} \left.\vphantom{\begin{bmatrix} a \\ a \\ a \end{bmatrix}}\right\} \text{Observations},$$

$$C = \overbrace{\begin{bmatrix} \infty & d_{12}^2 & \infty \\ d_{21}^2 & \infty & \infty \\ d_{31}^2 & d_{32}^2 & d_{33}^2 \end{bmatrix}}^{\text{Predictions}} \left.\vphantom{\begin{bmatrix} a \\ a \\ a \end{bmatrix}}\right\} \text{Observations}. \tag{25}$$

The prediction with the smallest statistical distance d_{ij}^2 from the observation is the strongest candidate. To resolve these kinds of conflicts, the cost matrix is passed on to the *Assignment Solver* block which treats it as the assignment problem [10, 12].

4.7. Assignment Solver. The assignment solver determines the finalized one-to-one pairing between predictions and observations. The pairings are made in a way to ensure minimum total cost for all the finalized pairings. The assignment problem is modeled as follows.

Given a cost matrix C with elements c_{ij}, find a matrix $X = [x_{ij}]$ such that

$$C = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} \text{ is minimized} \tag{26}$$

subject to:

$$\begin{aligned} \sum_i x_{ij} &= 1, \quad \forall j, \\ \sum_j x_{ij} &= 1, \quad \forall i. \end{aligned} \tag{27}$$

Here x_{ij} is a binary variable used for ensuring that an observation is associated with one and only one prediction

and a prediction is associated with one and only one observation. This requires x_{ij} to be either 0 or 1, that is, $x_{ij} \in \{0, 1\}$.

Matrix X can be found by using various algorithms. The most commonly used among them are the *Munkres algorithm* [12] and the *Auction algorithm* [10]. We use the former in our application due to its inherent modular structure.

Matrix X below shows a result of the *Assignment Solver* for a 4×4 cost matrix. It shows that observation 1 is to be paired with prediction 3, observation 2 with prediction 1 and so on:

$$X = \overbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}}^{\text{Predictions}} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} \text{Observations.} \quad (28)$$

The finalized *observation-prediction* pairs are passed on to the the relevant Kalman filters to start a new cycle of the loop for estimating the current states of the targets, predicting their next states and the error covariances associated with these states.

All the steps IV-C through IV-G are repeated indefinitely in the loop in Figure 3. However, there are certain cases where some additional steps have to be taken too. Together these steps are called *Track Maintenance*. The *Track Maintenance* and the circumstances where it becomes relevant are explained in the next section.

4.8. Track Maintenance. The *Track Maintenance* block consists of three functions namely the *New Target Identifier*, the *obs-less Gate Identifier* and the *Track Init/Del*.

In real conditions there would be one or more targets detected in a scan which did not exist in the previous scans. On the other hand there would be situations where one or more of the already known targets would no longer be in the radar range. In the first case we have to ensure if it is really a new target or a *false alarm*. The *New target Identification* subblock takes care of such cases. In the latter case we have to ascertain that the target has really disappeared from the radar FOV. The *Observation-less Gate Identification* subblock is responsible for dealing with such situations.

A new target is identified when its observation fails all the already established gates, that is, when all the elements of a row in the *Gate Mask* matrix M are zero. Such observations are candidates for initiating new tracks after confirmation. The confirmation strategies we use in our work are based on empirical results cited in [4]. In this work, 3 observations out of 5 scans for the same target initiate a new track. The *new target identifier* starts a counter for the newly identified target. If the counter reaches 3 in five scans, the target is confirmed and a new track is initiated for it. The counter is reset every five scans thus effectively forming a sliding window.

The disappearance of a target means that, no observations fall in the gate built around its predicted state. This is indicated when an entire column of the *Mask matrix* is filled with zeros. The tracks for such targets have to be deleted after confirmation of their disappearance. The disappearance is confirmed if the concerned gates go without an observation for 3 consecutive scans out of 5. The *obs-less gate identifier* starts a counter when an empty gate is detected. If the counter reaches 3 in three consecutive scans out of 5, the disappearance of the target is confirmed and its track is deleted from the list. The counter is reset every five scans.

The *Track Init/Del* function prompts the system to initiate new tracks or to delete existing ones when needed.

5. Implementation Platform and the Tools

For the system under discussion we work with Altera's NiosII development kit StratixII edition as the implementation platform. The kit is built around Altera's StratixII EP2S60 FPGA.

5.1. Design Tools. The NiosII development kits are complemented with Altera's Embedded Design Suite (EDS). The EDS offers a user friendly interface for designing NiosII based multiprocessor systems. A library of ready-to-use peripherals and customizable interconnect structure facilitates creating complex systems. The EDS also provides a comprehensive API for programming and debugging the system. The NiosII processor can easily be reinforced with custom hardware accelerators and/or custom instructions to improve its performance. The designer can choose from three different implementations of the NiosII processor and can add or remove features according to the requirements of the application.

The EDS consists of the three tools, namely the QuartusII, the SOPC Builder and the NiosII IDE.

The system design starts with creating a QuartusII project. After creating a project, the user can invoke the SOPC Builder tool from within the QuartusII. The designer chooses processors, memory interfaces, peripherals, bus bridges, IP cores, interface cores, common microprocessor peripherals and other system components from the SOPC Builder IP library. The designer can add his/her own custom IP blocks and peripherals to the SOPC Builder component library. Using the SOPC Builder, the designer generates the Avalon switch fabric that contains all the decoders, arbiters, data path, and timing logic necessary to bind the chosen processors, peripherals, memories, interfaces, and IP cores.

Once the system integration is complete, RTL code is generated for the system. The generated RTL code is sent back into the QuartusII project directory where it can be synthesized, placed and routed and finally an FPGA can be configured with the system hardware.

After configuring the FPGA with a Nios II based hardware, the next step is to develop and/or compile software applications for the processor(s) in the system. The NiosII IDE is used to manage the NiosII C/C++ application and system library or board support package (BSP) projects and makefiles. The C/C++ application contains the software application files developed by the user. The system library includes all the header files and drivers related to the system

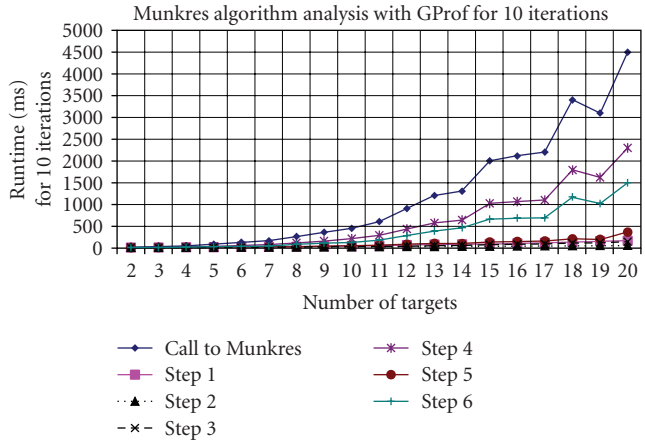

Figure 7: Munkres algorithm profile obtained through GProf.

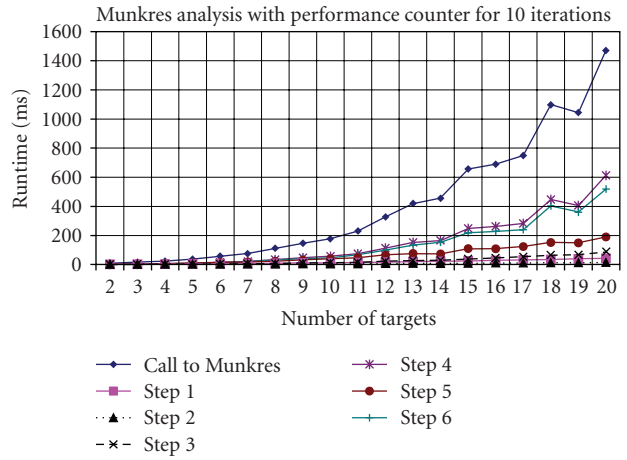

Figure 8: Profile of Munkres Algorithm obtained through Performance Counter.

hardware components. The system library can be used to select project settings such as the choice of *stdin, stdout, stderr* devices, system clock timer, system time stamp timer, various memory locations, and so forth. Thus using the system library, the designer can choose the optimum system configuration for an application.

5.2. Application Profiling and the Profiling Tools. A NiosII application can be profiled in several ways, the most popular among them being the use of the *GProf* profiler tool and the *Performance Counter* peripheral.

5.2.1. GProf. The *Gprof* profiler tool, called *nios2-elf-gprof*, can be used without making any hardware changes to the NiosII system. This tool directs the compiler to add calls to the profiler library functions into the application code.

The profiler provides an overview of the run-time behavior of the entire system and also reveals the dependencies among application modules. However adding instructions to each function call for use by the GNU profiler affects the code's behavior in numerous ways. Each function becomes larger because of the additional function calls to collect profiling information. Collecting the profiling information increases the entry and exit time of each function. The profiling data is a sampling of the program counter taken at the resolution of the system timer tick. Therefore, it provides an estimation, not an exact representation of the processor time spent in different functions [14].

