Embedded multiprocessor architectures for automative driver assistance systems

Jehangir Khan

Université de Valenciennes et du Hainaut-Cambrésis

Numéro d'ordre: 09/43

Embedded Multiprocessor Architectures for Automotive Driver Assistance Systems

Dissertation

Presented and defended on: 26th November 2009

For obtaining

Doctorate Degree

At the Université de Valenciennes et du Hainaut-Cambrésis, France

Specialty: Automatique et Informatique des Systèmes Industriels et Humains

Domain: Embedded Systems

by

Jehangir KHAN

Referees:

Mladen Berekovic	Professeur Faculty of Electrical Engineering, Technique Univeritat (TU) Braunschweig (Allemagne).
Olivier Sentieys	Professeur INRIA Rennes - Bretagne Atlantique.

Examiners:

Mazen Saghir	Professeur University of Texas A&M à Doha (Qatar).
Amer Baghdadi	Maître de conférences, Télécom Bretagne (ex ENST) Brest.

Supervisor:

Pr. Smail NIAR	LAMIH, Université de Valenciennes

Co-supervisor:

Pr. Atika MENHAJ-RIVENQ	IEMN, Université de Valenciennes

Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines – UMR 8530

Acknowledgements

Summary

Automotive crashes are responsible for the highest number of accidental deaths all over the world. Researchers, automotive manufacturers and government authorities around the world are continuously looking for solutions to this problem. Research has shown that half of the accidents can be avoided if a driver is alerted to an impending collision a fraction of a second in advance. A mechanism for warning the driver of an approaching danger is called a Driver Assistance System (DAS).

Accident statistics show that a great majority of the vehicle crashes result from front-end collisions. Hence minimizing frontal collisions would significantly decrease road accidents. To predict a front-end collision sufficiently in advance, the obstacle must be detected from a distance. Moreover, for the DAS to be really effective, an imminent collision must be sensed in all circumstances, especially in poor weather where the DAS is needed most. A radar sensor fulfills both the prerequisites of long range obstacle detection and all-weather operation. However, only detecting obstacles can be useful to a certain extent. To establish whether an obstacle is on a collision course with the host vehicle, its trajectory must be foreseen before it comes close to the host vehicle. Determining the trajectory of a moving object requires its dynamic behavior to be monitored over a period of time. In a real traffic scenario more than one obstacle can pose danger to the host vehicle, hence trajectories of multiple objects have to be monitored simultaneously. An apparatus which is capable of performing such functions is called a Multiple Target Tracking (MTT) system.

In this thesis we propose a DAS using the principles of Multiple Target Tracking to monitor the dynamics of obstacles hundreds of meters ahead and to avoid a collision of the host vehicle with them. While theoretically such a system offers one of the best answers to the road accident problem, its practical implementation is not a trivial task. It involves complex computations and consequently, needs a long processing time. However, to alert a driver to an approaching danger in real time, the computations must be performed very rapidly. We use multiple processors in our system to share the computation load and thereby reduce the processing time. Multiple processors running in parallel not only speed up the computation but also address the power consumption issues of the embedded systems.

We use FPGA (Field Programmable Gate Array) as the implementation platform for our multiprocessor system. FPGAs offer the flexibility needed for the ever evolving embedded systems and they are very cost effective. A multiprocessor system implemented in an FPGA makes its architecture flexible and reconfigurable while the processors can be reprogrammed

when needed. Thus FPGA based multiprocessor systems guarantee flexibility in hardware as well as in software therefore they scale very easily. We optimize the system architecture to minimize its hardware size while still meeting the realtime deadlines of the application. Minimized hardware not only leads to reducing energy consumption of the system but also enables us to fit the system in a smaller FPGA which plays an important role in reducing the cost of the system.

Résumé

Les accidents de véhicules automobiles sont responsables du plus grand nombre de décès dans le monde. Les chercheurs, les constructeurs automobiles et les autorités gouvernementales internationaux sont continuellement à la recherche de solutions pour résoudre ce problème. La recherche a montré que la moitié des accidents peut être évitée si le conducteur est alerté d'une collision imminente une fraction de seconde à l'avance. Un mécanisme d'alerte d'un danger proche est appelé Driver Assistance Systems (DAS).

Les statistiques montrent qu'une grande majorité des accidents de véhicules se passent à la suite d'une collision frontale. Minimiser les collisions frontales devrait donc diminuer considérablement les accidents de la route. Pour prévoir une collision frontale suffisamment à l'avance, l'obstacle doit être détecté à distance. En outre, pour que le DAS soit réellement efficace, une collision imminente doit être prévue en tenant compte de toutes les circonstances : par exemple plus il fait mauvais, plus le DAS est nécessaire. Un capteur radar remplit les conditions préalables de détection d'obstacles à longue portée en tenant compte des conditions météorologiques. Toutefois, seule la détection des obstacles peut être utile dans une certaine mesure. Pour déterminer si un obstacle se trouve sur une trajectoire de collision avec le véhicule d'accueil, sa trajectoire doit être prévue avant qu'il n'arrive près du véhicule d'accueil. La détermination de la trajectoire d'un objet en mouvement exige que son comportement dynamique soit suivi sur une période de temps. Dans un scénario de trafic réel, plus d'un obstacle peut être considéré comme un danger pour le véhicule d'accueil, c'est pourquoi les trajectoires d'objets multiples doivent être surveillées simultanément. Un appareil capable d'exercer de telles fonctions est appelé un système de suivi d'obstacles multiples ou Multiple Target Tracking (MTT).

Dans cette thèse nous proposons un DAS en utilisant les principes du MTT pour suivre la dynamique d'obstacles situés à plus d'une centaine de mètres et pour éviter une collision du

véhicule hôte avec ceux-ci. En théorie, un tel système offre une des meilleures réponses au problème des accidents de la route, mais sa mise en œuvre pratique n'est pas une tâche triviale. Elle implique des calculs complexes et, par conséquent, les besoins de traitement prennent du temps. Cependant, pour aviser le conducteur d'un danger imminent en temps réel, les calculs doivent être effectués très rapidement. Nous utilisons plusieurs processeurs dans notre système afin de partager la charge de calcul et de réduire ainsi le temps de traitement. Les processeurs multiples fonctionnant en parallèle permettent non seulement d'accélérer le calcul, mais aussi d'aborder les questions de consommation d'énergie du système embarqué.

Nous utilisons des FPGA (Field Programmable Gate Array) comme plateforme de mise en œuvre de notre système multiprocesseur. Les FPGA offrent la souplesse nécessaire pour les systèmes embarqués en constante évolution et sont très rentables. Un système multiprocesseur réalisé dans un FPGA rend son architecture flexible et reconfigurable tandis que les processeurs peuvent être reprogrammés si nécessaire. Ainsi les systèmes multiprocesseurs à base de FPGA garantissent une souplesse dans le matériel ainsi que dans les logiciels, et par conséquent leur passage à échelle est aisé. Nous optimisons l'architecture du système afin de minimiser la taille du matériel tout en respectant les délais en temps réel de l'application. La minimisation du matériel ne conduit pas seulement à réduire la consommation d'énergie du système, mais nous permet aussi d'adapter le système dans un FPGA plus petit. Cela joue un rôle important dans la réduction du coût du système.

Scientific output of the work

1. Trade-off exploration for target tracking application in a customized multiprocessor Architecture*(to appear)*. *Design and Architectures for Signal and Image Processing*, Special Issue of the EURASIP Journal of Embedded Systems.

2. Radar Based Collision Avoidance System Implementation in a Reconfigurable MPSoC *The 9th International Conference on ITS Telecommunications, ITST 2009, 20-22 Oct. 2009 Lille France.*

3. Driver Assistance System Design and its Optimization for FPGA Based MPSoC *IEEE Symposium on Application Specific Processors, SASP 2009, July 27-28, 2009 San Francisco, California*

4. Multiple Target Tracking System Design for Driver Assistance Application., *Design & Architectures for Signal and Image processing*, Brussels, Belgium, November 2008.

5. An MPSoC Architecture for the Multiple Target Tracking Application in Driver Assistant System.*19th IEEE International Conference Application-specific Systems, Architectures and Processors (ASAP)* Leuven, Belgium, July 2008.

6. A Multiple Target Tracking SoC for Transport Security. *Colloque du GDR SOC SIP*, Paris, June 2007.

7. Système d'aide à la conduite par l'utilisation du suivi d'obstacles multiples sur architecture multiprocesseurs reconfigurable. *Séminaire Systèmes Embarqués : Sécurité, Confort, Aide à la conduite*, ESIEE Amiens, May 2008.

8. An Efficient Digital Correlator Architecture for an Anti-Collision Radar System. *ACES 2006*, Edegem Belgium, October 2006.

9. Resource Optimization in Ati-collision Correlation Radar. Mémoire de DEA, UVHC, France, Septembre 2006.

10. A low Speed Digital Correlator Architecture Optimized For Resource Savings. *Reconfigurable Communication-centric SoCs (RECOSoC)*, Montpellier France, July 2006.

Table of Contents

List of figures

1

Introduction

Loss of any human life is tragic in any circumstances but the accidental death of a human being in his or her prime age has the most devastating effects on the emotional and social well being of the society. Unfortunately, road accidents are the number one cause of accidental deaths and the majority of their victims are in their prime age. The work presented in this document is a step towards the cause of reducing the number of road accidents and hence minimizing the loss of invaluable human lives. This introductory chapter first highlights the magnitude of this cause and then it outlines the solution we propose to serve the cause.

1. General Introduction

Thousands of people around the world lose their lives to road accidents every year. Thousands others get seriously injured and most of them become disabled for life. A survey showed that 94,000 people died in road accidents in 2006 in USA, Europe and Japan (1). Data (2) for the three years from 2005 to 2007 as illustrated in Figure 1, show that the road accident fatality rates in the major developed countries remain almost constant. It is not hard to imagine that the statistics are even worse in the most populous regions of the world where transportation safety structures are less developed. For example, India has the worst road fatality rate in the world. In 2004 India registered 85000 fatalities in road accidents. In the same year 130,000 people got crippled for life in India due to road accidents (3). The social, economical and psychological repercussions of these accidents are of enormous proportions. The saddest aspect of these fatalities is that most of the victims are less than 40 year old (4). Individuals in this age

group are the most productive members of society who provide, on the average, for five others dependent on them. Hence the loss of one individual in this age group not only causes emotional distress but also economical misery and suffering for those dependent on him or her. As a consequence, the damages ripple through the fabric of the whole socio-economic setup. According to a study, in the US only, the estimated cost of road accidents exceeds US $167 billion every year (5).

Analyses have shown that most of the accidents are caused by the driver's inattention due to physical and mental fatigues. In Europe two thirds of the road accidents happen due to lack of attention on the part of the driver (1). The situation becomes even worse in low visibility conditions due to poor weather or night time driving. Correlations between collisions and driver reactions have shown that a considerable number of accidents can be avoided by recognizing a hazard in sufficient time and making appropriate driving maneuvers (6). It has been shown that if drivers reacted half a second earlier than they normally do, they would avoid approximately half of all accidents (1). The authors in (7) estimated that the crash rate could be reduced by at least 50% with some kind of warning system onboard a vehicle. An analysis (8) showed that between 37% and 74% of the collisions can be avoided by using an obstacle detection system.

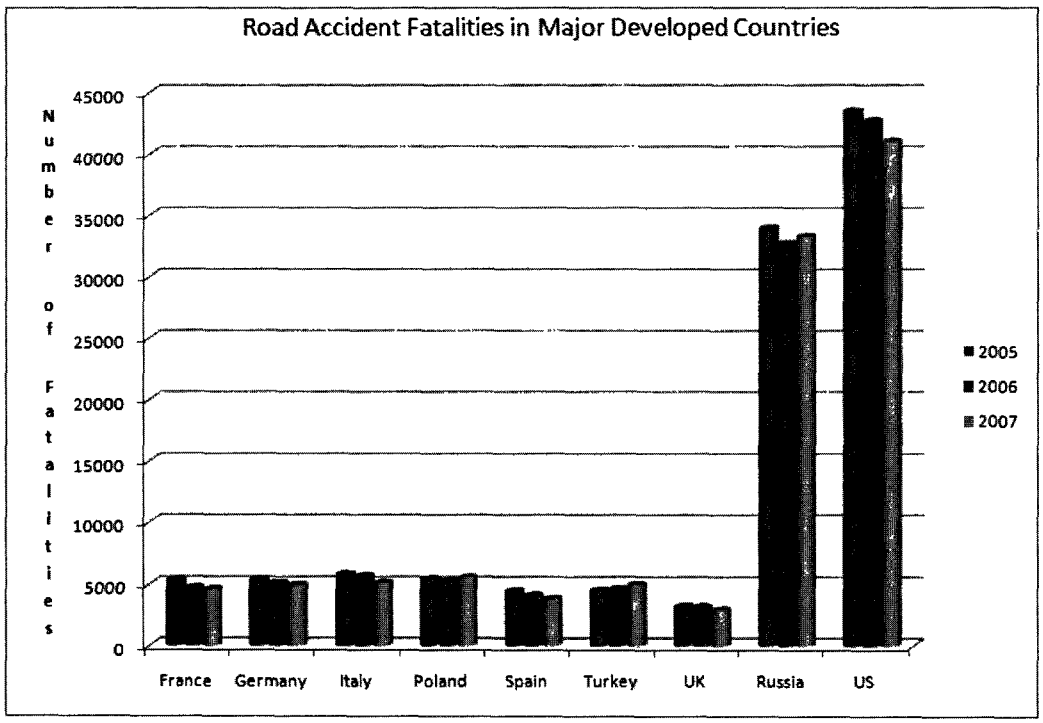


Figure 1: Road Accident Fatalities in Developed Countries --- *Source:* (2)

Such results can be achieved by warning signals to the driver or by automatic control of the vehicle. Electronic systems capable of alerting the driver to impending dangers in a timely manner are called Driver Assist Systems (DASs). These systems alleviate the mental pressures and physical fatigues a driver has to endure in today's driving environment. Reduced mental and physical labor guarantees a more attentive and vigilant driver. Scientific studies confirm that DASs reduce the number mental tension peaks when the driver uses an electronic warning or assistance system. Figure 2 shows the results of a study (9) done by Honda Motor Corporation for the Lane Keep Assist System. When the driver uses the DAS, he or she goes through a reduced number of mental pressure peaks and hence feels more attentive and alert to hazardous situations.

The undisputable advantages of the DASs have convinced the researchers as well as the vehicle manufactures to continue to develop more and more sophisticated systems. Governments around the world are increasingly joining hands with researchers and vehicle manufacturers to find out a way of reducing accidents and to mitigate the effects in cases where accidents cannot be avoided.

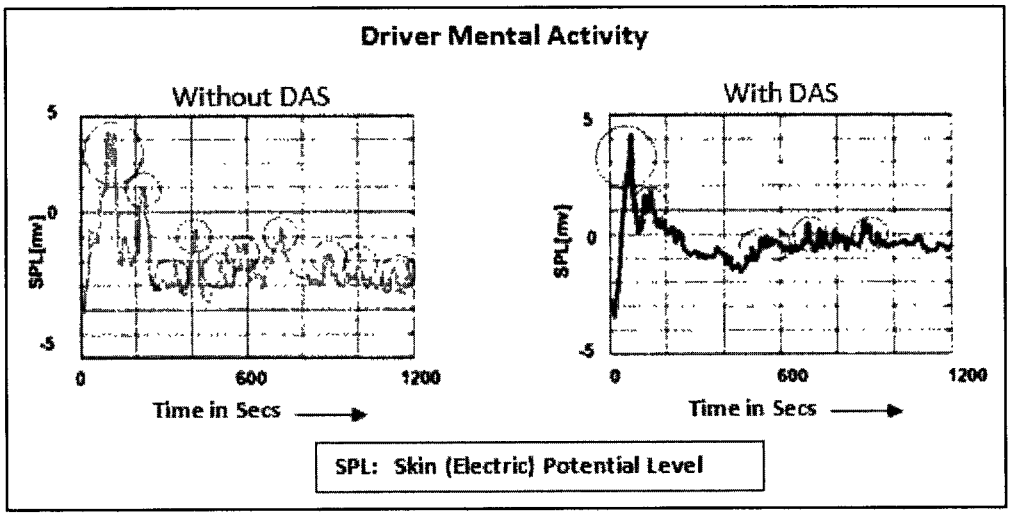


Figure 2: DAS's help reduce Drivers' Mental Stress --- *source* **(9)**

2. Problem Diagnosis and Our Proposed Solution

To suggest an effective solution to a problem, it is of vital to analyze the problem and diagnose its causes. Results of automotive accident profiling reported in (10) and illustrated in

Figure 3, show that frontal collisions account for 66.9 % of all automotive accidents. Hence it can be inferred that avoiding frontal collisions would result in 66.9% reduction in automotive accidents.