5.2.2. Performance Counter. A performance counter peripheral is a block of counters in hardware that measure the execution time taken by the user-specified sections of the application code. It can monitor as many as seven code sections. A pair of counters tracks each code section. A 64-bit time counter counts the number of clock ticks during which the code in the section is running while a 32-bit event counter counts the number of times the code section runs. These counters accurately measure the execution time taken by designated sections of the C/C++ code. Simple, efficient and minimally intrusive macros are used to mark the start and end of the blocks of interest (the measured code sections) in the program [14].

Figure 7 shows the Munkres algorithm's profile obtained through Gprof. The algorithm was executed on NiosII/s with 100 MHz clock and 4 KB instruction cache. The *call to Munkres* represents the processor time of the overall algorithm for up to 20 obstacles. *Step* 1 through *Step* 6 represent the behavior of individual subfunctions which constitute the algorithm.

Figure 8 shows the profile of the same algorithm obtained through the performance counter for the same processor configuration. Clearly the two profiles have exactly the same form. The difference is that while Gprof estimates that for 20 obstacles the algorithm takes around 4500 ms to find a solution, the performance counter calculates the execution time to be around 1500 ms. This huge difference is due to the overhead added by Gprof when it calls its own library functions for profiling the code.

We profiled the application with both the tools. The Gprof was used for identifying the dependencies and the performance counter for precisely measuring the latencies. All the performances cited in the rest of the paper are those obtained by using performance counter.

6. System Architecture

We coded our application in ANSI C following the generally accepted efficient coding practices and the O3 compilation option. Before deciding to allocate processing resources to the application modules, we profiled the application to know the latencies, resource requirements and dependencies among the modules. Guided by the profiling results, we distributed the application over different processors as distinct functions communicating in a producer-consumer fashion as shown in Figure 9. Similar considerations have been proposed in [2, 15, 16].

The proposed multiprocessor architecture includes different implementations of the NiosII processor and various peripherals as system building blocks.

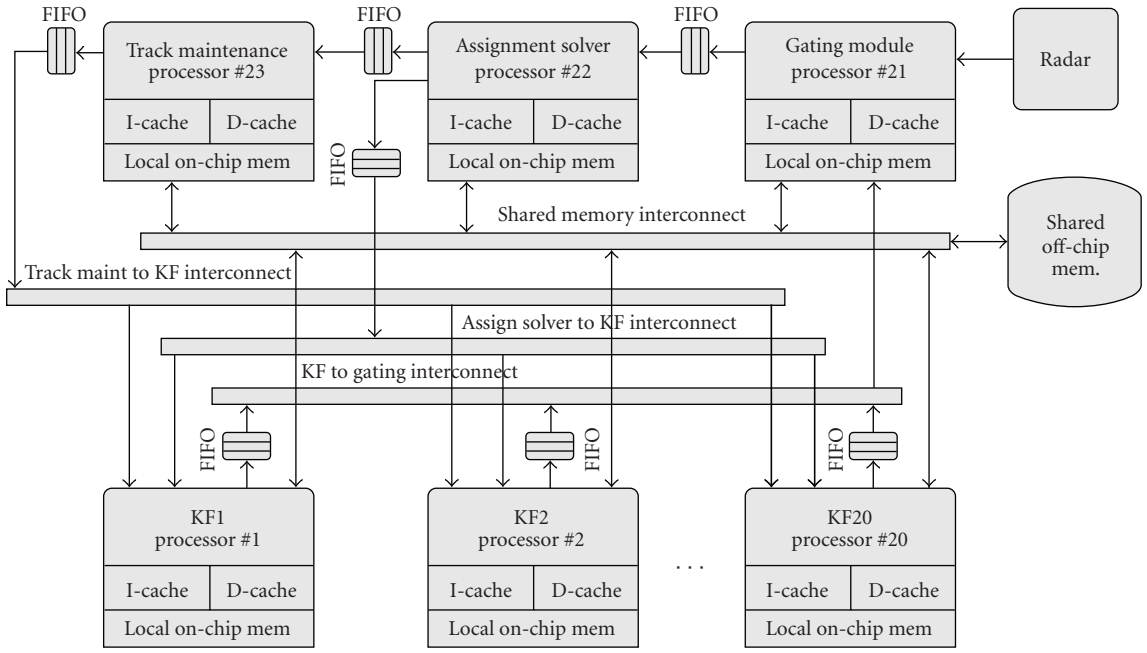

Figure 9: The proposed MPSoC architecture.

The NiosII is a configurable soft-core RISC processor that supports adding or removing features on a system-by-system basis to meet performance or cost goals. A NiosII based system consists of NiosII processor core(s), a set of on-chip peripherals, on-chip memory and interfaces to off-chip memory and peripherals, all implemented on a single FPGA device. Because NiosII processor systems are configurable, the memories and peripherals can vary from system to system.

The architecture hides the hardware details from the programmer, so programmers can develop NiosII applications without specific knowledge of the hardware implementation.

The NiosII architecture uses separate instruction and data buses, classifying it as *Harvard* architecture. Both the instruction and data buses are implemented as Avalon-MM master ports that adhere to the Avalon-MM interface specification. The data master port connects to both memory and peripheral components while the instruction master port connects only to memory components.

The Kalman filter, as mentioned earlier, is recursive algorithm looping around prediction and correction steps. Both these steps involve matrix operations on floating point numbers. These operations demand heavy processing resources to complete in a timely way. This makes the filter a strong candidate for mapping onto a separate processor. Thus for tracking 20 targets at a time, we need 20 identical processors executing Kalman filters.

The *Gate Computation* block regularly passes information to *Gate Checker* which in turn, is in constant communication with *Cost Matrix Generator*. In view of these dependencies, we group these three blocks together, collectively call them the *Gating Module* and map them onto a single processor to minimize interprocessor communication. Interprocessor communication would have required additional logic and would have added to the complexity of the system. Avoiding unnecessary interprocessor communication is also desirable for reducing power consumption.

The *assignment-solver* is an algorithm consisting of six distinct iterative steps [12]. Looping through these steps demands a long execution time. Moreover, these steps have dependencies among them. Hence the assignment solver has to be kept together and cannot be combined with any of the other functions. So we allocated a separate processor to the assignment solver.

The three blocks of the *Track Maintenance* subfunction individually don't demand heavy computational resources, so we group them together for mapping onto a processor. As can be seen in Figure 9, every processor has an I-cache, a D-cache and a local memory. Since the execution time of the individual functions and their latencies to access a large shared memory, are not identical, dependence exclusively on a common system bus would become a bottleneck. Additionally, since the communication between various functions is of *producer-consumer* nature, complicated synchronization and arbitration protocols are not necessary. Hence we chose to have a small local memory for every processor and a large off-chip memory device as shared memory for non critical sections of the application modules. As a result the individual processors have lower latencies for accessing their local memories containing the performance critical codes. In Sections 7.2 and 7.4 we will demonstrate how to systematically determine the optimal sizes of these caches and the local memories.

Every processor communicates with its neighboring processors through buffers. These buffers are dual-port FIFOs with handshaking signals indicating when the buffers

are full or empty and hence regulating the data transfer between the processors. This arrangement forms a system level pipeline among the processors. At the lower level, the processors themselves have a pipelined architecture (refer to Table 1). Thus the advantages of pipelined processing are taken both at the system level as well as at the processor level. An additional advantage of this arrangement is that changes made to the functions running on different processors do not have any drastic effects on the overall system behavior as long as the interfaces remain unchanged. The buffers are flushed when they are full and the data transfer resumes after mutual consent of the concerned processors. The loss of information during this procedure does not affect the accuracy because the data sampling frequency as set by the radar PRT, is high enough to compensate for this minor loss.

Access to the I/O devices is memory-mapped. Both data memory and peripherals are mapped into the address space of the data master port of the NiosII processors. The NiosII processor uses the Avalon switch fabric as the interface to its embedded peripherals. The switch fabric may be viewed as a partial cross-bar where masters and slaves are interconnected only if they communicate. The Avalon switch fabric with the slave-side arbitration scheme, enables multiple masters to operate simultaneously [17]. The slave-side arbitration scheme minimizes the congestion problems characterizing the traditional bus.

In the traditional bus architectures, one or more bus masters and bus slaves connect to a shared bus. A single arbiter controls the bus, so that multiple bus masters do not simultaneously drive the bus. Each bus master requests the arbiter for control of the bus and the arbiter grants access to a single master at a time. Once a master has control of the bus, the master performs transfers with any bus slave. When multiple masters attempt to access the bus at the same time, the arbiter allocates the bus resources to a single master, forcing all other masters to wait.