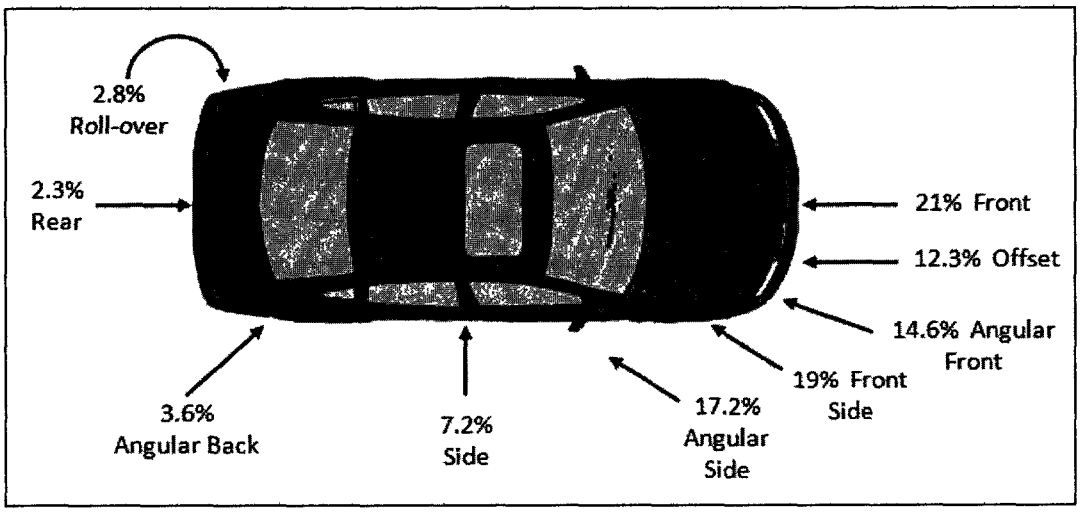


Figure 3: Accident Profile of an Automotive---*source:* **(10)**

It is proven that the stopping distance when braking suddenly, is 22 meters at a speed of 40 kilometers per hour and 44m at a speed of 60 km/h (11). This *Skid-to-Stop* distance increases nonlinearly with the vehicle speed in fair weather conditions. In poor weather the roads become slippery and the stopping distance increases even further. This means that regardless of the level of brake performance, the sensing of danger before the driver can respond is integral to the prevention of accidents.

A radar can sense an obstacle from a considerable distance and it works in poor weather as well as in fair weather. If the obstacle in question is not a stationary one, it is of consequential importance not only to detect it but to know its dynamic behavior and the evolution of its trajectory. Furthermore, to avoid a collision on real roads, multiple obstacles have to be monitored simultaneously. Monitoring the dynamic behavior and trajectory evolution of several objects is termed as *Multiple Target Tracking*.

We propose a DAS based on a Multiple Target Tracking (MTT) algorithm that uses an automotive radar sensor as shown in Figure 4. We use a low-cost radar for obstacle detection and plug it into our MTT system to turn it into a precise and high performance tracker. The system can monitor upto 20 moving or stationary obstacles and generates alert signals for the ones that are dangerous. The radar is mounted on the front side of the host vehicle to detect obstacles 200 meters ahead and within the 12° coverage angle. The MTT system behind the radar tracks these

obstacles in realtime. Thus we target the 66.9% frontal collision zone and generate warning signals for the driver 200 meters before the vehicle arrives at the obstacle. So the driver has more than the half second time cited above (1), to react and take a preventive action.

To be able to rapidly process multiple targets, we use a multiprocessor architecture for the for our MTT system. To make it cost effective, reprogrammable and flexible we implement the system in FPGA using soft-core processors.

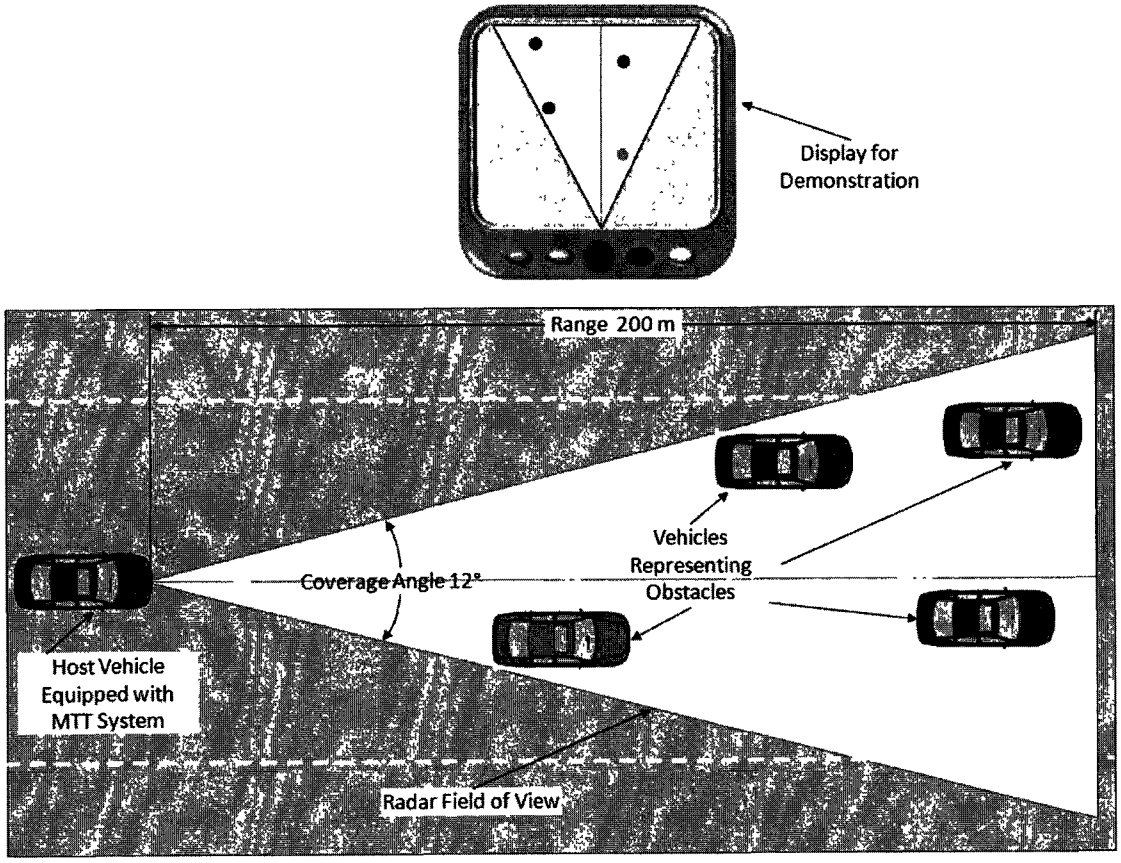


Figure 4: A simplified illustration of our proposed DAS

3. Plan of the Document

After introducing the theme of the subject of the thesis in this chapter, we move on to the internal details of the system in the coming chapters. In Chapter 2 we present available literature

related to our work. We give a critique of the existing solutions and compare and contrast our solution to them. In Chapter 3 we provide details of our MTT application. We explain the application's mathematical model and illustrate the key concepts used in the modeling process. In Chapter 4 we describe the software development model of the application and its mapping to a preliminary hardware architecture. We also introduce the implementation platform and the associated design and development tools in Chapter 4. In Chapter 5 we analyze the performance of the system and lay out a plan for optimizing it. Following the optimization plan, we customize the system architecture to meet the application performance requirements in the best possible way. We end chapter 5 by presenting the finalized optimum system architecture. In Chapter 6 we present the conclusions of the work and provide directions for future work.

2

Related Work and Motivation

In the last several years various kinds of driver assistance systems have emerged. Some solution providers also offer electronic platforms dedicated for developing driver assistance systems. Most of these systems and platforms have limited functionalities and in most cases they are too costly to be deployed on a large scale. If the cost of these systems could somehow be brought down they may be used in conjunction with the solution we propose. Here we present some of these systems and discuss their merits and demerits.

1. Introduction

This chapter describes a review of the existing academic and industrial efforts being done in the fields related to the subject of this thesis. The related work is divided into three categories according to the main themes of the thesis.

In the first category, section 2, we survey some of the general Driver Assistance Systems (DAS) available either commercially or researched academically. In this category we highlight the salient functional characteristics of these systems. We do not emphasize on underlying hardware or software architecture or the implementation details. In this section we also present a short critique of the existing solutions. At the end of the section we discuss our solution based on MTT. After introducing our MTT based solution, we discuss works related to MTT in section 3.

In the second category, section 4, we describe design platforms specifically targeting driver assistance or automotive safety applications. Here we look not only into the functional characteristics but also into the architectural and technological aspects of these systems. We

highlight the pros and cons of these platforms followed by a short description of our proposed architecture.

In the third category, section 5, we describe MPSoC architectures designed for applications other than DASs. The interest here is not the application but the design and implementation of the architectures. Here we discuss the reasons for the popularity of the MPSoCs and we argue why we do not choose fixed MPSoC architectures. At the end of the section we highlight the reasons why we prefer an FPGA platform and soft-core processors for the implementation of MPSoC.

In section 6 we present the hardware / software co-design methodology that we follow for designing our system. We conclude the chapter with a summary in section 7.

2. Driver Assistance Systems

Driver Assistance Systems can be classified into several categories. These categories include the Adaptive Cruise Control (ACC) systems, the Lane Keep Assistance Systems (LKAS), Lane Departure Warning Systems (LDWS), Parking Assistance systems etc. In the following sections we describe the salient features of some of the projects and systems targeting these applications.

2.1. The INTERSAFE Project

The European Project PREVENT (12) was aimed at developing preventive and corrective safety systems for automotives. The effort was to prevent accidents from happening and to minimize the gravity of the consequences when the accident cannot be prevented. It is a grand collaborative project undertaken jointly various European academic and industrial partners. There are several subprojects within the main project. INTERSAFE (13) was established to support the vision of IP PReVENT to create electronic safety zones around vehicles by developing and demonstrating a set of complementary safety functions. It is based on laser scanners, video cameras and bidirectional vehicle-to-infrastructure (V2I) communication. The Laser scanner system obtains a relative position within the intersection by detecting landmarks such as posts and other similar fixed objects next to the intersection, which are registered in a digital map. The video system uses lane markings at the intersection for relative localization which are registered in a digital map. Laser scanners were integrated into both left and right front

corners of the demonstrator vehicle. Thus a combined scan area of 220 degree around the vehicle is achieved.

2.2. Honda HiDS

Honda intelligent Driver Support (HiDS) system comprises of Adaptive Course Control system and Lane Keep Assist System (9). ACC is responsible for controlling vehicle speed and headway distance to a preceding vehicle. The ACC takes into account a Forward Speed of 45-100km/h. The Lane Keep Assist System (LKAS) Assistance provides steering movement to keep the vehicle in the center of its lane. The types of sensors used and the implementation details are not known.

2.3. SEAT ADAS

SEAT exhibited a prototype of their system called Alhambra ADAS (Advanced Driver Assistance Systems) at the 10th World Conference on Intelligent Transport Systems and Services held in Madrid in 2003 (14). The main features are an adaptive speed regulating ACC, excessive speed warning when approaching traffic signals and road markings recognition. The vehicle is also equipped with "intelligent" headlights, which regulate the adaptive lights beam intensity according to road conditions. The ACC system automatically adjusts the cruise speed of the vehicle. It detects other vehicles, using sensors mounted in the headlights, instead of the usual radar, and maintains a safe distance by activating controls brake and accelerator.

This prototype includes the Stop & Go function, thus complementing the ACC system at low speeds (0 to 40 km / h), and automatically stopping the vehicle. Another feature is the detection of traffic signals by radio frequency. The vehicle receives information about the speed limit, warns the driver and adjusts the car speed. A fourth system "reads" the road markings using a camera. An electronic unit calculates the distance between the car and the road signs and sound alert or vibration of the steering wheel is generated to warn the driver of a danger. This performance is achieved with the help of a prediction system using a database complementary to the navigation system, including a series of parameters such as latitude and longitude of several reference points like the sidewalk or the radius of curvature, and dynamic data of the vehicle. The resulting information help regulate the speed with respect to the vehicle trajectory.

2.4. The SARI / RADARR Project

The project RADARR (Recherche des Attributs pour le Diagnostic Avancé des Ruptures de la Route) is a subproject of the SARI (Surveillance Automatisée de la Route pour l'Information des conducteurs) (15) program supported by the French government in the framework of the PREDIT initiative for land transport safety. The objective of this study is to design an information system alerting drivers of a potential loss of control of their vehicle. The risk is considered linked to a physical disruption of the route in open country. One of the aspects of the project is to identify and quantify the trajectory limits on roads for vehicles which are then graded hierarchically according to the level of their dangerousness. A laser rangefinder is used for identifying the vehicle trajectory limits. The risk is assessed and then used to define a typology of messages or signals intended to get the driver's attention.

Another objective of the project is to measure the trajectory of vehicles turning around a curve in the road. The measurement system is based on three digital cameras covering the vehicle from three angles: from the front, from the back and from the top. The system is equipped with a laser rangefinder that provides information on a semi-plane of the scene. Once the effectiveness of the system put is proven, it will be evaluated for deployment.

2.5. The CHAMELEON project

The CHAMELEON (16) was another multi-partite project supported by the European Union from 2000 to 2002. The main objectives of the project were the development of pre-crash sensorial system for impending crash detection. The work was geared more towards the mitigation of the severity of the accident in case of an impending crash. They focused on the improvement of the sensor technologies and on the research in object classification and sensor fusion techniques. The sensors being researched were contributed by collaborating industrial partners and included laser scanners and microwave radars.

Various crash scenarios were simulated in the lab and their effects were analyzed to determine the number and types of the sensors and their respective technical characteristics necessary for minimizing the gravity of a crash.

2.6. Other less known DAS's

Research on sensor fusion is also reported in (15). Here the sensors under consideration are video cameras and laser scanners. The research focuses on single object tracking with a laser scanner and pattern matching for lane and object recognition.

A concept of intelligent navigator is proposed in (17). From both the current traffic condition obtained from visual data and the driver's goal and preference in driving, it autonomously generates advice to the driver. These advices include safety related and tactical maneuvers such as emergency braking due to an abrupt deceleration of the front vehicle, lane changing due to the congested situation etc. A Three-level reasoning architecture is proposed for generating advice in dynamic and uncertain traffic environments.

An active vision system for realtime traffic sign recognition system is presented in (18). The system is composed of two cameras; one is equipped with a wide-angle lens and the other with a telephoto lens, and a PC with an image processing board. The system first detects candidates for traffic signs in the wide-angle image using color, intensity, and shape information. For each candidate, the telephoto camera is directed to its predicted position to capture the candidate in a larger size in the image. The recognition algorithm is designed by intensively using built-in functions of an off-the-shelf image processing board to realize both easy implementation and fast recognition.

2.7. Critique of the Presented DASs

The DAS presented above are effective but there are some practical issues that have to be addressed. For example, the INTERSAFE (13) is limited to road crossings only. Secondly, the laser scanners and cameras rely on the land marks in a digital map. The land marks are very likely to change over time and the digital map must be updated on all the systems in the field. The use of cameras for road mark identification has its own limitations. For example they are ineffective in poor weather conditions. Similarly the range of cameras is very short implying that for identifying an object; the host vehicle must get close to the object. This can be highly dangerous when the host vehicle is running at high speed or when the road is slippery due to rain or snow. Most of all this solution relies on an infrastructure that must be placed at every road crossing. This, obviously, is a very costly proposition due to its initial fixed cost and the recurring maintenance cost.

The Honda HiDS (9) sounds interesting but since the technical details are not known, it is hard to evaluate it. Moreover systems like HiDS are not only proprietary and limited to the specific vehicle models, but they are also very costly often more than or comparable to the cost of the vehicle itself. Hence driver of a vehicle different than the specific model cannot use these system systems while the costs of the specific equipped models are not in the budgetary range of everybody.

The system proposed by SEAT (14) works more or less on the same lines as the INTERSAFE (13) with the addition of the ACC. So it has all its limitations discussed except that the SEAT system can keep a safe distance from an obstacle. Furthermore being limited to specific high-end SEAT models, it has the same disadvantages as the Honad HiDS.

The SARI/RADARR (15) project seems to be concerned more with road profiling than developing an onboard safety mechanisms. It does propose the generation of a warning signal to the driver when the driver departs from the predefined safe trajectory or when the vehicle approaches a curve on a pre-charted road. What happens when the road is not already charted or when the drivers stays on the defined trajectory but there is an obstacle ahead, is not considered. In a way it is dependent on it is dependent external on actors like the INTERSAFE (13).

The CHAMELEON (16) project is concerned with damage control rather than damage prevention. It discusses the safety mechanisms and sensors which may minimize the damage when the crash is inevitable. Preventing the crash from happening is not among the objectives of the project. Solution like these can be used to complement crash prevention systems so that the damage can be reduced when it is not possible to avoid the crash.

Most of the other less know systems are almost exclusively using cameras and vision systems to alert the driver to a potentially critical situation. As discussed above, camera based systems are not effective in all conditions.

2.8. Our Proposal

We propose a DAS based on a Multiple Target Tracking (MTT) algorithm that uses automotive radar as a front end sensor. An MTT system monitors the dynamic behavior of several obstacles at a time. In the context of Driver Assistance Systems, an MTT system detects obstacles in front of the host vehicle and monitors their distance, speed and trajectory. If the behavior of any of the obstacle fulfills preset alert conditions, the driver of the host vehicles is alerted in advance to deal with any dangerous situation. In case the vehicle is fully equipped for autonomous driving, the signals generated by the MTT can also be used to automatically control the vehicle if necessary. The alert signals generated by the MTT system can also be used to

trigger the onboard safety systems if the obstacle's behavior is rated above a predefined danger level. So it also can incorporate the pre-crash safety mechanism when the crash cannot be avoided at all.

We use a low-cost radar for obstacle detection and plug it into our MTT system to turn it into a precise and high performance tracker. The tracking algorithm helps differentiate between real danger and false alarms, so that the driver is not panicked by triggering alert signals unnecessarily. Another advantage of the tracking algorithm is that can cancel the interference for other similar or dissimilar systems. The system we propose can monitor upto 20 moving or stationary obstacles and generates alert signals for only the ones that are really dangerous.

The use of radar as sensor in our system has the advantages of longer range as compared to camera based systems. It performs better in bad visibility conditions and has lower computational requirements (19). It is an all weather system that works as efficiently in a stormy dark night as in a sunny bright day. Moreover, since radar helps detect obstacles at longer distances, it ensures longer time for vehicle drivers to react to a dangerous situation.

Our system is applicable on highways with sparse high speed traffic as well as on the urban roads with dense low speed traffic. It does not rely on any infrastructure or digital maps.

We propose a plug and play system so that it is not limited to any specific vehicle manufacturer or a vehicle model. The low cost of the system makes it accessible to every vehicle driver.

3. Work Related to Multiple Target Tracking

Studies have been done on the isolated parts of MTT system (20), (21) but, to the best of our knowledge, design of the complete MTT based DAS has not been addressed in full before. 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 which is a part of MTT, is proposed in (19). It is not only limited to the filter but it also is a fully hardware implementation. Fully hardwired 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 (22). In addition to being a completely hardwired implementation, the work presented here is inconclusive.

The data association aspect of MTT has been dealt with nicely in (21) 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 (23) 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 (24). 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 (23) and Modified Faddeev based implementation of the Kalman filter (24) 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 where roads curve and bend frequently. Similarly, other algorithms like (24) 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.

4. Platforms for Automotive Applications

Some of the vendors provide electronic development platforms for driver assistance systems. These platforms include programmable processors, heterogeneous multiprocessor

systems and microcontrollers dedicated to a certain type of driver assistance application. In the following sections we describe some of the available platforms.

4.1. IMAPCAR

NEC upgraded the IMAP-VISION processor to IMAP-CE, which was unveiled at the ISSCC held in 2003 in the United States. The processor was renamed IMAPCAR (Integrated Memory Array Processor for CARs) (25) in 2006. It is used by Toyota Lexus in their cars (26) as a safety system.

IMAPCAR uses an SIMD system for which 128-parallel processing units follow identical commands and a 4-way VLIW system capable of simultaneous execution of four commands in one cycle. Each processing element has a RISC architecture with a 24-bit multiply and accumulate unit is equipped with 2 Kbyte SRAM for unit to enhance execution performance. The processor elements are interconnected via a shift register style ring network. A single 16-bit RISC control processor with 32KB program and 2KB data caches is used to control the 128 processing elements.

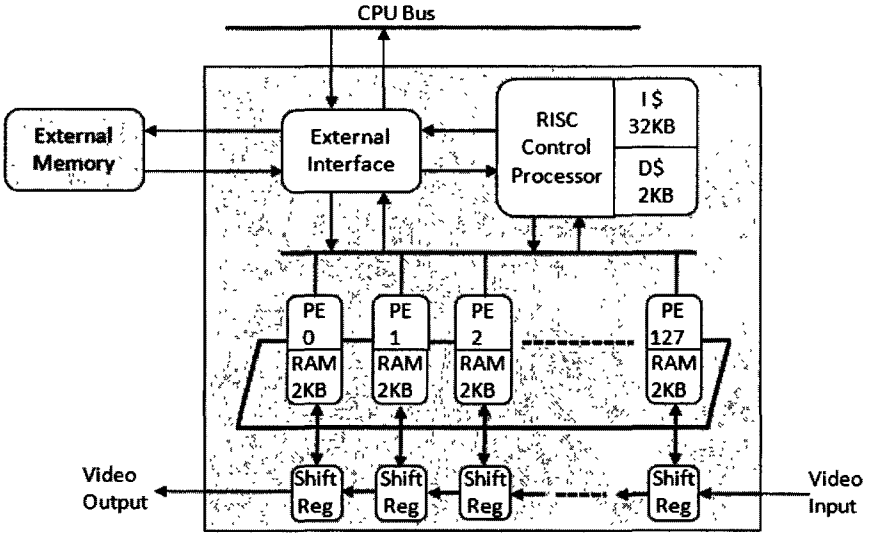


Figure 5: The IMAPCAR Architecture

In December 2008 (27) NEC announced the second generation IMAPCAR2 processors with support for both SIMD and MIMD operations. The 128 processing engines now support 16bits rather than 8-bits supported the earlier SIMD-only IMAPCAR.

The IMAPCAR processor is primarily aimed at the image processing requirements of the automotive safety systems. An image is loaded column wise into the 128 local memories. A processing element has therefore direct access to all pixels in a column of the picture. One of the shortcomings of the IMAPCAR design is that it is not easy to exploit the task level parallelism found in high level image processing tasks.

4.2. EyeQ2

The EyeQ2 is a joint venture by Mobileye and STMicroelectronics (1). It uses two floating point MIPS32 34Kf processor cores. The two MIPS cores exchange data using the ITU (Inter Thread Communication Unit). Besides the two MIPS cores the EyeQ2 includes seven vision computing engines, and a 16-channel direct-memory-access (DMA) controller.

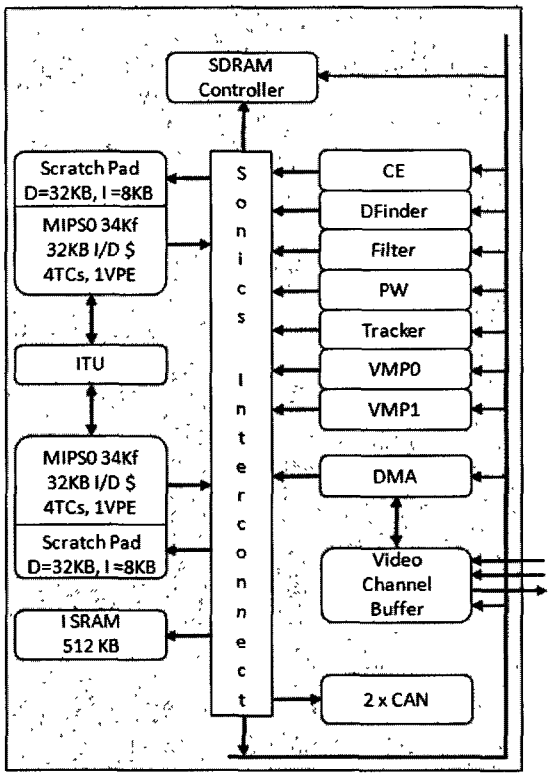


Figure 6: The EyeQ2 Architecture

The vision computing engines are fixed logic processing elements or hardware accelerators used for image pre-processing, object classification etc. The vision computing engines available in the EyeQ2 are CE (Classifier Engine), DFinder (Disparity Finder) which is used for stereo vision, Filter, PW (Preprocessor Window), Tracker which is used for motion analysis and two Vector Microcode Processors (VMPs) which utilize parallel vector, scalar and table lookup units.

The two MIPS cores and the seven vision computing engines are connected with an interconnection network from Sonics called SMX (Sonics Multi-service eXchange). The engines and CPU logic perform all of the vision computations required by applications such as pattern matching and image classification. The system interfaces to the outside world through two CAN controllers.

4.3. VIP-II

The VIP-II (Vision Instruction Processor version 2) is the successor of the VIP-I developed by Infineon (28).

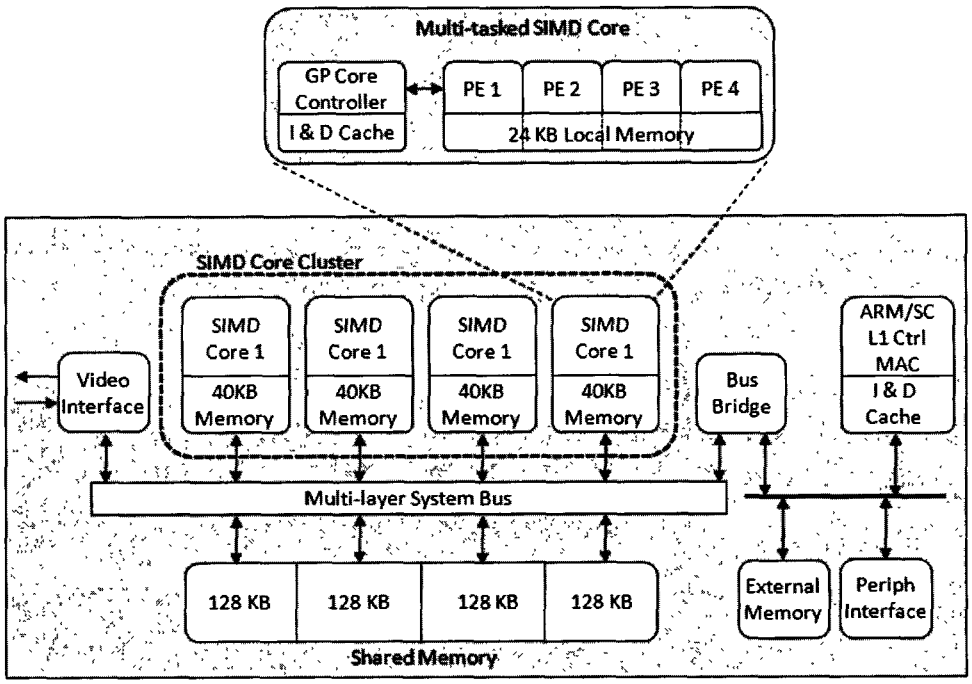


Figure 7: The VIP-II Architecture

The processor is designed for vision based automotive safety applications. The VIP-II features 4 multi-tasked SIMD cores and an ARM 9 processor. Each SIMD core consists of a general-purpose core and four PE (processing element) arrays. The cores use VLIW instructions to perform arithmetic operations and memory access in parallel. Each processing element has 4-stage pipeline. Data dependencies between the pipeline stages are avoided because each of the pipeline stages operates on an instruction from a different thread. To make this possible each processing element is provided with four instruction caches, four register files and four program counters.

Every SIMD core is controlled by a general purpose core. The four cores are connected via a multi-layer system bus. An additional general purpose processor (ARM9) handles the communication tasks and main control flow. The four general purpose controllers within the SIMD cores and the ARM9 are all programmed in C. To program the SIMD cores a C language extension, called DPCE (Data Parallel C Extension), is used.

4.4. MPC5561 Microcontroller

Designed by Freescale Semiconductor (29), the MPC5561 MCU is a member of the MPC5500 family of microcontrollers. It features a FlexRay network controller and Freescale's e200 core, which is customized for automotive safety applications. The e200 core has a 32-bit PowerPC architecture with additional signal processing instructions. It has a 32KB unified cache, a Memory Management Unit (MMU) and a 32-channel DMA. It has interfaces for 192KB SRAM and 1MB Flash memory apart from the conventional microcontroller peripherals like timers, watchdog etc. The FlexRay is a communications system designed to provide distributed control for automotive applications. It has a dual-channel architecture for redundancy for the reliability requirements of safety systems. The FlexRay is yet another networking scheme for automotive applications. The other well known schemes are CAN (Controller Area Network) and LIN (Local Interconnect Network).

The e200 core is connected to the memories, the DMA controller and the Flexray network controller through a crossbar switch. Two bridges interface the external peripherals to the crossbar.

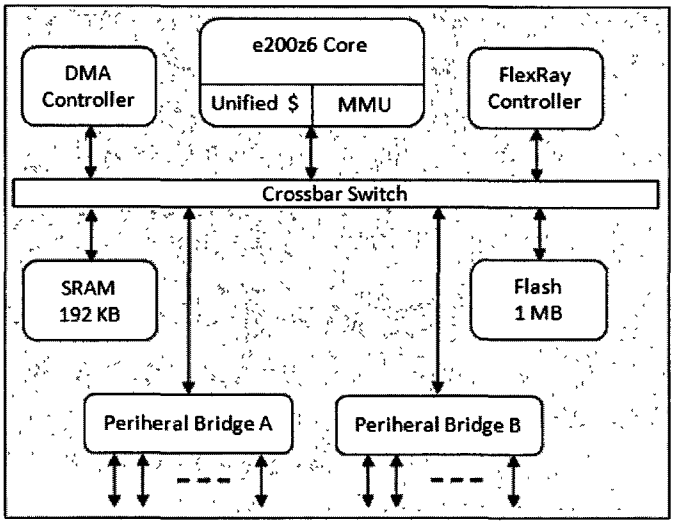


Figure 8: The MPC5561 Architecture

4.5. TMS570F

In November 2008 Texas Instruments announced the TMS570F (30) microcontroller unit (MCU). It is claimed to be industry's first dual core Cortex-R4F processor-based floating-point MCU that allows automotive system designers to implement both single and double precision floating point math depending on performance requirements. According to the report it uses accelerated multiply, divide and square root functions to improve system performance. The TMS570F MCU platform uses two identical ARM Cortex-R4F processors combined with an initial two Mbytes on-chip flash memory. Industry standard peripherals include FlexRay protocol controller, up to three CAN and two LIN modules along with TI's timer co-processor and two 12-bit analog to digital converters (ADC). Targeted applications include chassis control, braking, electronic vehicle stability and steering and airbag electronics etc. Architectural details of the MCU are not provided by TI.

4.6. Critique

The first three of the processors described above are solely dedicated to vision based safety and assistance systems. As mentioned earlier, vision based systems are only fair-weather

and short range systems. Their performance degrades considerably in poor weather and long ranges.

The IMAPCAR (25), (27) does not support task level parallelism exploitable in most of the DAS applications. It is provided to vehicle manufactures only hence everyone cannot benefit from it. It is programmed in a C language extension specially developed for this architecture, which is called 1DC (one dimensional C). This is another disadvantage which limits the designers to the single proprietary development environment and increases the cost of the already costly system even further. Furthermore, the architecture is not reconfigurable and hence inflexible and un-scalable.

The EyeQ2 (1) also has the same restrictions as the IMAPCAR. The architecture is mainly designed for pixel level parallel processing of the images captured by the camera. Furthermore, the platform is provided to vehicle manufacturers exclusively. This makes its application highly restricted and out of the reach of the everyday vehicle users.

The vision computing engines are fixed in hardware and carry out specific image processing tasks. To scale the system for future evolutions would necessitate complete redesign of the architecture. The processor is provided only to the vehicle manufacturers like Volvo and BMW who use them in the expensive high-end vehicles.

To program the SIMD cores in the VIP-II (28), a C language extension, called DPCE (Data Parallel C Extension) is used which makes it costly and difficult to program. Thus it can be afforded by only a minority of the drivers who can afford such costly vehicles. To help reduce the high accident rates the DAS's must be economical enough to be within the reach of all the vehicle owners.

The latter two of the processors discussed above are microcontrollers targeting automotive safety measures in pre-crash situations. Such systems can complement driver assistance systems rather than replace them. The MPC5561microcontroller (29) is meant for controlling various automotive safety mechanisms and interconnecting intelligent devices onboard a vehicle. It is not a driver assistance system per say, rather it can be considered as a pre-crash damage mitigation system. The TMS570F (30) also falls in the same category as the MPC5561. It also concentrates on protective measures in case of an accident. Therefore these systems can be used in conjunction with sophisticated DAS's for assuring security of the passengers in cases where accidents cannot be avoided.