The Avalon system interconnect fabric uses multimaster architecture with slave-side arbitration. Multiple masters can be active at the same time, simultaneously transferring data to independent slaves. Arbitration is performed at the slave where the arbiter decides which master gains access to the slave only if several masters initiate a transfer to the same slave in the same cycle. The arbiter logic multiplexes all address, data, and control signals from masters to a shared slave. The arbiter logic evaluates the address and control signals from each master and determines which master, if any, gains access to the slave next. If the slave is not shared,it is always available to its master and hence multiple masters can simultaneously communicate with their independent slaves without going through the arbiter.

6.1. System Software. When processors are used in a system, the use of system software or an operating system is inevitable. Many NiosII systems have simple requirements where a minimal operating system or a small footprint system software such as Altera's Hardware Abstraction Layer (HAL) or a third party real-time operating system is sufficient. We use the former because the available third party real time operating systems have large memory footprints while one of our objectives is to minimize the memory requirements.

The HAL is a lightweight runtime environment that provides a simple device driver interface for programs to communicate with the underlying hardware. The HAL application programming interface (API) is integrated with the ANSI C standard library. The API facilitates access to devices and files using familiar C library functions.

HAL device driver abstraction provides a clear distinction between application and device driver software. This driver abstraction promotes reusable application code that is independent of the underlying hardware. Changes in the hardware configuration automatically propagate to the HAL device driver configuration, preventing changes in the underlying hardware from creating bugs. In addition, the HAL standard makes it straightforward to write drivers for new hardware peripherals that are consistent with existing peripheral drivers [17].

6.2. Constraints. The main constraints that we have to comply with are as follows.

We need the overall response time of the system to be less than the radar PRT which is 25 ms. This means that the slowest application module must have less than 25 ms of response time. Hence the first objective is to to meet this deadline.

The FPGA (StratixII EP2S60) we are using for this system, contains a total of 318 KB of configurable on-chip memory. This memory has to make up the processors' instruction and data caches, their internal registers, peripheral port buffers and locally connected dedicated RAM or ROM. Thus the second constraint is that the total on-chip memory utilization must not exceed this limit. We can use off-chip memory devices but they are not only very slow in comparison to the on-chip memory but they also have to be shared among the processors. Controlling access to shared memory needs arbitration circuitry which adds to the complexity of the system and further increase the access time. On the other hand we cannot totally eliminate the off-chip memory for the reasons stated above. In fact we must balance our reliance on the off-chip and on-chip memory in such a way that neither the on-chip memory requirements exceed the available amount of memory nor the system becomes too slow to cope with the time constraints.

Another constraint is the amount of logic utilization. We must choose our hardware components carefully to minimize the use of the programmable logic on the FPGA. Excessive use of programmable logic not only complicates the design and consumes the FPGA resources but also increases power consumption. For these reasons we optimize the hardware features of the the individual processors and leave out certain options when they are not absolutely essential for meeting the time constraints.

7. Optimization Strategies

To meet the constraints discussed above, we plan our optimization strategies as follows.

Table 1: Different NiosII implementations and their features.

	Nios II/f Fast	Nios II/s Standard	Nios II/e Economy
Pipeline	6 Stages	5 Stages	None
HW Multiplier and Barrel Shifter	1 Cycle	3 Cycles	Emulated in Software
Branch Prediction	Dynamic	Static	None
Instr. cache	Configurable	Configurable	None
Data cache	Configurable	None	None
Logic elements	1400–1800	1200–1400	600–700

(i) Select the appropriate processor type for each module to execute it in the most efficient way.

(ii) Identify the optimum cache configuration for each module and customize the concerned processor accordingly.

(iii) Explore the needs for custom instruction hardware for each module and implement the hardware where necessary.

(iv) Identify the performance critical sections in each module and map them onto the fast on-chip memory to improve the performance while keeping the on-chip memory requirements as low as possible.

(v) Look for redundancies in the code and remove them to improve the performance.

In the following sections we explain these strategies one by one.

7.1. Choice of NiosII Implementations. The NiosII processor comes in three customizable implementations. These implementations differ in the FPGA resources they require and their speeds. NiosII/e is the slowest and consumes the least amount of logic resources while NiosII/f is the fastest and consumes the most logic resources. NiosII/s falls in between NiosII/e and NiosII/f with respect to logic resource requirements and speed.

Table 1 shows the salient features of the three implementations of the NiosII processor.

Note here that the code written for one implementation of the processor will run on any of the other two with a different execution speed. Hence changing from one processor implementation to another requires no modifications to the software code.

The choice of the right processor implementation is dependent on the speed requirements of a particular application module and the availability of sufficient FPGA logic resources. Optimization of the architecture trades off the speed for resource saving or vice versa depending on the requirements of the application.

A second criterion for selecting a particular implementation of the NiosII processor is the need (or lack thereof) for instruction and data cache. For example if we can achieve the required performance for a module without any cache, the NiosII/e would be the right choice for running that module.

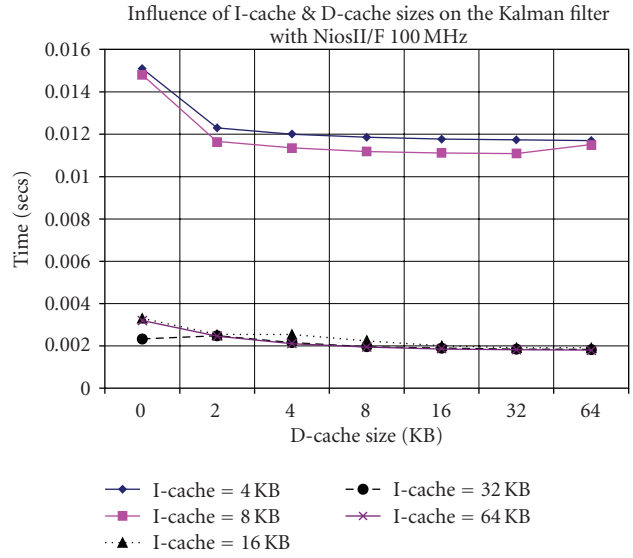

Figure 10: Kalman Filter performances for different Caches Sizes.

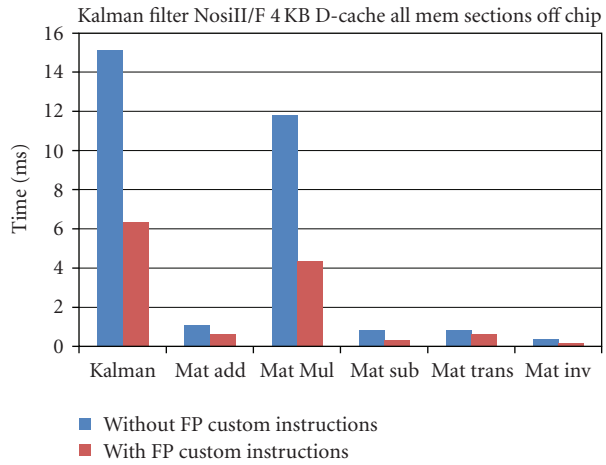

Figure 11: Kalman Filter performances with 4 KB I-cache.

On the other hand, if a certain application module needs instruction and data cache to achieve a desired performance, NiosII/f would be chosen to run it. If only instruction cache can enable the processor to run an application module with the desired performance then we shall use NiosII/s for that module. The objective is to achieve the desired speed with the least possible amount of hardware.

7.2. I-Cache and D-Cache. The NiosII architecture supports cache memories on both the instruction master port (instruction cache) and the data master port (data cache). Cache memory resides on-chip as an integral part of the NiosII processor core. The cache memories can improve the average memory access time for NiosII processor systems that use slow off-chip memory such as SDRAM for program and data storage.

The cache memories are optional. The need for higher memory performance (and by association, the need for

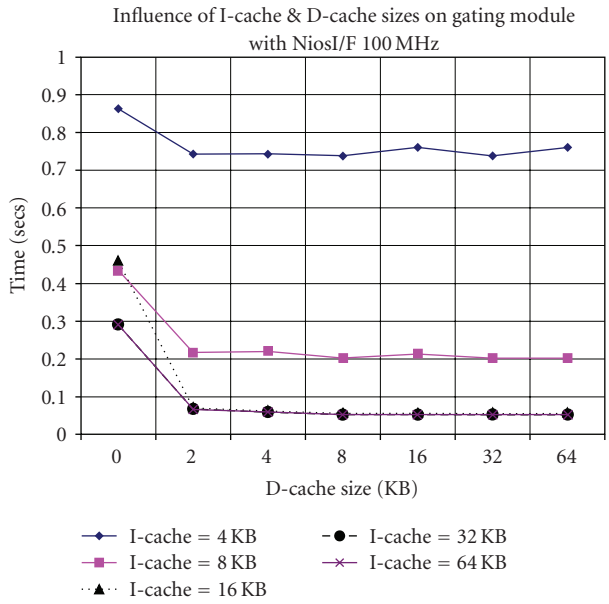

Figure 12: Cache behavior for gating Module.

cache memory) is application dependent. Many applications require the smallest possible processor core, and can trade-off performance for size. A NiosII processor core might include one, both, or neither of the cache memories. Furthermore, for cores that provide data and/or instruction cache, the sizes of the cache memories are user-configurable. The inclusion of cache memory does not affect the functionality of programs, but it does affect the speed at which the processor fetches instructions and reads/writes data.