4.7. Our Proposal

Our work is unique in several aspects. We propose radar based DAS implemented as an MPSoC. We customize each of the processors according the needs of the application task it is

running. It makes the system fast, small sized and energy efficient. The individual processor can be separately programmed allowing for in system upgrading and improvement. The system designer can replace one algorithm with another for a specific task to make the system perform more efficiently and/or more accurately. The programming is done plain ASCII C, so tweaking the existing application or adding more functionality does not require a specialist in platform specific languages. 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 (26). Our reconfigurable MPSoC architecture of the system is inherently flexible, programmable and scalable. Adding additional processing elements or auxiliary hardware components does not affect the working of the existing architecture. 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 any vehicle.

5. MPSoC Architectures for Other Applications

Several multiprocessor systems have been designed to target applications other than the driver assistance systems. Although the applications, for which these systems are intended, are different than our application, their architectures are of interest to us. The systems presented in the following sections have multiprocessor architectures, an attribute they share with our work.

5.1. C-5 NP

The C-5 NP is specifically designed by Freescale Semiconductor Incorporation (31), for communications applications. It deals with the networking tasks like packet processing, table lookup processing, and queue management. The C-5 NP contains 18 processors (16CPs, XP, and FP) and three coprocessors that operate as shared resources for the CPs and each other, and perform networking-specific tasks. The programmable Channel Processors (CPs) are responsible for receiving, processing, and transmitting cells or packets. The Executive Processor (XP) provides network control and management functions in user applications. The Fabric Processor

(FP) manages the high-speed fabric interface. The Buffer Management Unit (BMU) manages centralized payload storage during the forwarding process. The Table Lookup Unit (TLU) provides table search and associated data storage services to the CPs, XP, and FP. The Queue Management Unit (QMU) manages application-defined descriptor queues among the CPs, FP, and the XP.

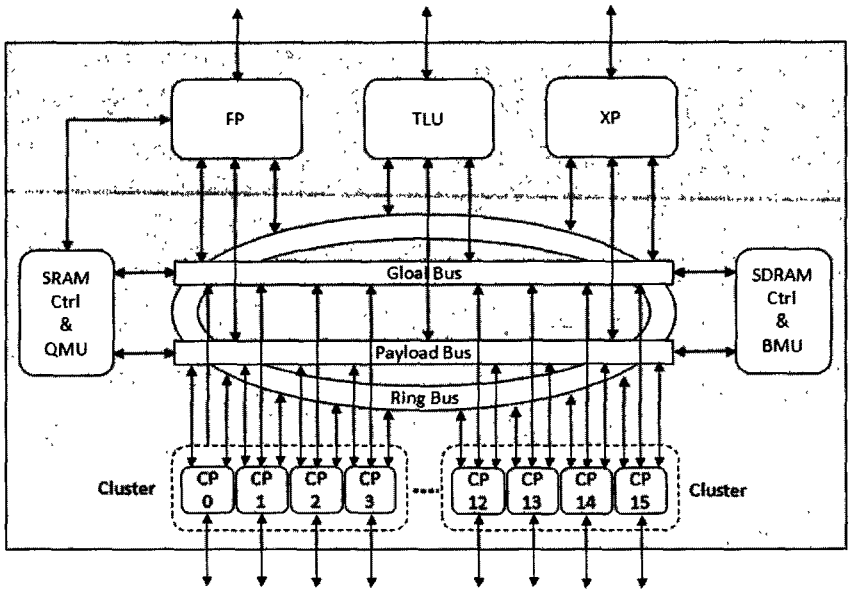


Figure 9: C-5 NP Architecture

The C-5 NP also contains three independent data buses that provide internal communication paths between the eighteen processors (16CPs, XP, and FP) and the three coprocessors, supporting concurrent processing. The Payload Bus is a slotted, multichannel shared, arbitrated bus which carries payload data and payload descriptors between the processors and the BMU and QMU. The Ring Bus provides bounded latency transactions between the processors and the TLU. It also supports inter-processor communication. The Global Bus is also a slotted, multichannel, shared, arbitrated bus Supports inter-processor communication via a conventional flat memory-mapped addressing scheme.

5.2. Viper Nexperia

The Philips Viper Nexperia (32) is an MPSoC designed for multimedia applications. Its architecture includes two CPUs: a MIPS (R3940) and a Trimedia (TM32) VLIW processor. The MIPS acts as a master running the operating system, whereas the Trimedia runs video processing functions and acts as a slave that carries out commands from the MIPS. The system includes three buses, one for each CPU and one for the external memory interface. A 64-bit memory bus is used by the MIPS and TM32 CPUs and on-chip blocks requiring memory access. Two PI buses and a crossover PI-to-PI or memory-mapped I/O (MMIO) bridge enable each processor to control or observe peripheral block status. Hardware accelerators for image composition, scaling, MPEG-2 decoding and video input processing are also attached to the buses. The Viper can implement a number of different mappings of physical memory to address spaces. Programmable CPU cores allow new features, services, or standards to be supported through software upgrades without changing silicon.

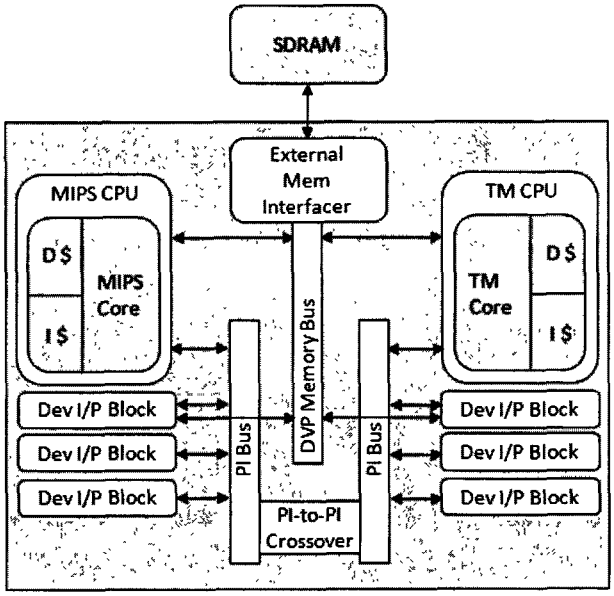


Figure 10: The Viper Nexperia Architecture

5.3. OMAP

The Texas Instruments OMAP (Open Multimedia Applications Platform) (33) is designed for mobile phone wireless and multimedia applications. OMAP comes in many flavors.

The OMAP5912 architecture contains a TMS320C55x DSP core from TI and an ARM925T CPU. The ARM acts as a master, and the DSP acts as a slave. The ARM is used for the operating system, user interface, and OS applications. The DSP is used for signal processing applications, such as MPEG4 video, speech recognition, and audio playback. Both processors utilize an instruction cache and a memory management unit (MMU) each, for virtual-to-physical memory translation and task-to-task memory protection.

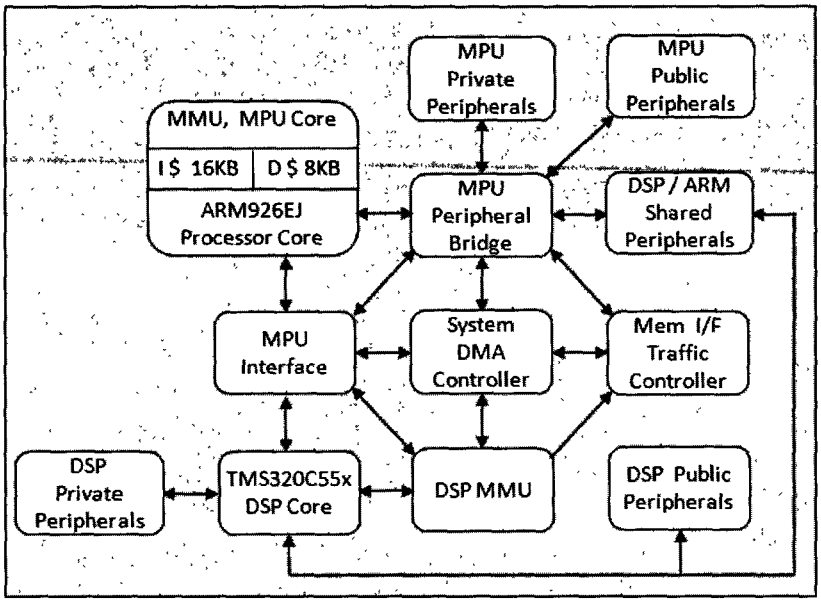


Figure 11: The OMAP Architecture

5.4. ARM MPCore

The ARM MPCore (34) is a homogeneous multiprocessor that also allows some heterogeneous configurations. The architecture can accommodate up to four CPUs. Both the data and instruction caches can be sized between 16KB and 64KB for each processor. The caches snoop for consistency. The interconnection fabric can be configured either as dual or single 64-bit AMBA 3 AXI bus. Each processor can also be configured with an optional Vector Floating Point (VFP) unit.

Some degree of irregularity is afforded by the memory controller, which can be configured to offer varying degrees of access to different parts of memory for different CPUs. For example, one CPU may be able only to read one part of the memory space, whereas another part of the memory space may not be accessible to some CPUs.

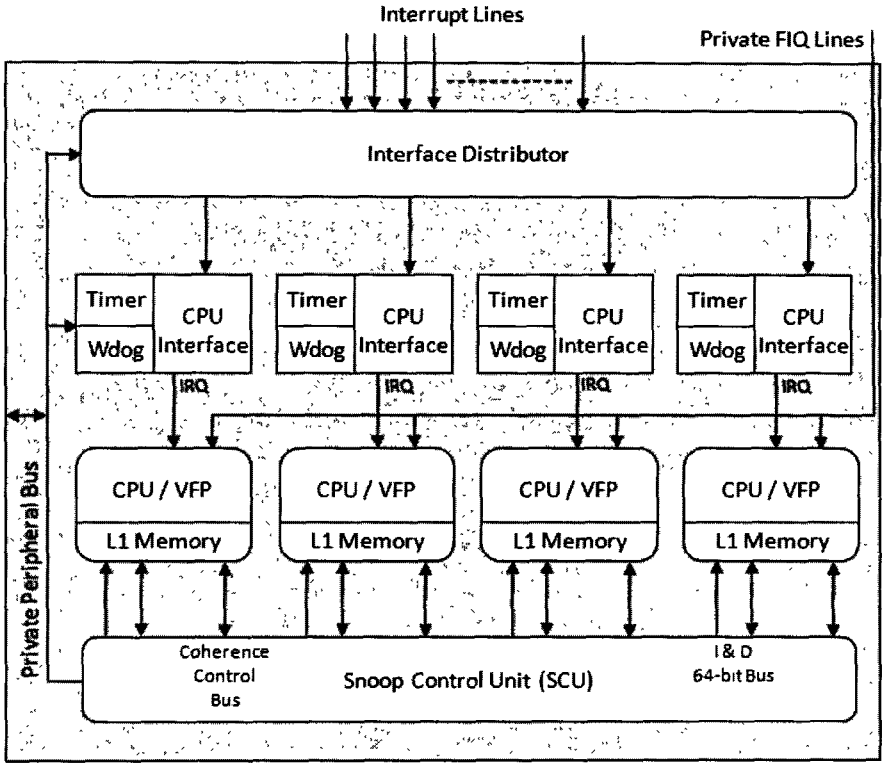


Figure 12: The ARM MPCore Architecture

5.5. The Cell processor

The Cell architecture (35) is designed by Sony, Toshiba and IBM for the PlayStation 3. It is comprised of hardware and software "cells". The software cells consist of data and programs (known as jobs or apulets), these are sent out to the hardware cells where they are executed.

The architcture contains a main core called Power Processing Element (PPE) and 8 special cores called Synergistic Processing Elements (SPE). The PPE is a classic 64 bit PowerPC processor with 512K cache. The processing elements, PowerPC, and the I/O interfaces are connected by the Element Interconnect Bus (EIB), which is built from four 16-B-wide rings. Two rings run clockwise, and the other two run counterclockwise. Each ring can handle up to three non-overlapping data transfers at a time.

The PPE runs the operating system and most of the applications but compute intensive parts of the OS and applications are offloaded to the SPEs. An SPE is a self contained vector processor which acts as an independent processor. Every SPE contains 128 x 128 bit registers, four (single precision) floating point units and four Integer units. The SPEs also include a small 256 Kilobyte local store (LS) instead of a cache. The SPEs have very fast access to their LS but to access the main memory they must request the interconnection bus for asynchronous transfers. Each core can be explicitly programmed with independent threads. The memory is shared and the user has to manage data copying among the cores. Like the PPE the SPEs are in-order processors and have no Out-Of-Order capabilities.

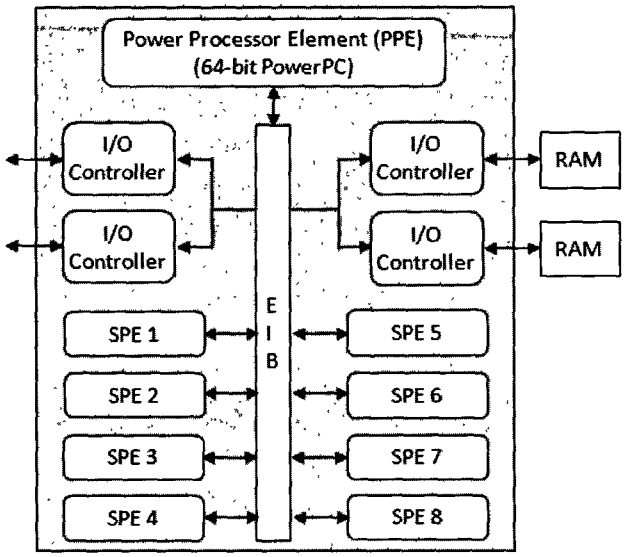


Figure 13: The Cell Architecture

5.6. Discussion and critique

The systems discussed above and many other emerging application specific systems demonstrate the popularity of the multiprocessor architectures tailor made for the applications they are running. This ever growing popularity of the MPSoCs is caused by some serious limitations uniprocessor systems have.

Uniprocessor architectures have hit the wall with respect to performance and power consumption in the past decade. They, even with the highest clock speed, cannot keep up with the real-time processing requirements of the modern day embedded applications. Real-time systems require "real" parallelism and concurrency in the execution of applications (36).

Uniprocessor architectures provide "apparent" concurrency through the use of Multi-tasking operating systems but the applications are still running sequentially on the underlying hardware. Consequently they are too slow for today's complex real-time embedded applications. In reaction to these issues, Multiprocessor System-on-Chip (MPSoCs) architectures have emerged as an alternative means for achieving higher performance.

An MPSoC uses multiple processing cores operating in parallel performing various tasks to execute a complex global application efficiently and rapidly. MPSoCs provide real concurrency by segregating tasks and running them in parallel to improve predictability and performance. Energy consumption can be efficiently managed in MPSoCs by allowing processors to idle when their tasks are finished.

Heterogeneity is one of the greatest advantages of these systems because it improves realtime performance and predictability. For example, some operations in an application are standardized and can be implemented in conventional ways. However there are many specialized operations which need specialized units for higher throughput. Specialized units are tailor made according to the performance of the specific operations and hence are power efficient and their behavior is more predictable compared to a general purpose processor.

Memory subsystems for MPSoCs are custom built for the requirements of the tasks the processors are executing. They combine off-chip bulk memory with on-chip specialized memory. This not only improves performance but the memory traffic can also be easily regulated to further reduce energy consumption.

A multi-core device, which combines two or more processors on a single die, offers increased performance over single-core devices. In comparison to a traditional processor, dual-core system offers at least double the performance at the same clock frequency. Tests on a dual-core system reported in (28), have shown that the same performance can be achieved at 200 MHz as a single-core system operating at 500 MHz.

An important side benefit of the improved performance is that the power consumption and heat generation of a multi-core device are lower for the same level of performance as a single-core device. In addition, the faster clock of the single-core device requires faster memory, which further increases the power consumption and requires special packaging to dissipate the heat.

5.7. Our proposal

While the advantages of the multiprocessor architectures are undisputable, there are some very fundamental decisions that must be taken before initiating an application specific

multiprocessor project. The implementation platform, the types of processors, the memory hierarchy, the interconnection framework and other components have to be determined very carefully according the needs of the application.

We use FPGAs as the implementation platform for our MPSoC. FPGAs provide enormous raw processing power compared with standard microprocessors. They give the designer the choice to run applications in hardware and in software when configured with a processor IP core. Thus FPGAs provide the flexibility of reconfiguration and the liberty of reprogramming. These are two highly sought after features needed for system evolution that neither general purpose microprocessors nor ASICs can provide. General purpose microprocessors can be reprogrammed for a desired application but their hardware architecture cannot be modified. Moreover they are slow because of the sequential execution of the code. On the other hand ASICs are fast but their architecture is cast in concrete for one specific application hence they are absolutely inflexible. FPGAs combine the best of the two worlds. They can be configured with soft-core processors which can execute a software application and thus offer the versatility of a general purpose processor. They can also be configured with hardwired circuitry and thus provide high speed execution where needed. FPGAs can even have a mix of programmable soft-core processors and hardwired circuitry to accelerate certain parts of the application for achieving higher processing speed.

System performance can be easily scaled at any phase of the design cycle by adding processors, custom instructions, hardware accelerators, and by leveraging the inherent parallelism of FPGAs (37). FPGAs are highly receptive to IP reuse and support designing and verifying new customized IPs. FPGA designs using pre-verified IPs reduce system design and verification time. FPGA designs do not have the up-front Nonrecurring Engineering (NRE) costs which is characteristic of ASIC designs.

The choice of the FPGA platform allows us to use soft-core processors and IPs in our design. An FPGA designer can configure a group of programmable logic blocks to act as a processor. These are typically called "soft core" processors (38). All of the peripheral devices such as counter timers, interrupt controllers, memory controllers, communication functions and etc., are also implemented as soft cores in the FPGA logic blocks.

Using soft-core processors in an FPGA based MPSoC architecture has numerous advantages. Soft core processors can be instantiated as many times as the designer requires as long as there are enough FPGA resources available. Higher levels of overall application performance can be readily achieved by instantiating multiple soft-core processors. Moreover, when soft-core processors are used in the architecture of an MPSoC, their hardware features can be customized according to the requirements of the application. For example an FPGA designer can customize his/her soft-core processor configuration by sizing, including or excluding certain features of the processor according to the performance needs of the application. Soft-core

processors support generation-time configuration options to allow designers to trade off performance and cost. Examples of generation-time configuration options include pipeline implementation, cache size, multiplier implementation, divider implementation, barrel shifter implementation, and tightly coupled memories etc. The designer can include or exclude these features at generation time and thus optimize the architecture. A soft-core processor can also be complemented by a hardware accelerators configured out of the FPGA logic blocks.

6. The Hardware / Software Co-design Flow

We follow the Y-chart hardware/software co-design flow (43) for the design of our MTT implementation in FPGA based MPSoC. The Y-chart co-design flow is illustrated in Figure 14. According to the Y-chart approach, an application model, derived from a target application domain, describes the functional behavior of an application in an architecture-independent manner.

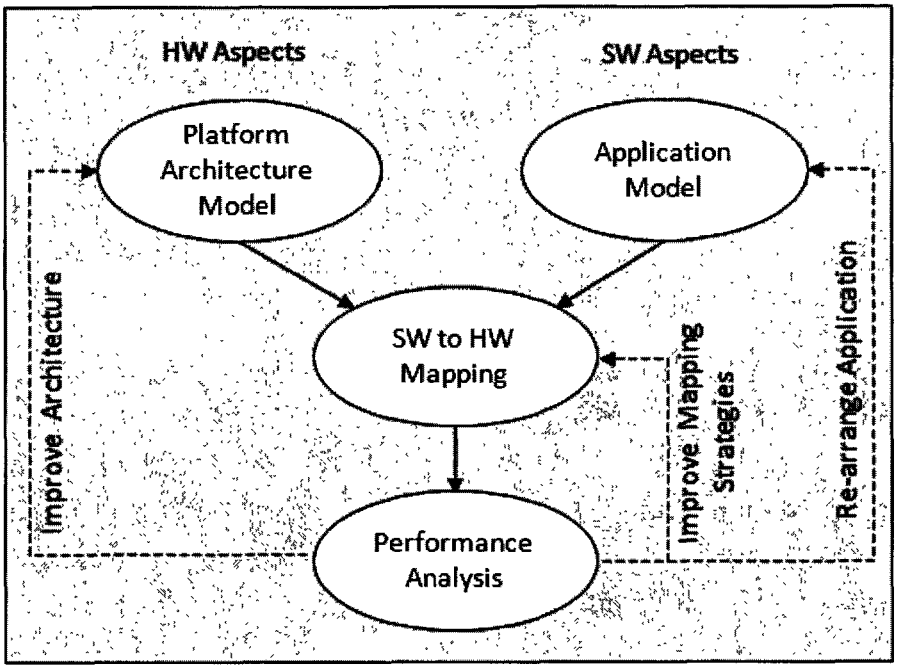


Figure 14: The HW/SW Co-design Flow

The application model is often used to study a target application and obtain rough estimations of its performance needs, for example to identify computationally expensive tasks. This model correctly expresses the functional behavior, but is free from architectural issues, such as timing characteristics, resource utilization or bandwidth constraints. Next, a platform architecture model, defined with the application domain in mind, defines architecture resources and captures their performance constraints. Finally, an explicit mapping step maps an application model onto an architecture model for co-simulation, after which the system performance can be evaluated quantitatively. The dotted lines in Figure 14 indicate that the performance results may inspire the system designer to improve the architecture, modify the application, or change the projected mapping. Hence, Y-chart modeling methodology relies on independent application and architecture models in order to promote reuse of both simulation models to the conceivable largest extent.

This methodology forms the outline of the approach we use for designing and optimizing our system. We will replicate Figure 14 in the beginning of the coming chapters with the irrelevant parts dimmed off, to emphasize only on the steps that are the subject of the chapter.

7. Chapter Summary

In this chapter we presented a survey of the existing DAS solutions and compared and contrasted our proposed solution with them. Processors designed for DASs and automotive safety applications were discussed next, followed by a brief account of the MPSoC architecture we propose. We described some of the existing MPSoC architectures followed by the motivations behind our choice of the FPGA platform and soft-core processors for our system. In the end we presented an outline of the Y-chart HW/SW co-design flow that we follow in coming chapters.

3

Multiple Target Tracking Application Modeling for Automotive Safety

Use of target tracking in general and multiple-target tracking in particular has been traditionally restricted to military applications. The peculiar conditions where the military MTT systems are supposed to operate cannot be generalized for use in automotive applications. An MTT system meant for automotive driver assistance applications, must take into account the specific dynamics of the obstacles encountered on roads. In this chapter we describe the design of our own MTT application and its mathematical model specifically tailored to the requirements of automotive safety use.

1. Introduction

This chapter describes the design of the MTT application we propose for our driver assistance system. The proposed MTT based system monitors the dynamic behavior of several obstacles at a time. It detects obstacles in front of the host vehicle and monitors their distance, speed and trajectory. If the behavior of any of the obstacle fulfills preset alert conditions, the driver of the host vehicles is alerted in advance to deal with any dangerous situation.

Tracking obstacles in real time instead of just detecting them, is necessary for several reasons. First, to assess whether an obstacle can pose a danger for the host vehicle, we need to know the recent history of its dynamic behavior. If the dynamic behavior of the obstacle is rated

as dangerous, precautionary action has to be taken to avoid a collision. Tracking generates a history of the obstacle's dynamic behavior.

Second, tracking helps eliminate false alarms which typically appear momentarily and then disappear. This discontinuous and momentary behavior is readily identified through tracking. A real target has continuous trajectory over a period of time and will not be misinterpreted as a false alarm or vice versa. Thus alerts signals are generated only for really dangerous obstacles so that the driver can concentrate on taking a preventive action.

Third, in situations where a smaller obstacle is momentarily masked behind a larger one, tracking still keeps a record of its existence for some time. When the smaller target suddenly re-emerges from behind the larger target, the tracking system will generate an alert if it is on a collision course with the host vehicle.

A side advantage is that interference from other radars can be nullified using a tracking system. Radar is an active sensor which sends out an electromagnetic wave towards the obstacles and detects the obstacles by processing the reflected wave. Several radars operating close to one another with the same frequency range can interfere with one another. However, if every radar tracks the targets that it has already detected, chances of interference from other radars are greatly reduced.

The available literature on MTT systems mainly concern aerial target tracking. The dynamics of the aerial targets are very different than those an automotive driver has to deal with on the roads. Hence the application designed for aerial target tracking cannot be directly applied to road vehicle tracking.

The radars meant for automotive use have a shorter range and a narrower field of view. Unlike the ground-based air-traffic radars, there is no mechanical scanning of the field of view by the automotive radars. The automotive radar covers a fixed conical region in front of it. The volume of this conical region depends on the radar range and its coverage angles in azimuth and elevation.

The traffic density on the roads is very different than the air traffic density. The nature and types of obstacles on the roads are completely different and varied. In this case an obstacle may be a stationary road sign, a pedestrian, a car or a truck all with different dynamic behaviors. The road traffic conditions also change constantly e.g. conditions on an urban road are not the same as on a highway or in a rural or a mountainous region. The traffic on the roads is not expected to stop or get delayed in poor weather conditions.

All these factors make road target tracking applications very different from aerial target tracking applications. However, being a relatively new field, literature about the road target tracking is very rare, although we do find small bits and pieces scattered around various other fields of research. Consequently we have to redesign the MTT application for automotive safety keeping all these factors in mind. This is the principal theme of this chapter.

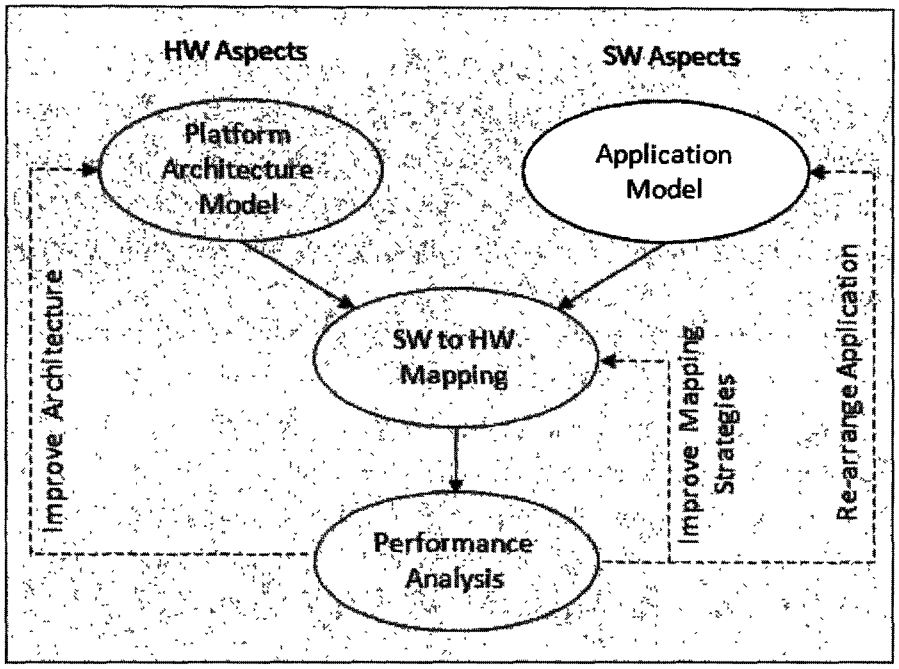


Figure 15: Application Modeling Aspects of the Co-design Flow

Starting with the right hand side of the Co-design flow (43) in Figure 15, we discuss the application design and modeling aspects of our Multiple Target Tracking (MTT) system. First we explain the terms and definitions used in the literature and describe the basic concepts of target tracking. Then we describe the components of a generalized MTT system. After that we present our model of the MTT system for automotive safety applications. We divide the application into various logical components and explain mathematical models for them. The application is modeled in such a way that it can be coded into manageable functions for modular software development. To render the application development process easily understandable, we use diagrams and illustrations alongside key mathematical concepts. Since the main theme of the application is automotive safety, illustrations mostly use vehicle images. Flow charts and pseudo codes are provided where required.

2. Multiple Target Tracking (MTT) Application

The purpose of target tracking is to collect data from the sensor field of view (FOV) containing one or more potential objects of interest and to partition the sensor data into sets of observations, or tracks (44).

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 as a vector that contains 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 its 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 the one that characterizes the target's dynamic behavior and its position in space in a 100% correct and exact manner. Because of the noise in the propagation channel, imperfections in the sensor and randomly varying conditions, the true state of the target cannot be determined. However, 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 systems deal with three types of states:

The *Observed State* or *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 an *observation model or measurement model.* The observation 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.*

The *Predicted State or Prediction* is the second representation of the target's true state. Prediction is done for the next cycle before the sensor sends the observation about it. 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.* The process model mathematically relates the predicted state with 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.*

Estimated State or 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 work, the term *scan* is used to specify the periodic sweep of the 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 duration between two successive radar scans. The PRT for the radar unit (45) 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 imposes an upper limit on the latency of the slowest module in the application.

2.2. General Principles

To explain the principle of target tracking, we consider tracking a single object for the moment. The general idea behind target tracking is illustrated in Figure 16. In this illustration, at *scan 0*, the initial *measured* or *observed state* of a target is taken as a *seed state*. The seed state is the a priori information about the targets. It is used to initialize the tracking system. It may be a measured state or a previously known state of a target obtained through some other means such as a centralized detection system. Using the seed state and the prediction process model, the next state of the target is predicted before new information about the target state is sent by the radar in *scan 1*. The predicted state, being based on an imperfect mathematical model, is not a 100% accurate representation of the target's *true state*. When the radar sends back new information in *scan 1*, we get a second representation of the *true state* of the target. This representation is not 100% accurate either due to the tolerances in the radar measurements and the noisy propagation channel. At this stage we have two representation of the target's true state, each with a different amount of inaccuracy. The goal here is to obtain a third representation which is closer to the true state than both the predicted and measured states. This is done by associating weights with the predicted and the measured states. The values of these weights are inversely proportional to the variances in the prediction model and the measurement model. This third representation is the *estimated* or corrected state. The predicted state for *scan 2* is based on the corrected state of *scan 1*. When measurements are received in *scan 2*, the state is corrected again. This process continues as long as the tracking system is powered on.

Since every predicted state is calculated using the previously corrected state hence the

current correction embodies all the previous corrections. This obviates the need to store and use the whole history for calculating the current state. This in turn helps substantially in minimizing the system memory requirements.

The tracking process explained above takes a single target into account. Single target tracking (STT) systems are designed to operate in a closed loop for tracking a single object. The STT system typically operates in the manner shown in Figure 17 with the objective of keeping the sensor pointed at the single target of interest. The STT tracking loop operates on the discriminant (or measured error) data that measures the offset between the sensor current pointing angle and the target location. Offsets in range or range rate can also be used. The STT tracking loop operates to null these offsets. Specifically a radar based STT system attempts to keep the antenna directed at the target. Additionally, a radar based STT system will typically define range or range rate gates that are adjusted to remain centered about the target range and range rate predictions.

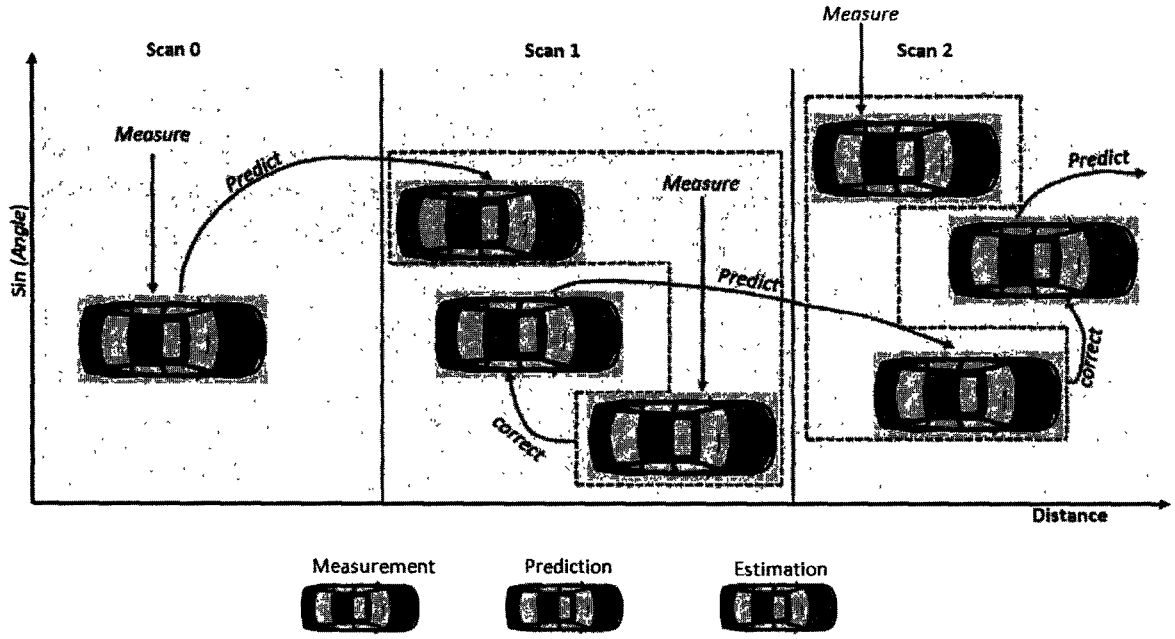


Figure 16: State prediction and estimation in STT

Referring again to Figure 17, for an STT system, the discriminanat or measured error, data are directly input to a filter. Because the sensor is assumed to be dedicated to a single target, there is no need to perform a complex data association function, such as that discussed latter in an MTT system.