Optimal cache configuration is application specific. For example, if a NiosII processor system includes only fast, on-chip memory (i.e., it never accesses the slow off-chip memory), an instruction or data cache is unlikely to offer any performance gain. As another example, if the critical loop of a program is 2 KB, but the size of the instruction cache is 1 KB, this instruction cache will not improve execution speed. In fact, an instruction cache may degrade performance in this situation [17]. We must determine the optimum instruction and data cache sizes that are necessary for achieving the desired performance for each module.

Both the Instruction and Data Cache sizes for NiosII/f can range from 0 KB to 64 KB in discrete steps of 0 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, and 64 KB. We experimented with various combinations of I-cache and D-caches sizes to determine the optimum cache sizes for each module. In the following sections we discuss the outcome of these experiments and the guidance that we took from them.

7.2.1. Kalman Filter Cache Requirements. Using the performance counter with a NiosII/F processor, we measured the performance of the Kalman filter with different instruction and cache sizes. Figure 10 shows the influence of I-cache and D-cache sizes on the processor time of the Kalman filter running on NiosII/f with 100 MHz clock using the off-chip RAM.

Two very important conclusions can be drawn from this figure. One, whatever the I-cache or D-cache size, the processor time does not exceed 15 ms. Two, beyond 16 KB I-cache and 2 KB D-cache, the execution time is mostly independent of the D-cache size. Based on these observations, we can say that 16 KB is the optimum I-cache size for the processors executing the Kalman filters. However, as mentioned earlier, for tracking a maximum of 20 obstacles we need 20 of these processors. Viewed in isolation, 16 KB may not seem a large amount of memory but replicating it 20 times is practically not possible. To find out the total amount of memory required by this configuration, we compiled a QuartusII project with a NiosII/f having 16 KB I-cache. The total on-chip block memory used by a single processor, accounted for 7% of the memory available on our FPGA (StratixII EP2S60). Besides, we have to keep in mind that the other processors in the system also have on-chip memory requirements. Consequently we have to settle for a smaller I-cache and hence lower speed to avoid this prohibitive on-chip memory usage.

The good news here is that even with 4 KB I-cache and no D-cache, the processor time is below the 25 ms threshold. Thus an I-cache of 4 KB would be the right choice for the Kalman filters in these circumstances. Furthermore, since we do not a use a D-cache, replacing NiosII/f by NiosII/s would help reduce the logic size from the 1400–1800 LEs range down to the 1200–1400 range which accounts for a sizeable gain in size, considering the 20 processors for the filters.

Figure 11 shows the performance of the Kalman filter on NiosII/s with 4 KB I-cache, no D-cache, 100 MHz clock and using off-chip memory exclusively. Even by keeping all memory sections in the off-chip device and using no floating point custom instructions, the runtime is around 15 ms. Thus we can conserve the scarce on-chip memory by using only 4 KB of I-cache for the processors running Kalman filters without slowing down the system beyond tolerable limits. The on-chip block memory usage in this case drops down to only 3% of that exploitable on the FPGA which is more than 50% drop. For 20 Kalman processors the total on-chip memory usage is 60% of that available to the user.

7.2.2. Gating Module Cache Requirements. The gating module's behavior with respect to the I-cache and D-cache is shown in Figure 12. A remarkable speed up is observed when I-cache size changes from 4 KB to 8 KB and again when it changes from 8 KB to 16 KB. Beyond 16 KB the speed up for the I-cache is insignificant.

The D-cache size does not matter much as long as it is more than zero. The overall processor run time is minimum (70 ms) when I-cache size is 16 KB and D-cache size is 2 KB. Therefore, the right I-cache and D-cache sizes for the Gating module are are 16 KB and 72 KB, respectively. The total on-chip memory usage for the processor with this configuration is 8% of that available on the FPGA. This also includes the memory used by the internal registers of the processor.

Using these cache sizes we charted the performance of the processor while varying the number of obstacles

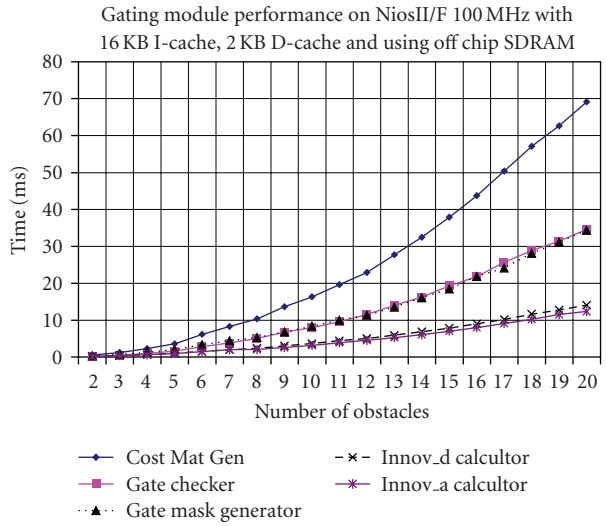

Figure 13: Gating Module performances with 16 KB I-cache and 2 KB D-cache.

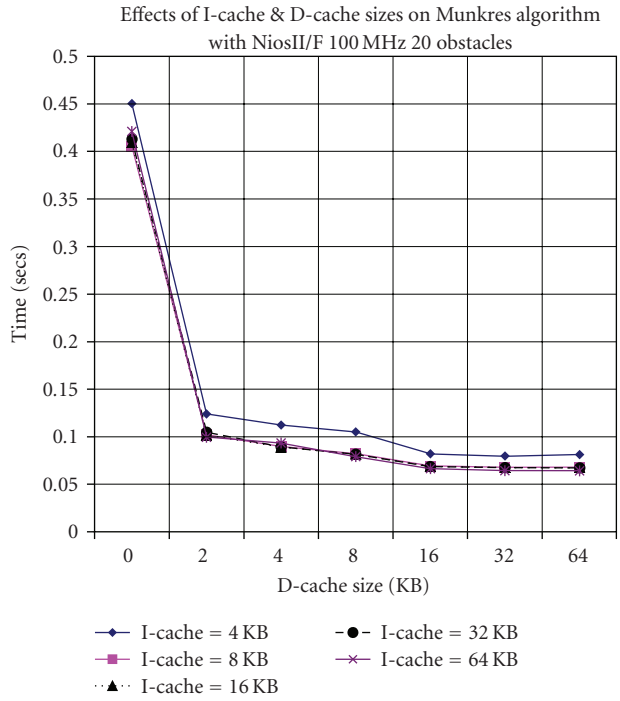

Figure 14: Cache performances for Munkres Algorithm.

from 2 through 20 as shown in Figure 13. The *Innov_d* and *Innov_a* calculators are two subroutines used by the *Gate Mask Generator* function to calculate distance and angle innovations. The sum of the times taken by these two subroutines is roughly equal to the time taken by the *Gate Mask Generator*. The *Gate Checker* and the *Gate Mask Generator* functions are in turn called by *Cost Mat Gen* which is the top level function of the *Gating module*. The *Cost Mat Gen* represents the overall behavior of the whole *Gating Module*.

Although the overall runtime for 20 obstacles is minimum (70 ms) for the given configuration, yet it is much higher than the 25 ms we are aiming for. In Sections 7.3.2 and 7.4.2 we discuss the techniques employed for further improving this execution time.

7.2.3. Munkres Algorithm Cache Requirements. Using a cost matrix with floating point elements and a range of instruction and data cache sizes, Munkres algorithm showed the behavior depicted in Figure 14.

The first observation here is that when the D-cache size is more than zero, the runtime decreases profoundly whatever the I-cache size. Looking closely at the figure we can eliminate 4 KB from the list of competitors for the I-cache size. An 8 KB I-cache along with 16 KB D-cache results in the minimum execution time, that is, 71.07 ms. Hence this is the optimum I-cache/D-cache combination for this module. A NiosII system with these cache sizes uses 9% of the on-chip block memory available on the FPGA.

Figure 15 shows the performance of the algorithm using this system composition for the number of obstacles ranging from 2 to 20.

We notice here that the two main contributors to the total runtime are *Step* 4 and *Step* 6. This is because these two functions contain nested loops and they are invoked multiple times during the solution finding process.

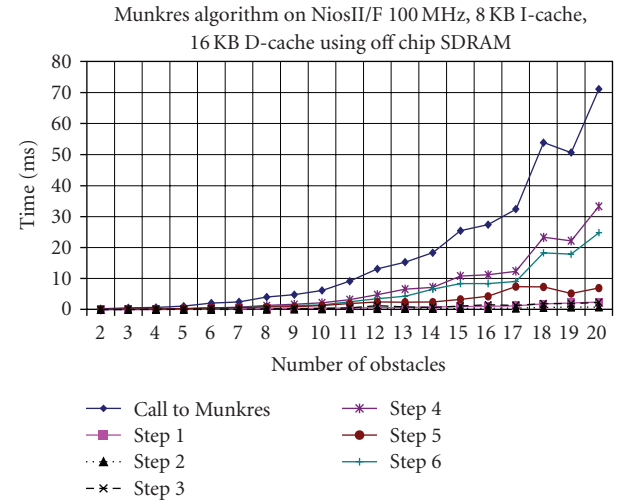

Figure 15: Munkres Algorithm performance with 8 KB I-cache and 16 KB D-Cache.

The overall run time for 20 obstacles, is 71 ms which is higher than the 25 ms bound. We need to further optimize the processor to decrease this runtime. In Sections 7.3.3, 7.4.3, and 7.5 we explain the steps taken for achieving this goal.

7.3. Floating Point Custom Instructions. The floating-point custom instructions, optionally available on the NiosII processor, implement single precision floating-point arithmetic operations in hardware. They accelerate floating-point operations in NiosII C/C++ applications. The basic set of floating

point custom instructions includes single precision floating-point addition, subtraction, and multiplication. Floating-point division is available as an extension to the basic instruction set.

The NiosII software development tools recognize a *C* code that takes advantage of the floating-point instructions present in the processor core. When the floating-point custom instructions are present in the target hardware, the NiosII compiler generates the code to use the custom instructions for floating-point operations, including addition, subtraction, multiplication, division and the *newlib* math library [14].

The best choice for a hardware design depends on a balance among floating-point usage, hardware resource usage and system performance. While the floating-point custom instructions speed up floating-point arithmetic, they add substantially to the size of the hardware design. When resource usage is an issue, it is advisable to rework the algorithms to minimize floating-point arithmetic (see Section 7.5).

We used the floating point custom instructions in the processors to assess the tradeoffs between performance and hardware size for each processor. Sections 7.3.1, 7.3.2 and 7.3.3 examine the outcome of this assessment and the recommendations based thereon.

7.3.1. Kalman Filter and Floating Point Custom Instructions. As mentioned earlier, the Kalman filter's runtime never exceeds 15 ms so there is no need at the moment to accelerate it further at the cost of precious FPGA resources. Nevertheless we tested the floating point custom instructions' impact on the Kalman filter's performance for better understanding the trade-offs and for exploring the opportunities for the eventual future optimization. Figure 11 shows the results of these tests.

An overall speed up of more than 50% is achieved in comparison to the scenario when no floating point custom instructions are used. The most significant improvement is witnessed in case of the *Mat Mul* subfunction. This improvement can be attributed to two factors. One, *Mat Mul* relies heavily on floating point multiplication and second, it is called 11 times in a single iteration of the filter algorithm. Floating point custom instructions are the most effective in such situations hence this remarkable improvement. This speed up comes at the cost of a bulkier hardware. The hardware size increases by 8% when floating point custom instructions are used. We stick to our earlier decision of using regular NiosII/s with 4 KB I-cache and no other add-ons for the Kalman filter in the present work. Since the use of floating point instructions reduces the execution time for the Kalman filter considerably, in our future work we will take this option to process more than one targets per processor.

7.3.2. Gating Module and Floating Point Custom Instructions. In case of the *Gating Module* the use of floating point custom instructions is a necessity rather than an option. The reason is that even with the optimum cache size selection, the *Gating Module* takes 70 ms to execute. Moreover, the Gating module runs on only one processor so we don't have to replicate the floating point custom instructions hardware. Figure 16 shows the performance of the *Gating Module* after the floating point custom instructions are added to the processor.

Floating point custom instructions with NiosII processor for the *Gating Module* improve the overall performance by approximately 50%. If we compare Figure 16 with Figure 13, we notice two interesting differences between the two figures. The first and very obvious difference is the drop from 70 ms to 37 ms of the overall runtime for 20 obstacles.

The second difference is that the curve for the *Gate Checker*, which was earlier above the *Innov_a* and *Innov_d*, is now below them. This change in behavior is due to the fact that in addition to the floating point multiplication and division, the *Gate Checker* uses the *sqrt*() function of the ANSI C math library. The *sqrt*() function itself relies on multiply and divide operations internally. Hence the floating point custom instructions improve the performance of the *Gate Checker* more than the *Innov_a* and *Innov_d* which do not use the *sqrt*() function.

Although by using the floating point custom instruction we managed to bring the execution time from 70 ms down to 37 ms for the *Gating Module*, yet we are still above our 25 ms target. In Section 7.4.2, we explore other possibilities of improving the performance of the *Gating Module* even further.

7.3.3. Munkres Algorithm and Floating Point Custom Instructions. Floating point custom instructions bring Munkres algorithm's execution time from 71 ms down to 47 ms for 20 obstacles, as shown in Figure 17.

Although this is a 33.8% improvement yet 47 ms is almost twice the time we aim to attain, that is, 25 ms. This motivates us to look for ways and means to further decrease this time. To arrive at this goal, we employ several techniques as explained in Sections 7.4.3 and 7.5.

7.4. On-Chip versus Off-Chip Memory Sections. The HAL-based systems are linked using an automatically generated linker script that is created and managed by the NiosII IDE. The linker script controls the mapping of the code and the data within the available memory sections. It creates standard code and data sections (*.text*, *.data*, *and .bss*), plus a section for every physical memory device in the system.

Typically, the *.text* section is reserved for the program instructions. The *.data* section is the part of the object module that contains initialized static data, for example, initialized static variables, string constants, and so forth. The *.bss* (Block Started by Symbol) section defines the space for non initialized static data. The *heap* section is used for dynamic memory allocation, for example, when *malloc*() or *new*() are used in the C or C++ code, respectively. The *stack* section is used for holding the return addresses (program counter) when function calls occur.

In general, the NiosII design flow automatically specifies a sensible default partitioning. However, we may wish to change the partitioning in certain situations. For example, to improve the performance, we can place performance-critical

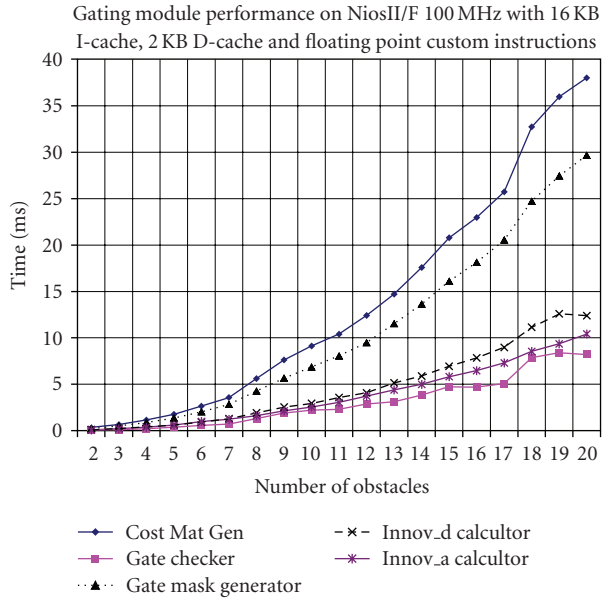

Figure 16: Gating Module performance with 16 KB I-cache, 2 KB D-Cache and Floating Point Custom Instructions.

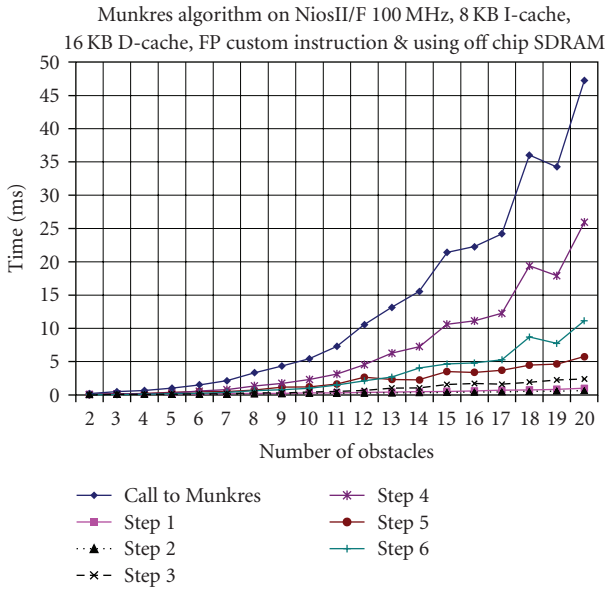

Figure 17: Munkres Algorithm performance with 8 KB I-cache, 16 KB D-Cache and Floating Point Custom Instructions.

code and data in the fast on-chip RAM. In these cases, we have to allocate the memory sections manually.

We can control the placement of the *.text*, *.data*, *heap* and *stack* memory partitions by altering the NiosII system library or BSP settings. By default, the *heap* and the *stack* are placed in the same memory partition as the *.rwdata* section. We can place any of the memory sections in the on-chip RAM if needed, to achieve the desired performance.