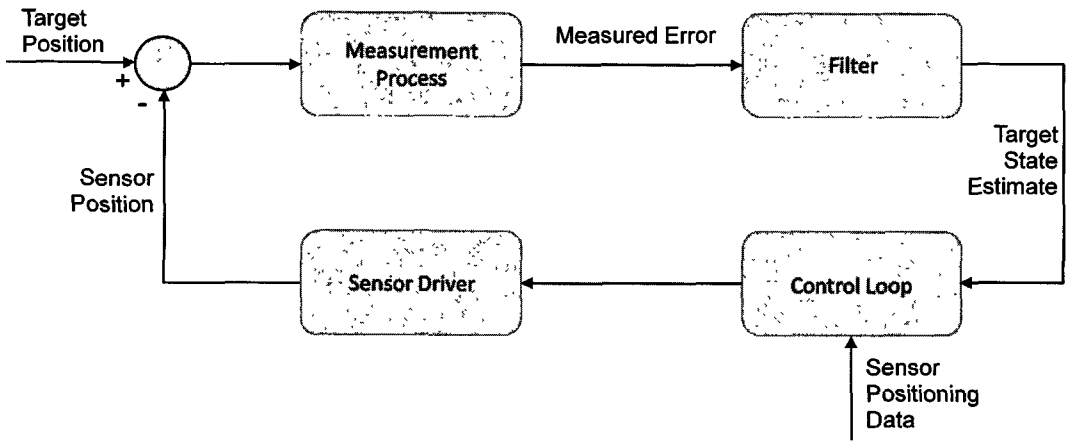


Figure 17: The Single Target Tracking Loop

The filtered estimate of the target position relative to the sensor pointing angle is used as an input to a control loop that attempts to keep the sensor pointing at the target.

Single Target Tracking systems are useful in applications like camera based surveillance where a single scene of interest has to be focused on. In contrast, driver Assistance Systems must cope with more than a single obstacle. Hence a tracking-based DAS would use an embedded MTT application rather than an STT application. Figure 18 shows a simplified demonstration of the MTT based DAS.

Suppose in *scan 1* the radar detects three obstacles (green cars) and sends their states back to the tracking system. As in the case of the STT, the tracking system predicts the next states (blue cars) for all the three targets. In *scan 2*, the radar detects four targets and reports their states to the system. Now to calculate the estimated states we have to take into account the predicted states and the *corresponding* observed states just reported by the radar. But to find the corresponding observed state for every predicted state, we have to resolve the following issues.

To be able to correct the states we have to choose the observed state of *one and only one* target and pair it with the relevant predicted state. Since in *scan 1* three targets were detected hence we have predicted states for three targets. In *scan 2*, the radar reports the presence of four targets in its FOV. This gives rise to twelve possible parings (dotted arrows) among the predictions and the observations while we should have *at most* three pairings. So the first issue is to eliminate all the irrelevant parings.

Obviously *at least* one of these four targets is a new entry. We have to identify which one(s) is (are) the new entry (entries). In a different scenario the radar could have reported the presence of less than three targets in *scan 2*. This implies that one or more of the targets detected in *scan 1* are no more in the radar FOV. In such cases we have to identify the

irrelevant predictions and delete them from the system memory.

A complex *data association* logic is needed to deal with all these issues in MTT which don't exist in STT. The data association logic is also required in order to sort out recurrent sources that are not of interest (such as background clutter), and false signals that have little or no correlation in time. The main components of a typical MTT application are described in the following subsection.

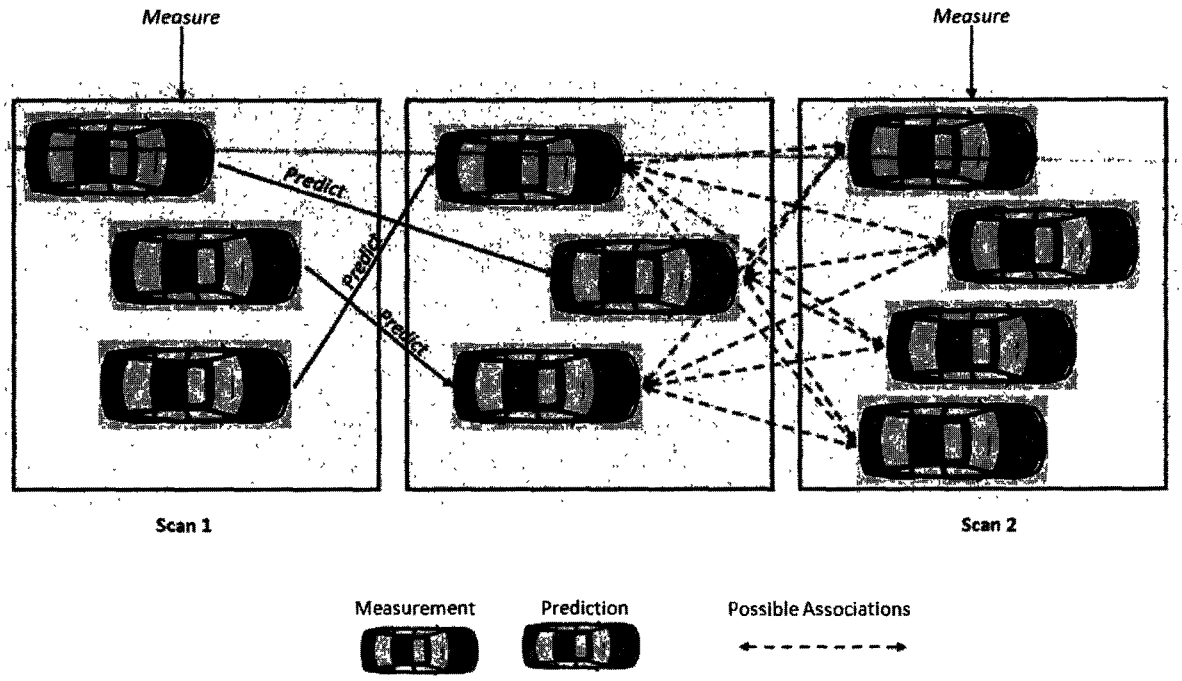


Figure 18: State prediction and estimation in MTT

2.3. MTT building blocks

A generalized view of Multiple Target Tracking (MTT) system is given in Figure 19. This illustration helps in understanding the system. In reality the partitioning lines among the functioning blocks are not so clearly defined. 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 further divided into three sub-blocks: *Track maintenance*, *Observation-to-Track Assignment* and *Gate Computation*.

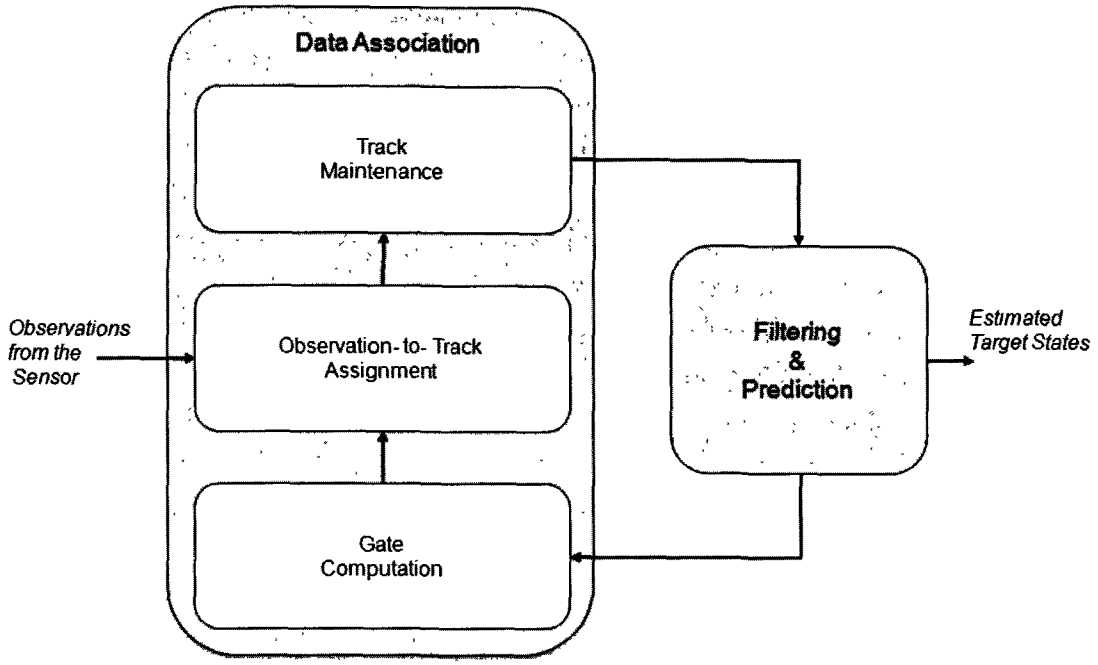


Figure 19: Building Blocks of a Generalized MTT System

The *data association* block resolves the issues discussed in above. The *Observation-to-Track Assignment* sub-block sorts out the observations coming from the radar and couples them with the relevant predictions made earlier.

The *Track Maintenance* sub-block is responsible for dealing with the new entries into the radar FOV and also those which disappear from the radar FOV in the current scan. All this information is passed on to the *Filtering and Prediction* block.

The *Filtering and Prediction* block predicts the next states for all the current targets and estimates their current states. The estimated states are the system output. The filtering and prediction block also passes the prediction error covariance and the predicted state information back to the *Gate Computation* sub-block of the data association.

The *Gate Computation* sub-block defines demarcating limits around the predicted states based on the prediction error covariance. The higher the covariance, the higher these limits are. The limits for all the elements of a state vector together form a multidimensional probability *gate*. In our application it is a 2 dimensional rectangular plane since we are interested only in two parameters i.e. the distance and the azimuth. The predicted state for a target is positioned at the center of the corresponding gate.

The gates are used by the *Observation-to-Track Assignment* sub-block to screen out improbable observation-prediction pairings. All observations falling within the limits of a

particular gate are potential candidates for pairing with the prediction at the center of that gate. In the stage of the *Observation-to-Track Assignment* only the most probable pair is retained for correcting the state of the concerned target.

Figure 19 represents a text book view of an MTT system that can be found in (44) and in (46). The practical implementation and internal details of the design may vary depending on the end use and implementation technology. For example the filtering and prediction part may be implemented choosing from a variety of algorithms such as *alpha-beta filters* (47), *mean-shift algorithms* (48), *Kalman filters* (44), (49), (47) etc. Similarly, the *Observation-to-Track Assignment* part is usually modeled as an *Assignment Problem* which is extensively used in operations research. The assignment problem itself may be solved in a variety of ways, for example using the *Auction algorithm* (50), (51) or the *Hungarian (or Munkres) algorithm* (52), (53).

The choice of algorithms for the sub-blocks is driven by factors like application environment, implementation platform, system architecture, amount of the available processing resources, hardware size of the end product, track precision and system response time etc.

3. MTT Mathematical Modeling: Our Approach

As stated above, the internal details, the choice of algorithms and the interactions among different functions of the MTT system is driven by factors like implementation platform, system architecture etc. Based on the arguments given in Chapter 1and chapter 2, we chose to use a multi-processor architecture for our system and FPGA as the implementation platform. A multi-processor architecture can be exploited very efficiently if the underlying application is divided into simpler modules which can run in parallel. Parallelism is one of the strongest features of the FPGA based system implementations. Moreover, simple multiple modules can be managed, modified and upgraded easily and independently of one another as long as the interfaces among them remain unchanged.

For the purpose of modular implementation we organized the application into sub-modules as shown in Figure 20. The functioning of the system is explained as follows.

Assuming recursive processing as shown by the outer loop (between *Data Association* and *Filtering and Prediction*) in Figure 20, *tracks* would have been formed on the previous radar *scan*. When new *observations* are received from the radar the processing loop is executed.

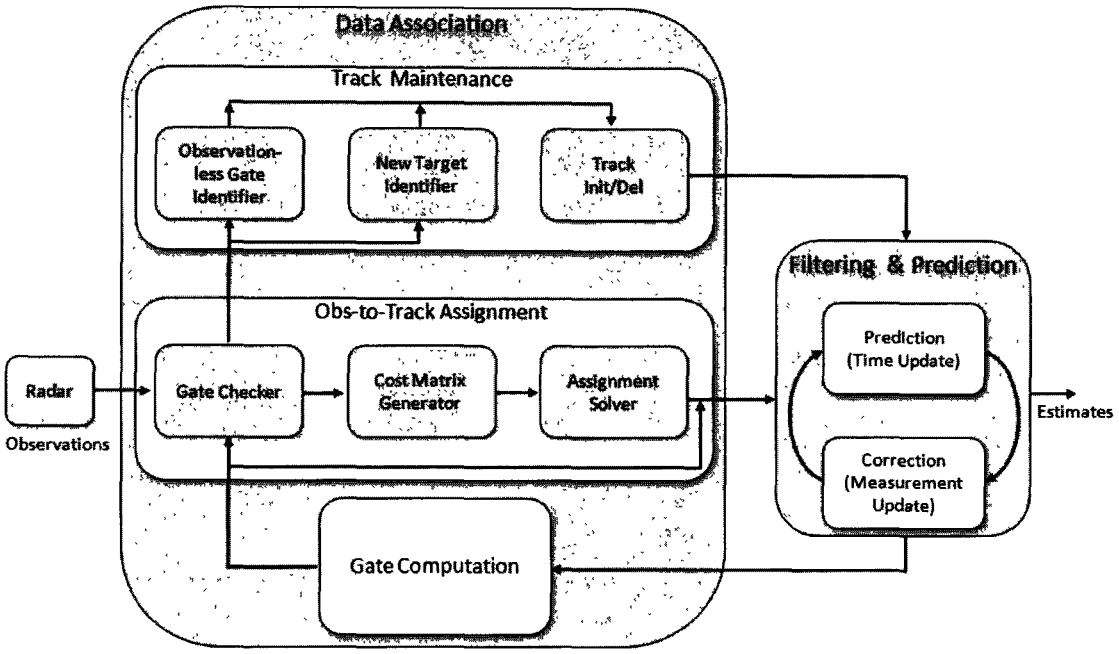


Figure 20: Our Design of the MTT Application

In the first cycle, the *incoming observations* would simply pass through the *Gate Checker, Cost Matrix Generator* and *Assignment Solver* on to the filters' inputs. The 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 sensor. It estimates the current state of the target and predicts its next state before the next observation is available. As mentioned earlier the estimation process and the MTT application as a whole rely on mathematical models for its operation. The mathematical process model and the measurement model we proposed are detailed in sections 3.1 and 3.2 respectively. All the components of the proposed MTT implementation are explained in sections 3.3 through 3.8.

3.1. Process Model

The *process model* mathematically projects the current state of a target into the future. This can be presented in a linear stochastic difference equation as

$$Y_k = A\,Y_{k-1} + B\,U_k + W_{k-1} \qquad (3.1)$$

Where:

- Y_{k-1} and Y_k are n-dimensional state vectors that include the quantities to be estimated.
- Vector Y_{k-1} represents the state of a target at instant $k-1$ while Y_k represents the state at instant k.
- The $n \times n$ matrix A in the difference equation (3.1) relates the state at time step $k-1$ to the state at time step 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 the state transformation from instant $k-1$ to instant k, in the absence of any driving signal and process noise.
- The $n \times l$ matrix B relates the optional control input $U_k \in \Re^{\ell}$ 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 (own ship) motion.