Ideally we would put all the memory sections in the fast on-chip memory but the amount of the on-chip memory in the FPGA is limited. Hence we have to rely greatly on the off-chip SDRAM or SSRAM. However accessing the off-chip memory is inherently far slower than the on-chip memory. Moreover, different processors would have to go through the arbitration logic to access the shared off-chip memory device. This would increase the memory access time even further. Consequently, on the one hand we cannot use the off-chip memory exclusively since it would slow the system down beyond the acceptable limits. On the other hand, we have to minimize our dependence on on-chip memory for each processor due to the scarcity of the on-chip memory. We therefore have to balance our reliance on dedicated on-chip memory and the shared off-chip memory without compromising the performance too much.

Table 2: Memory requirements of various application modules.

Section Name	Memory Foot Print
Kalman filter	
Whole Code + Initialized Data	81 KB
.text Section alone	69.6 KB
.data Section alone	10.44 KB
stack Section alone approximately	2 KB
heap section alone approximately	1 KB
Gating module	
Whole Code + Initialized Data	63 KB
.text Section alone	51.81 KB
.data Section alone	8.61 KB
stack Section alone	approximately 2 KB
heap section alone	approximately 1 KB
Munkres algorithm	
Whole Code + Initialized Data	62 KB
.text Section Alone	52.34 KB
.data Section Alone	10.44 KB
.stack Section Alone	approximately 2 KB
heap section alone	approximately 1 KB

Compiling the application modules in the NiosII IDE gives us an estimate of the memory needs of these modules. We selected the appropriate compiler compression options to generate a compact object code for each module. Table 2 summarizes the memory requirements of all the application modules.

We can see in this table that the memory requirements of the whole code and the *.text* sections for all the modules are too high to be accommodated in the on-chip memory. However if a certain module uses *malloc*() or *new*() abundantly, placing the *heap* section in the on-chip memory can improve its speed by a large margin. Similarly if a module makes frequent calls to other functions, putting the *stack* section in the on-chip memory can help achieve a higher execution speed for that module.

We performed experiments by placing the memory sections for the different modules in the off-chip and the on-chip memories and observed some interesting results. These results are discussed in the following sections.

7.4.1. Kalman Filter and Memory Sections. Although the Kalman filter takes 15 ms with only 4 KB I-Cache and no further optimization, yet we investigated the prospects of improving it further through on-chip memory placement. The outcome of this investigation is summarized in Figure 18. As before *Kalman* represents the overall algorithm and the other bars are its constituent subfunctions. These results are obtained with 4 KB I-cache and using floating point instructions.

Even with all memory sections in the off-chip device, the runtime is 6.35 ms while moving only the *stack* section to the on-chip memory reduces this time by more than 50%. Since the *stack* section of the memory requires only 2 KB, we can bring the time down to 3.2 ms by connecting 2 KB of on-chip dedicated memory to the processors for the *stack* and using a NiosII/S with 4 KB of I-cache and floating point custom instructions. One of our experiments showed that if we use a NiosII/f with 16 KB I-cache and 2 KB D-cache and all the other optimizations implemented, we can reduce the processing time for the filter to 1 ms. This opens up a new venue for our future work where we shall route several targets into a Kalman filter to reduce the number of processors for the filters. With this arrangement, we have two options. We can reduce the number of processors for the filters from 20 to 2 and thereby losing some of the flexibility. Alternatively, we can run all the 20 filters on separate processors and guarantee flexibility of being able to switch to other types of filters instead of the Kalman filter depending on the target characteristics. At the moment we are using the latter option.

7.4.2. Gating Module and Memory Sections. The *Gating Module*'s performance for different memory placement experiments is shown in Figure 19. Here we can deduce that a minimum execution time of 22 ms for 20 obstacles can be achieved by keeping all the memory sections in the on-chip memory. But this would require the on-chip RAM to be at least 61 KB. This combined the I-cache and D-cache, adds up to 79 KB. Obviously this is a very high requirement considering the limited amount of the on-chip memory. The next best solution of 23 ms is obtained when we place the *stack* and the *heap* sections in the on-chip memory. There is a very small speed loss but in this case only 3 KB of dedicated on-chip memory is sufficient to get this speed up. Clearly this is a considerable gain in on-chip memory saving compared to the earlier requirement of 79 KB. So the *Gating Module* can operate satisfactorily by using a NiosII/F processor with 16 KB I-cache, 2 KB D-cache, 3 KB dedicated on-chip RAM and floating point custom instructions.

The Innov_d and Innov_a calculators together account for more than half the execution time taken by the gating module (cf. Figure 16). Executing these two on separate processors in parallel will pave the way for scaling the system for more than 20 targets. As an alternative scaling solution we are currently experimenting on DSP-VLIW processors to exploit data level parallelism and hardware accelerators, since after the optimizations, we have enough space available for adding more circuitry (see Section 10).

7.4.3. Munkres Algorithm and Memory Sections. Placing various memory sections on chip does not have a noteworthy influence on the Munkres algorithm performance, although there was some improvement as shown in Figure 20. We gain only 6 ms if all the memory sections are put on chip. The next best gain is achieved by putting the *heap* on chip. this is due to the use of a few *malloc*() statements in the code. While neither of these gains is enough to reduce the execution time below 25 ms, the former is not even feasible given the memory foot print of the algorithm. We have to look elsewhere for a possible and practicable solution. The following section explains our approach to this issue.

7.5. Floating Point versus Integer Cost Matrix For Munkres Algorithm. Munkres algorithm operates on the cost matrix iteratively to find an optimum solution. It looks for the minimum value in every column and row of the cost matrix such that only one value in a row and a column is selected. It comes out with a solution when the sum of the selected elements of the cost matrix reaches its minimum. This procedure remains the same whether the elements of the cost matrix are floating point numbers or integer numbers. We found out that if we truncate the fractional part of the floating point elements of the cost matrix, the final solution is the same as in the case of the floating point cost matrix. Hence we can replace the floating point cost matrix by a "representative" integer cost matrix without sacrificing the accuracy of the final solution. This does not require that the all the elements of the cost matrix have to be different from one another; the algorithm still finds a unique solution even if all the elements of the cost matrix have the same numerical value hence it does not require "distinct" integer values in the cost matrix. The advantage of this manipulation however, is that with the integer cost matrix the mathematical operations become simpler and faster, reducing the runtime of the algorithm by a large margin. Additionally, using an integer cost matrix obviates the need for the floating point custom instruction hardware. Consequently the size of the processor is reduced by 8%.

We made the necessary modifications to the Munkres algorithm and the cost matrix generating function to incorporate this rearrangement. A glimpse of the advantage of these transformations can be seen in Figure 21 which shows the optimal cache configuration for the integer version of the Munkres algorithm.

Certainly 8 KB I-cache and 16 KB D-cache are still the best choices, the point worth noting here is that with the integer cost matrix, the runtime for the overall algorithm drops down to 24 ms as opposed to the 82 ms with floating point cost matrix (refer to Figure 14). This drop takes place while all the memory sections are placed in the off-chip SDRAM.

The Munkres algorithm analysis shows the following facts: Step 4 and Step 6 are the most time consuming subfunctions of the algorithm in the same order (cf. Figure 17). Although, after optimization, for 20 targets a single processor executes the algorithm within the required time interval, yet to scale the system up for more than 20 targets, these subfunctions can be executed on separate processors in

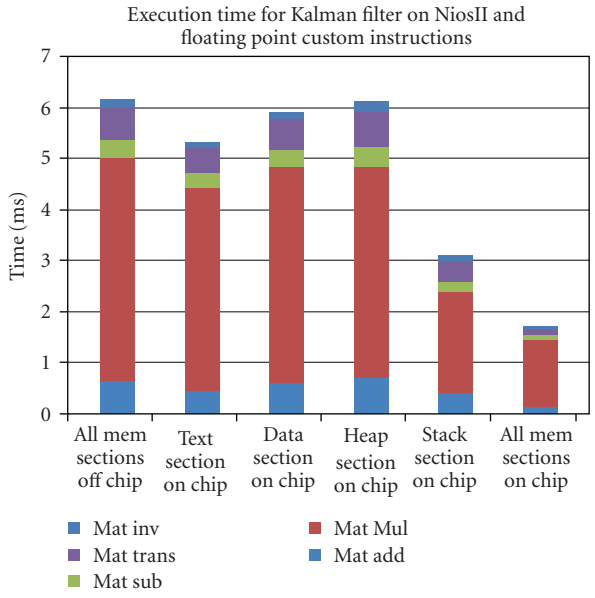

FIGURE 18: Effects of on chip and off chip memory sections on Kalman Filter performances.

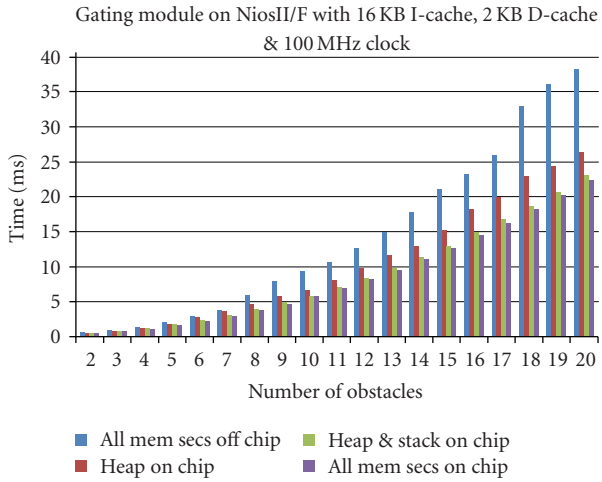

FIGURE 19: Effects of on chip and off chip memory sections on Gating Module.

parallel, to reduce the execution time of the algorithm. In a similar fashion to Gating Module, as an alternative, we are currently experimenting with DSP-VLIW processors to exploit data level parallelism and hardware accelerators.

7.6. Track Maintenance. So far we have not mentioned the track maintenance block of the MTT application in the context of optimization. The reason for this deliberate omission is that a very short processing time is required for this block. A simple NiosII/e processor executes this block in 8 ms. In future we may even eliminate this processor and run the track maintenance block as a second task on one of the other processors.

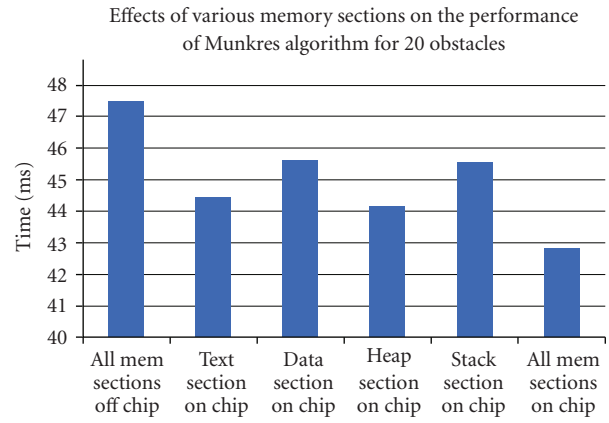

FIGURE 20: Effects of on chip and off chip memory sections on Munkres algorithm performances.

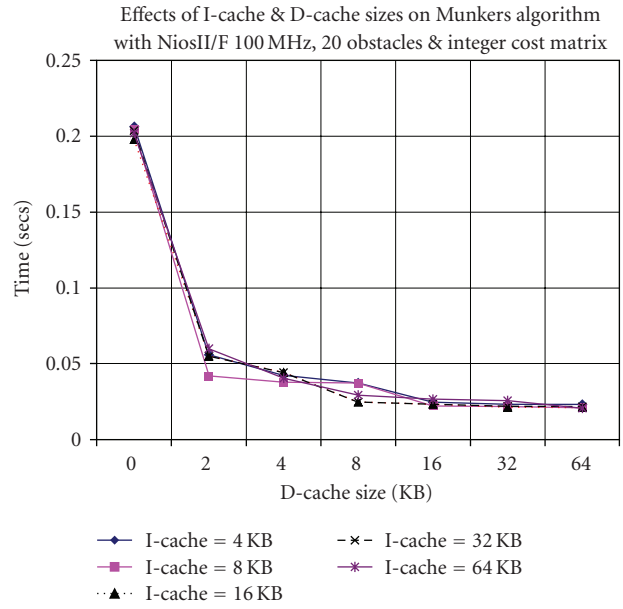

FIGURE 21: Cache Behavior for Munkres Algorithm with Integer Cost Matrix.

8. Discussion

After the success of the last optimization of the Munkres algorithm discussed in Section 7.5, we investigated the other modules for similar optimizations. We found out that this technique cannot be extended to all the modules in the application for the following reasons.

(i) The Kalman filter calculates the predicted states, prediction error covariances, estimated states and estimation error covariance for the targets. The differences in the values of these quantities from one radar scan to another are very small and they occur to the right of the decimal point. It takes hundreds of scans for these changes to flow over to the left of the decimal point. Hence the integer part of these floating point numbers remain unchanged for hundreds of scans. Nevertheless, these small differences play an important

role not only in the filter itself but also in the Gating Module.

In the filter, the estimated state and estimation error covariance are fed back to the prediction stage of the filter. The prediction stage uses them as the basis of predictions for the next scan. If we use just the integer parts of quantities, there would be no change in the estimated values for hundreds of scans. Obviously, this would introduce an error into the predictions. Due to the cyclic feedback between the prediction and correction stages of the filter, an avalanche of errors would be generated in a few seconds.

(ii) The predicted states and the prediction error covariances are also used by the *Gating Module* to locate the centers of the probability gates and to calculate the difference between the measured and the predicted target coordinates, that is, *innov_d* and *innov_a*, respectively. If we use only the integer part of the predicted and measured coordinates, there would be two catastrophic errors introduced into the system. First, because of the non-changing integer parts of the predicted coordinates, the gates would be centered at the same fixed locations for hundreds of scans. Second, for the same reasons, *innov_d* and *innov_a* would remain zero for hundreds of cycles. Zero innovations mean that the predicted coordinates are exactly identical to the measured coordinates which is practically impossible.

(iii) The *Gating Module* uses the prediction error covariance to calculate the dimensions of the probability gates. Using the constant integer part of the covariance would fix the gate dimensions to a constant size for hundreds of scans. This again, is unrealistic and would inject even more error into the system.

(iv) For the Munkres algorithm (the *Assignment Solver*) the case is different. The *Assignment Solver* is the last step of the application loop. By the time the application reaches this step, most of the floating point operations have already been completed resulting in the *Cost Matrix*. The output of the *Assignment Solver* is the *Matrix X* which has either $1's$ or $0's$ as its elements. The 1's in the matrix are used to identify the most probable *observation-prediction* pairs. No arithmetic operations are performed on the *Matrix X*.

Altera provides a tool called C2H compiler which is intended to transform *C* code into a hardware accelerator. At a stage in our work we tried this tool but it turned out that it has some serious limitations. It can be used for codes operating only on integers. So, for the above stated reasons, we could use it only for the Munkres algorithm. But again, the tool can accelerate only a single function and that function too must not involve complex computations. So we could not use it for *Step* 4 and *Step* 6 of the algorithm where we needed it most. The tool simply stops working when we try to accelerate either of these two functions.

In case of small functions (like *Step* 3) where it does work, the hardware size of the accelerator is almost half that of the processor to which it is attached while the speedup is nominal. In brief, if this tool is improved to remove these limitations it can be very useful. In its current status it is far from its stated goals.

9. Related Work

To our understanding, comprehensive literature about the implementation of a complete MTT system in FPGA, does not exist. Works about the application of MTT to DAS's are even harder to find.

Some work has been done on different isolated components of the MTT system but in different contexts. For example an implementation of the Kalman filter only, is proposed in [18]. It is not only limited to the filter but it also is a fully hardware implementation. As mentioned earlier in the introduction, fully hardware designs lack the flexibility and programmability needed for the ever evolving modern day embedded applications. Moreover, the authors report two alternative implementations of the Kalman filter namely the Scalar-Based Direct Algorithm Mapping (SBDAM) and the Matrix-Based Systolic Array Engineering (MBSAE). The former consumes 4564 logic cells whereas the latter consumes 8610 logic cells for a single filter each. Apart from the large sizes, the internal components of both the implementations are manually organized and re-organized to get the desired performance. This is obviously not scalable and repeatable in a complex system like ours where the filter is not the only component to be optimized.

An attempt to implement an MTT system in hardware for a maritime application is documented in [19]. In addition to being a completely hardware implementation, the work presented here is inconclusive.

The data association aspect of MTT has been dealt with nicely in [11] but the physical implementation of the system is not a consideration in this work. Only matlab simulations are reported for that part of the MTT.

Although the title of [20] sounds very close to our work, yet this work describes the theory of the Extended Kalman Filter (EKF) with a smoothing window. The paper discusses the velocity estimation of slow moving vehicles and emphasizes on the necessity of reducing the liberalization errors in the process. While the paper presents a viable solution to the problem of liberalization errors in EKF, the physical implementation of the EKF or the tracking system does not figure among the objectives of the work.

A systolic array based FPGA implementation of the Kalman filter only, is reported in [21]. This work concentrates on the use of a matrix manipulation algorithm (Modified Faddeev) for reducing the complexity of the computation. This article again, presents an interesting account of implementing the Kalman filter in an efficient way. In cases where very fast filtering is the main objective, this may be a good solution.

In fact software forms of the algorithms like EKF [20] and Modified Faddeev based implementation of the Kalman filter [21] can be easily integrated into our system. For example EKF is useful in situations where a target exhibits a abrupt changes in its dynamic behavior as in hilly regions. Similarly, other algorithms like [21] can be added on if required. So the works discussed above can be considered as complementary rather than competitors to our work.

Most of the available works treat the individual components of the MTT (mainly the Kalman filter) in isolation.

However, putting these and other components together to design a coherent MTT application and adapting it to automotive safety utilization, is not a trivial task.