3.2. 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

$$Z_k = HY_k + V_k \qquad (3.2)$$

Where:

- Z_k is the measurement or *observation vector* containing two elements, distance d and angle θ as shown below.
- The $m \times n$ *observation matrix* H in the measurement equation (3.2) relates the current state to the *measurement (observation) vector* Z_k. The dimension m is the number of elements in Z_k. In our application m is 2 as we have two elements d and θ in Z_k.
- The term V_k in equation (3.2) is a random variable representing the measurement noise.

- The terms W_k and V_k in equations (3.1) and (3.2) are zero-mean, white, Gaussian noises and are assumed to be independent of each other, with normal probability distributions mathematically denoted as $p(W_k) \sim N(0,Q)$ and $p(V_k) \sim N(0,R)$.

For implementation we chose the example case given in (20). In this example the matrices and vectors in equations (3.1) and (3.2) have the forms shown below. In the rest of this document the numerical values of all the matrix and vector elements are borrowed from this example. Quantity T in matrix A, is 0.025 seconds and it is the radar Pulse Repetition Time (PRT) specific to the radar unit (45) we are using in our system. Y_k, A and Z_k have the following forms:

$$Y_k = \begin{bmatrix} y_{11} \\ y_{21} \\ y_{31} \\ y_{41} \end{bmatrix} \qquad A = \begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad Z_k = \begin{bmatrix} d \\ \theta \end{bmatrix}$$

Here y_{11} is the target range or distance, y_{21} is range rate or speed, y_{31} is the angle (azimuth), y_{41} is the angle rate or the 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.

3.3. Filtering and Prediction

Having devised the process and measurement models, we need an *estimator* which would use these models to estimate the true state. The process and measurement models presented above for target dynamics can be classified as linear models with Additive White Gaussian Noise (AWGN). For this reason we use the Kalman filter because it is a recursive Least Square Estimator (LSE) considered to be the optimal estimator for linear systems with AWGN probability distribution (47), (49). The Kalman filter is extensively used in various applications where the process noise can be assumed to have a Gaussian probability distribution. In our future work we shall evaluate various other filters with respect to their precisions, processing speeds and computational resource requirements etc.

The "*Tracking filters*" block in Figure 20, is particularly important because we need as many filters 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.

Given the process and the measurement models in equations (3.1) and (3.2), the Kalman filter equations are

$$\hat{Y}_k^- = A\hat{Y}_{k-1} + BU_k \qquad (3.3a)$$

$$P_k^- = AP_{k-1}A^T + Q \qquad (3.3b)$$

$$P_k = (I - KH)P_k^- \qquad (3.3c)$$

$$K = P_k^- H^T (HP_k^- H^T + R)^{-1} \qquad (3.3d)$$

$$\hat{Y}_k = \hat{Y}_k^- + K(Z_k - H\hat{Y}_k^-) \qquad (3.3e)$$

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

The covariance update equation (3.3c), is based on the assumption that the Kalman gain, K, is computed identically from equation (3.3d). If, due to computational error, the gain calculation is not exact or if the gain is chosen in another manner, the stabilized form of the covariance update equation that should be used is

$$P_k = [I - KH]P_k^-[I - H]^T + KRK^T \qquad (3.3f)$$

Note that (3.3c) follows directly from (3.3f) when the gain K is computed exactly from (3.3d). However, if, due to computational error, the gain (3.3d) is not exact, the use of equation (3.3f) will enhance stability.

The covariance matrix is defined in terms of the zero-mean Gaussian estimation error vector:

$$P_k = E\left\{[Y_k - \hat{Y}_k][Y_k - \hat{Y}_k]^T\right\} \qquad (3.3g)$$

The letter ***E*** here signifies *Expected Value* or *Mean Value*.

The vector difference between measured and predicted quantities, $\tilde{Y}_k = Z_k - H\hat{Y}_k^-$ is defined to be the residual (or innovation) vector with residual covariance matrix S defined as

$$S = HP_k^- H^T + R \qquad (3.3h)$$

The newly introduced vectors and matrices in equations (3.3a) through (3.3h) have the following forms.

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

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

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

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

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

Matrices K and P_k^- have the following forms.

$$P_k^- = \begin{bmatrix} p_{11}^- & p_{12}^- & p_{13}^- & p_{14}^- \\ p_{21}^- & p_{22}^- & p_{23}^- & p_{24}^- \\ p_{31}^- & p_{32}^- & p_{33}^- & p_{34}^- \\ p_{41}^- & p_{42}^- & p_{43}^- & p_{44}^- \end{bmatrix} \qquad K = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \\ k_{31} & k_{32} \\ k_{41} & k_{42} \end{bmatrix}$$

Matrix P_k is similar in form to P_k^- except for the superscript '$^-$'. The scan index k is 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 21. In the prediction step (also called *time update*), the filter predicts the next state and the error covariance associated with the state prediction using equations (3.3a) and (3.3b) respectively. In the correction step (also called *measurement update*), the filter calculates the filter gain, estimates the current state and the error covariance of this estimation using equations (3.3c) through (3.3e).

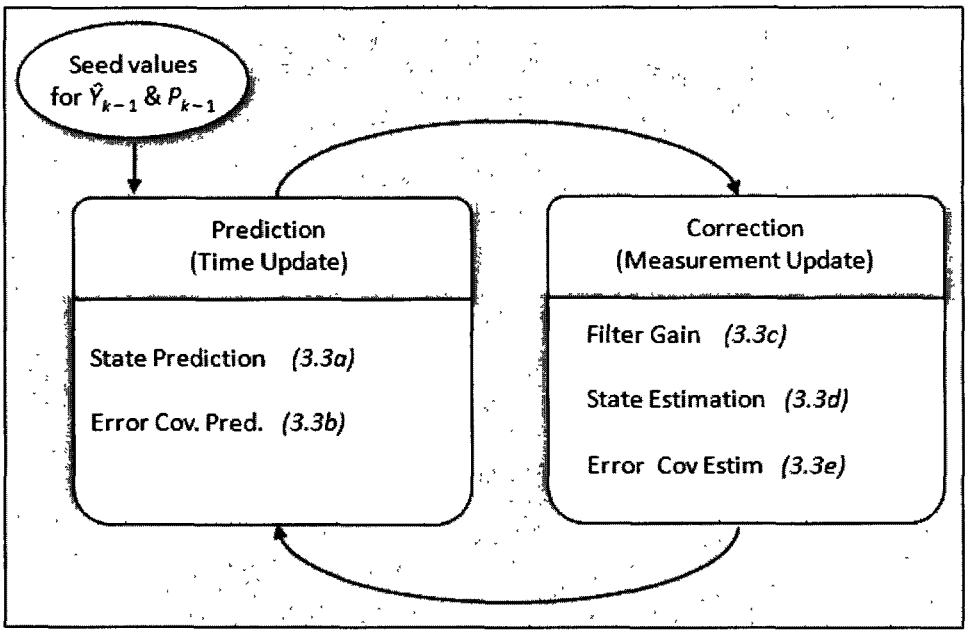


Figure 21: The Kalman Filter

Figure 22 shows the state (position) of a target estimated by the Klaman filter against the true position and the observed position (measured by the radar). Notice how closely the estimated position follows the true position as compared with the observed position after the 20 transitional iterations. The estimated position shown in Figure 22 starts at zero because we took the seed value to be zero. The transition duration (0 to 20 iterations) depends on the initial

values in the matrices Q and R.

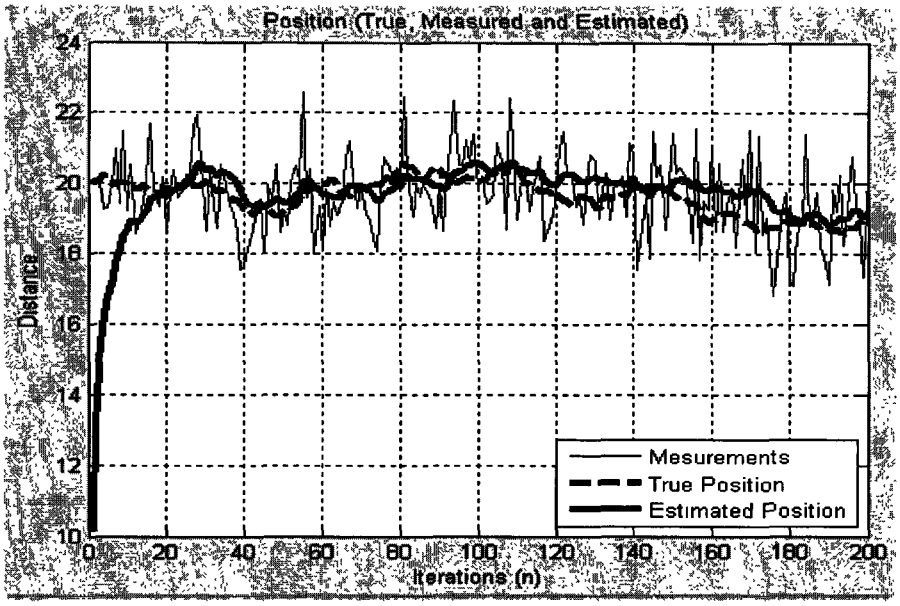


Figure 22: Kalman Filter Output

In the case of a single target tracking system, 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 just a control loop and an actuator to correct the pointing angle of the radar. Since we are dealing with multiple targets at the same time, we have to identify which of the incoming observed states to associate with which of the predicted states for making the estimation for each target. This is the job of *data association* block shown in Figure 20 The data association sub-modules are explained one by one in the following subsections.

3.4. Gate Computation

The first step in data association is 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 prediction. The dimensions of the gates are proportional to the prediction error covariance P_k^-. If the innovation $(Z_k - H\hat{Y}_k^-)$ for an observation is greater than the gate dimensions, the observation fails the gate and hence it cannot be associated with the concerned prediction. If an observation passes a gate, it may be associated with the prediction at the center of that gate. In fact, observations for more than one target may pass a particular gate. In such cases all these observations are associated with the single prediction. The *Gating* process can 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 3.7), a strictly one-to-one coupling is established between observations and predictions. The gate computation model is summarized below.

Define $\tilde{Y}$ to be the *residual* or *innovation* vector which is the difference between the actual measurement (observation) Z_k and the expected (predicted) measurement vector $\left[H\hat{Y}_k^-\right]$. In general for track i at scan k,

$$\tilde{Y}_i = Z_k - H\,\hat{Y}_i^- \qquad (3.4)$$

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

$$\left|Z_{kl} - H\,\hat{Y}_{il}^-\right| = \left|\tilde{Y}_{il}\right| \le K_{Gl}\sigma_r \qquad (3.5)$$

In equations (3.4) and (3.5) i is an index for track i, G signifies gate and l is replaced either by d or by θ, whichever is appropriate (see equations 3.10 and 3.11). The term σ_r is the *residual standard deviation* and is defined in terms of the *measurement variance* σ_z^2 and *prediction variance* $\sigma^2_{\hat{y}_k^-}$. A typical choice for K_{Gl} is $\left[K_{Gl} \ge 3.0\right]$. 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 (44), (46). This concept comes from the famous *3-sigma rule* in statistics (54). According to this rule, for many reasonably symmetric distributions, almost all of the population lies within three standard deviations of the mean. For the Gaussian distribution about 99.7% of the population lies within three standard deviations of the mean.

In its matrix form for scan k and track i, equation (3.4) can be written for scan k as follows

$$\tilde{Y}_{ik} = \begin{bmatrix} \tilde{y}_{ik11} \\ \tilde{y}_{ik21} \end{bmatrix} = \begin{bmatrix} d_i \\ \theta_i \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \hat{y}_{k11}^- \\ \hat{y}_{k21}^- \\ \hat{y}_{k31}^- \\ \hat{y}_{k41}^- \end{bmatrix}$$

This can be simplified down to

$$\tilde{Y}_{ik} = \begin{bmatrix} \tilde{y}_{ik11} \\ \tilde{y}_{ik21} \end{bmatrix} = \begin{bmatrix} d_i - \hat{y}_{k11}^- \\ \theta_i - \hat{y}_{k31}^- \end{bmatrix} \qquad (3.6)$$

Consequently equation (3.5) gives

$$\begin{vmatrix} \tilde{y}_{ik11} \\ \tilde{y}_{ik21} \end{vmatrix} \le K_{GI}\sigma_r \qquad (3.7)$$

Since the state vector contains two elements d and θ, the variances σ_r, σ_z^2 and $\sigma_{\hat{y}_k^-}^2$ also have two components each. The values of the σ_z^2 components are chosen from the diagonal elements of R i.e. $\sigma_{z_d}^2 = r_{11}$ and $\sigma_{z_\theta}^2 = r_{22}$.

The values of the $\sigma_{\hat{y}_k^-}^2$ components are chosen from the diagonal elements of P_{ik}^- i.e. $\sigma_{\hat{y}_d^-}^2 = p_{i22}^-$ and $\sigma_{\hat{y}_\theta^-}^2 = p_{i44}^-$.

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

$$\sigma_{rd} = \sqrt{r_{11} + p_{i22}^-} \qquad (3.8)$$

$$\sigma_{r\theta} = \sqrt{r_{22} + p_{i44}^-} \qquad (3.9)$$

From (3.7), (3.8) and (3.9) we get

$$\left|\tilde{y}_{i\,k11}\right| = \left|\tilde{y}_{i\,kd}\right| \le 3.0\sqrt{r_{11} + \bar{p}_{i22}} \qquad (3.10)$$

$$\left|\tilde{y}_{i\,k21}\right| = \left|\tilde{y}_{i\,k\theta}\right| \le 3.0\sqrt{r_{22} + \bar{p}_{i44}} \qquad (3.11)$$

We call the functions computing equation (3.10) and equation (3.11) as the *Innov_d calculator* and *Innov_a calculator* respectively. Equations (3.10) and (3.11) together put the limits on the *residuals (innovations)* $\tilde{y}_{i\,kd}$ and $\tilde{y}_{i\,k\theta}$. In other words, the difference between an incoming observation and the prediction for target i must comply with equations (3.10) and (3.11) for the observation to be assigned to prediction i. The *Gate Checker* sub-function explained next, tests all the incoming observations for this compliance.

3.5. Gate Checker

The *Gate Checker* tests whether an incoming observation fulfills the conditions set in equations (3.10) and (3.11). 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 equations (3.10) and (3.11) are complied with. In effect, the Gate Checker sets or resets the binary elements of an *NxN* matrix termed as the *Gate Mask* matrix M shown below, where N is the maximum number of targets that can be tracked.

Predictions

$$M = \begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1N} \\ m_{21} & m_{22} & \cdots & m_{2N} \\ \vdots & \vdots & \cdots & \vdots \\ m_{N1} & m_{N2} & \cdots & m_{NN} \end{bmatrix} \} \text{Observations}$$

$$m_{ij} = \begin{cases} 1 & \text{if observation } i \text{ obeys (3.10) \& (3.11) for track } j \\ 0 & \text{otherwise} \end{cases}$$

If an observation i fulfills both the conditions of equations (3.10) and (3.11) for a prediction j, the corresponding element m_{ij} of matrix M is set to 1 otherwise it is reset to 0. This process is illustrated pictorially in Figure 23.

The solid arrows marked with 1's along them, indicate that the observations and the predictions they are linking, have passed the gating conditions of equations (3.10) and (3.11). Notice here that at this stage a single observation is linked with more than one prediction and vice versa. The dotted arrows with 0's along them indicate that these pairs didn't pass the gating conditions and hence these pairings are not probable.

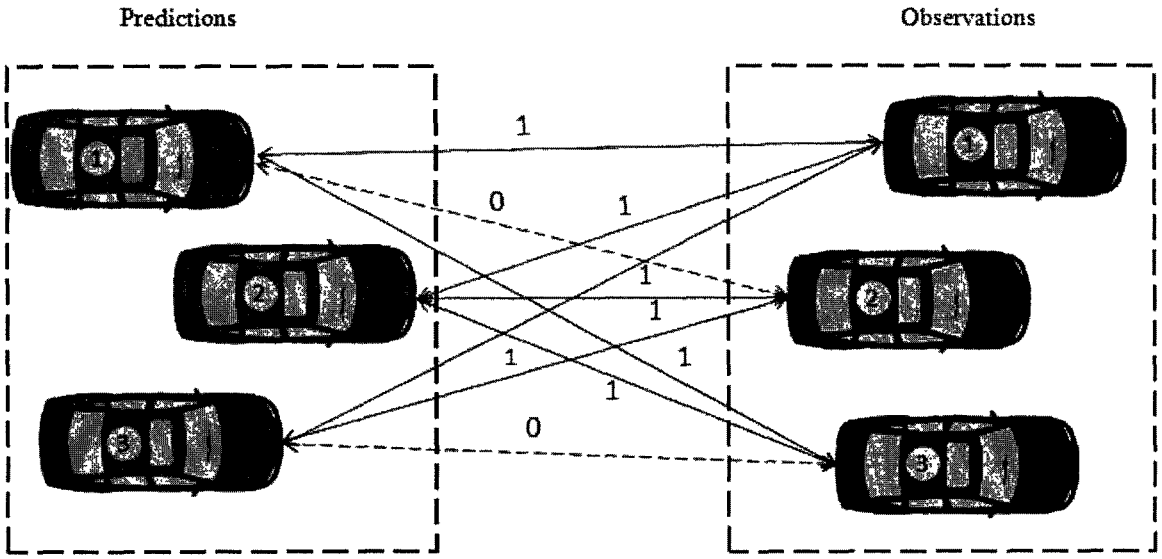


Figure 23: The Gate Checking Process

The matrix M generated for this example is as follows.

$$M = \overbrace{\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}}^{\text{Predictions}} \Big\} \text{Observations}$$

The Mask Matrix M would typically have more than one 1's in a column or a row indicating the possibilities of couplings. 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 attribute a cost to every probable observation-prediction coupling. This is done by the *Cost Generator* block explained next.