Our work is unique in several aspects. In contrast to the works mentioned above, we consider a complete MTT system implementation. Our reconfigurable MPSoC architecture of the system is inherently flexible, programmable and scalable. Thus it can evolve very easily with advances in technology and with improvements in application algorithms. Moreover, the use of several concurrently running processors meets the overall real time deadlines. Several low frequency processors running concurrently consume less power compared to a single processor with a high clock frequency and doing the same job [5]. The reconfigurability of the processors and other components in our design, allow for customizing them according to application requirements while keeping the hardware size as small as possible. The system we propose is a complete plug-and-play solution that can be easily integrated with the existing electronic systems onboard a vehicle.

Table 3: Summary of the final system.

	Kalman	Gating	Munkres	Track Maint.
Number of proc.	20	1	1	1
NIOSII type	S	F	F	E
I cache in KB	4	16	8	0
D cache in KB	0	2	16	0
Local mem in KB	0	3	3	0
% Mem. used on FPGA	60	8	9	1
FP custom instructions	No	Yes	No	No
Run time in mSec	15	23	24	8

10. Summary and Conclusion

We presented the procedure we adopted for designing and optimizing an application specific MPSoC. The Multiple Target Tracking (MTT) application is designed for Driver Assistance Systems (DAS). These systems are used in vehicles for collision avoidance and early warning for assisting the driver.

First, we described a general view of the MTT application and then we presented our own approach to the design and development of the customized application for Driver Assistance. We developed the mathematical models for the application and coded the application in ANSI C. We divided the application into easily manageable modules that can be executed in parallel.

After developing the application, we profiled it to identify performance bottlenecks and dependencies among the application modules. This helped us in allocating processing resources to the application modules and in laying out our optimization strategies. Using three different hardware implementations of the NiosII soft core embedded processor and other components, we devised a heterogeneous MPSoC architecture for the system.

To formulate our optimization strategies we also identified the constraints to be met. The constraints include the 25 ms time limit for the application execution, the limited amount of available on-chip memory and the size of the system hardware.

To avoid overusing the on-chip memory we optimized the I-cache and D-cache sizes for each application module. Understanding the I-cache and D-cache requirements not only helped us in accelerating the system but also in selecting the right configuration of the NiosII processor for each module.

The optimum cache configurations reduced the execution times by at least 50%. The *Gating Module* and the *Assignment Solver* needed further acceleration to arrive at

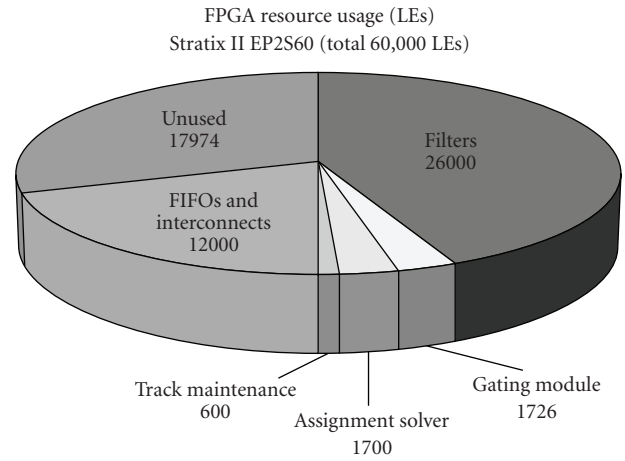

Figure 22: System Hardware Size.

25 ms cut-off set by the radar PRT. We incorporated the floating point custom instructions hardware in the relevant processors to accelerate them further. Floating Point Custom instructions reduced the runtime from 70 ms to 37 ms (47% speedup) for the *Gating Module* and from 71 ms to 47 ms (34% speedup) for the *Assignment Solver*. To bring these times below 25 ms, we needed to speed these modules up even more.

Shifting the whole application to the fast on-chip memory could greatly improve the speed however it is not feasible due to the large memory footprint of the application and the limited amount of the on-chip memory. We experimented with placing different memory sections like the *stack* and the *heap* in the fast on-chip RAM. Placing only the *stack* and the *heap* memory sections on-chip for the *Gating Module*, brought the runtime down to 23 ms which is below the 25 ms cut-off and hence we settled for it.

For the *Assignment Solver* (Munkres algorithm) we could gain only 6 ms in runtime by putting the entire module in the on-chip memory. This gain is neither enough to get us to our goal nor we can afford to put the entire module on chip, in the finalized system.

Exploring the algorithm we found that the final output of the algorithm remains unchanged if we drop down the fractional part of the floating point elements of the input *Cost Matrix*. This manipulation of the input matrix reduced the runtime for the algorithm to 24 ms without compromising the accuracy of the final solution. It also allowed us to do away with the floating point custom instructions. Consequently we use the lighter NiosII/s instead of heavier NiosII/f for the assignment solver.

Speed was not the only objective in choosing the system components and the optimization strategies, we wanted to keep the on-chip memory utilization and the hardware size in check too. We traded speed for FPGA resource economy where we could afford it, for example, in the case of the Kalman filters.

Taking into account the results of the optimizations, we finalized the components and their respective features. Table 3 summarizes the salient features of the finalized architecture with reference to Figure 9. The whole system fits in a single StratixII EP2S60 FPGA. The design uses 42,000 of the 60,000 logic elements (LEs) available on the FPGA as shown in Figure 22 and it meets the runtime constraints of the application.

References

[1] A. Techmer, "Application development of camera-based driver assistance systems on a programmable multi-processor architecture," in *Proceedings of IEEE Intelligent Vehicles Symposium (IV '07)*, pp. 1211–1216, Istanbul, Turkey, June 2007.

[2] M. Beekema and H. Broeders, "Computer Architectures for Vision-Based Advanced Driver Assistance Systems," http://www.xs4all.nl/~hc11/paper_ca_st.pdf.

[3] STMicroelectronics and Mobileye, "Stmicroelectronics and mobileye deliver second generation system-on-chip for vision-based driver assistance systems," Press release, May 2008.

[4] S. Blackman and R. Popoli, *Design and Analysis of Modern Tracking Systems*, Artech House, Boston, Mass, USA, 1999.

[5] Frank Schirrmeister Imperas, Inc., "Multi-core processors: fundamentals, trends, and challenges," in *Proceedings of the Embedded Systems Conference*, 2007, ESC351.

[6] E. Brookner, *Tracking and Kalman Filtering Made Easy*, John Wiley & Sons, New York, NY, USA, 1998.

[7] V. Nedovi, "Tracking moving video objects using mean-shift algorithm," Project Report, http://staff.science.uva.nl/~vnedovic/MMIR2004/vnedovicProjReport.pdf.

[8] Z. Salcic and C.-R. Lee, "FPGA-based adaptive tracking estimation computer," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 37, no. 2, pp. 699–706, 2001.

[9] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, vol. 82, pp. 35–45, 1960.

[10] D. P. Bertsekas and D. A. Castaon, "A forward/reverse auction algorithm for asymmetric assignment problems," http://web.mit.edu/dimitrib/www/For_Rev_Asym_Auction.pdf.

[11] P. Konstantinova, et al., "A study of target tracking algorithm using global nearest neighbor approach," in *Proceedings of the International Conference on Computer Systems and Technologies (CompSysTech '03)*, Sofia, Bulgaria, June 2003.

[12] Munkres' Assignment Algorithm, Modified for Rectangular Matrices http://csclab.murraystate.edu/bob.pilgrim/445/munkres.html.

[13] G. Welch and G. Bishop, "An introduction to the Kalman Filter," 2001, http://www.cs.unc.edu/~welch/kalman/.

[14] Altera Corporation, http://www.altera.com/literature/an/an391.pdf.

[15] R. Joost and R. Salomon, "Advantages of FPGA-based multiprocessor systems in industrial applications," in *Proceedings of the 31st Annual Conference of IEEE Industrial Electronics Society (IECON '05)*, pp. 445–450, Raleigh, NC, USA, November 2005.

[16] H. Penttinen, T. Koskinen, and M. Hnnikinen, "Leon3 MP on Altera FPGA," Final Project Report, August 2007, Altera Innovate Nordic.

[17] Altera Corporation, *NIOS II Processor Reference Handbook*, http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[18] Z. Salcic and C.-R. Lee, "Scalar-based direct algorithm mapping FPLD implementation of a Kalman filter," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 36, no. 3, part 1, pp. 879–888, 2000.

[19] Y. Boismenu, *Etude d'une carte de tracking radar*, Thèse de doctorat, Universit de Bourgogne, Dijon, France, 2000.

[20] Å. Göransson and B. Sohlberg, "Tracking low velocity vehicles from radar measurements," in *Proceedings of the IASTED International Conference on Circuits, Signals, and Systems (CSS '03)*, pp. 51–55, Cancun, Mexico, May 2003.

[21] G. Chen and L. Guo, "The FPGA implementation of Kalman filter," in *Proceedings of the 5th WSEAS International Conference on Signal Processing, Computational Geometry and Artificial Vision*, Malta, 2005.