3.6. Cost Matrix Generator

The Mask matrix with all the probable pairs 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 a *cost matrix C* as shown below.

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.

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1N} \\ c_{21} & c_{22} & \cdots & c_{2N} \\ \vdots & \vdots & \cdots & \vdots \\ c_{N1} & c_{N2} & \cdots & c_{NN} \end{bmatrix}$$

Predictions

Observations

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

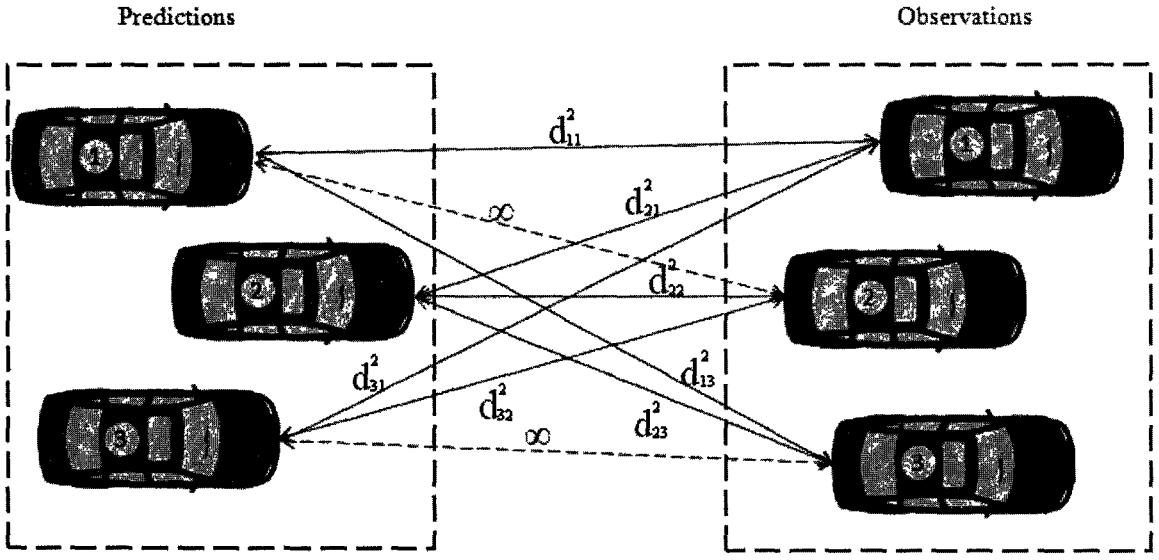


Figure 24: The Cost Matrix Generation Process

This can be illustrated as in Figure 24. The improbable links are attributed with the highest cost (∞) while the probable ones are attributed with costs proportional to the amount of their respective probabilities.

The *Cost Matrix C* corresponding to this example is as follows.

$$C = \overbrace{\begin{bmatrix} d_{11}^2 & d_{12}^2 & d_{13}^2 \\ \infty & d_{22}^2 & d_{23}^2 \\ d_{31}^2 & d_{33}^2 & \infty \end{bmatrix}}^{Predictions} \Big\} Observations$$

Note here that higher the number of the improbable links the easier and quicker it is for the assignment solver to arrive at the final one-to-one coupling.

The distance d_{ij}^2 is calculated as follows.

Define $$S_{ij} = H\ P_k^- H^T + R \qquad (3.12)$$

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 $\tilde{Y}_{ij}$ calculated as

$$d_{ij}^2 = \tilde{Y}_{ij}^T S_{ij}^{-1} \tilde{Y}_{ij} \qquad (3.13)$$

Equation (3.12) can be written in its matrix form as

$$S_{ij} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_{i11}^- & p_{i12}^- & p_{i13}^- & p_{i14}^- \\ p_{i21}^- & p_{i22}^- & p_{i23}^- & p_{i24}^- \\ p_{i31}^- & p_{i32}^- & p_{i33}^- & p_{i34}^- \\ p_{i41}^- & p_{i42}^- & p_{i43}^- & p_{i44}^- \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} r_{11} & 0 \\ 0 & r_{22} \end{bmatrix}$$

This simplifies down to

$$S_{ij} = \begin{bmatrix} \bar{p}_{i11} + r_{11} & \bar{p}_{i13} \\ \bar{p}_{i31} & \bar{p}_{i33} + r_{22} \end{bmatrix} \qquad (3.14)$$

And the inverse of S_{ij} is given as

$$S_{ij}^{-1} = \frac{\begin{bmatrix} \bar{p}_{i33} + r_{22} & -\bar{p}_{i13} \\ -\bar{p}_{i31} & \bar{p}_{i11} + r_{11} \end{bmatrix}}{\left(\left(\bar{p}_{i11} + r_{11}\right)\left(\bar{p}_{i33} + r_{22}\right) - \bar{p}_{i11}\bar{p}_{i33}\right)}$$

Using this inverse and equations (3.6), (3.13) and (3.14), d_{ij}^2 is calculated as follows.

$$d_{ij}^2 = \frac{\begin{bmatrix} \tilde{y}_{ik11} & \tilde{y}_{ik21} \end{bmatrix} \begin{bmatrix} \bar{p}_{i33} + r_{22} & -\bar{p}_{i13} \\ -\bar{p}_{i31} & \bar{p}_{i11} + r_{11} \end{bmatrix} \begin{bmatrix} \tilde{y}_{ik11} \\ \tilde{y}_{ik21} \end{bmatrix}}{\left(\left(\bar{p}_{i11} + r_{11}\right)\left(\bar{p}_{i33} + r_{22}\right) - \bar{p}_{i11}\bar{p}_{i33}\right)} \qquad (3.15)$$

Recall here that $\tilde{y}_{ik11} = \tilde{y}_{ikd}$ and $\tilde{y}_{ik21} = \tilde{y}_{ik\theta}$. The values of numerator and denominator in equation (3.15) are different for different targets. Hence the value of d_{ij}^2 is different for different observation-prediction pairs.

The cost matrix demonstrates a conflict situation where several observations are potential candidates to be associated with a particular prediction and vice versa. A conflict situation is illustrated in Figure 25. This figure depicts the general idea of the gating process. 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 prediction with the smallest statistical distance d_{ij}^2 from the observation is the strongest candidate.

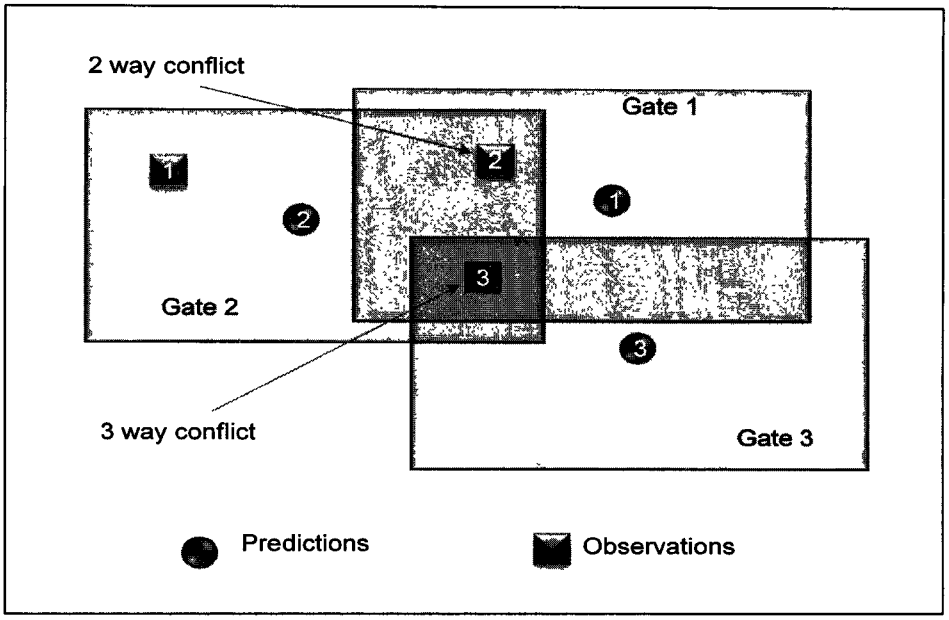


Figure 25: A conflict Situation in Data Association

The Mask Matrix M and Cost Matrix corresponding to Figure 25 are given below.

$$M = \overbrace{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}}^{\text{Predictions}} \Big\} \text{Observations} \qquad C = \overbrace{\begin{bmatrix} \infty & d_{12}^2 & \infty \\ d_{21}^2 & \infty & \infty \\ d_{31}^2 & d_{32}^2 & d_{33}^2 \end{bmatrix}}^{\text{Predictions}} \Big\} \text{Observations}$$

To resolve these kinds of conflicts, the cost matrix is passed on to the *Assignment Solver* block which treats it as the well known *assignment problem* in operations research (55).

3.7. Assignment Solver

The *assignment solver* determines the finalized one-to-one pairings between predictions and observation. The pairings are made in a way to ensure minimum total cost for all the finalized pairings. The finalized observation-prediction pairings are passed on to the tracking filters which use them for estimating the current states of the targets and predicting their next

states as well as the *error covariance* associated with these predictions.

The *assignment problem* is modeled as follows.

Given a cost matrix of elements $C = \{c_{ij}\}$

find a matrix $X = \{x_{ij}\}$,

such that $C = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$ is minimized

subject to $\begin{cases} \sum_i x_{ij} = 1 \forall j \\ \sum_j x_{ij} = 1 \forall i \end{cases}$

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 i.e. $x_{ij} \in \{0,1\}$.

Matrix X can be found by using various algorithms. The most commonly used among them are Munkres algorithm (52) and Auction algorithm (51). We use the former in our application due to its inherent modular structure. The algorithm is described by the flowchart shown in

Figure 26.

The Munkres (or Hungarian) algorithm describes a procedure for manually finding out the solution by starring (0*) and priming (0`) the zeros and covering (drawing a line over) and uncovering the rows and columns of a two-dimensional cost matrix. This is because in 1957 when the algorithm was published, computers were not accessible to many people. As shown in the flow chart the algorithm consists of six discrete steps.

To arrive at the final solution, it passes repetitively from step 3 through step 6 as indicated by the three loops in the chart. We implemented these steps as separate procedures called up iteratively by the main program. The functioning of the main program and these procedures is explained by the pseudo code that follows the flowchart.

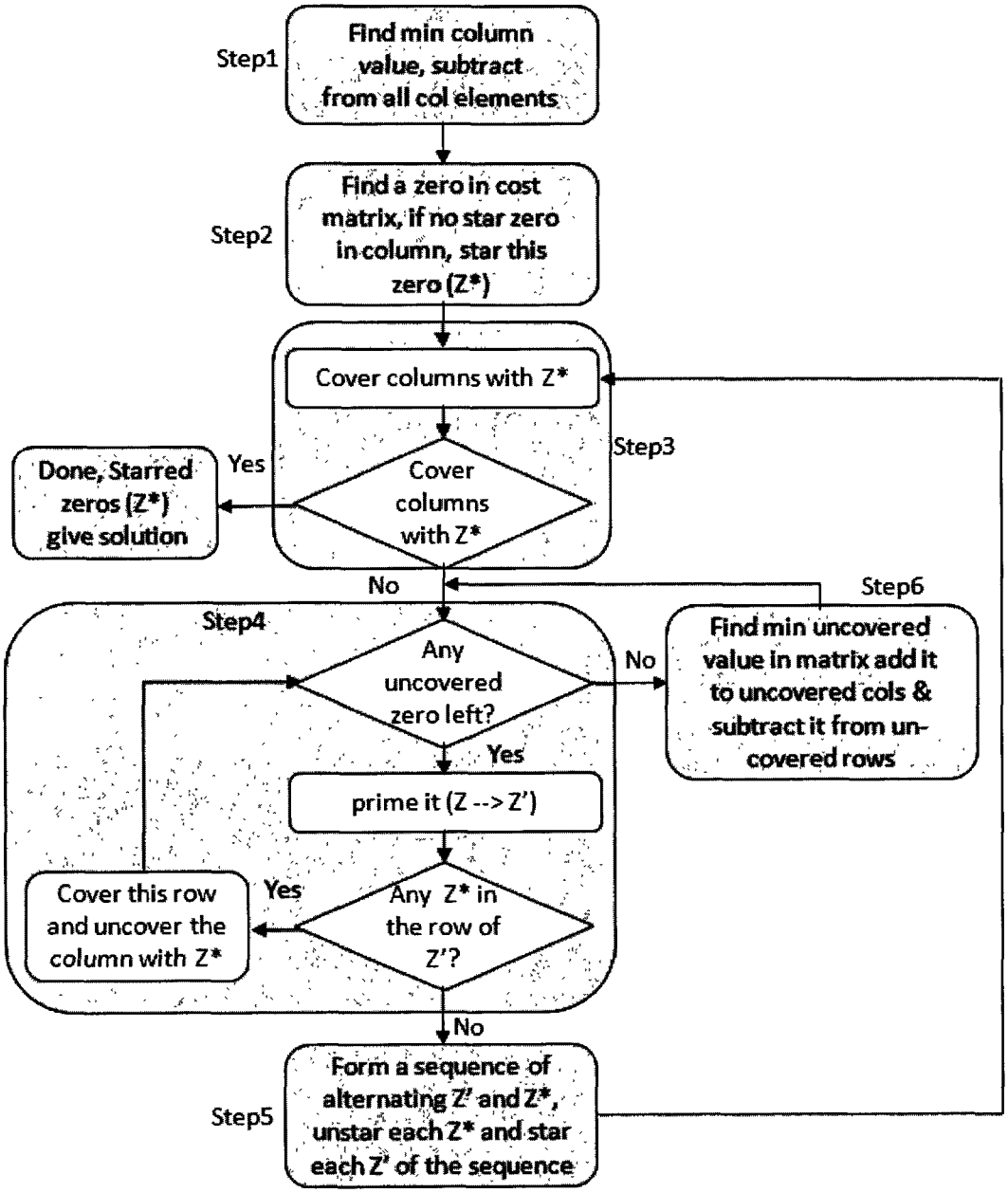


Figure 26: The Munkres Algorithm for the Assignment Problem

```
main:

{

//Global variable declaration

stepnum : integer;
done : boolean;

// Main body

      Begin

            done = false;

            stepnum = 1;

while (done = false) loop
```

/* In each pass of the outer loop (step 5 to step 3), the step procedure called, sets the value of a variable **stepnum**, for the next pass. When the algorithm is finished the value of **stepnum** is set to some value outside the range 1..6 so that the variable **done** will be set to **true** and the program will end. In the completed program the starred zeros flag the row/column pairs that have been assigned to each other. */

```
read stepnum

     if(stepnum = 1) goto step1;

     else if(stepnum = 2) goto step2;

     else if(stepnum = 3) goto step3;

     else if(stepnum = 4) goto step4;
```

```
      else if(stepnum = 5) goto step5;

      else if(stepnum = 6) goto step6;
      else   done = true;

 end loop;
}
```

Step 1

For every row of the cost matrix, find the smallest element.
Subtract it from every element in its row.
stepnum = 2

Step2

Find a zero Z in the resulting cost matrix.
If there is no starred zero already in its row or column, star this zero (Z → Z*).
Continue until all zeros have been considered.
Stepnum = 3

Step3

Cover every column containing a Z*.
Terminate the algorithm if all columns are covered. In this case, the locations of the Z* entries in the matrix provide the solution to the assignment problem.
done = true
stepnum = 7

Step4

Find an uncovered **Z** in the cost matrix and prime it, **Z** → **Z′**. If no such zero exists,
stepnum = 6 .

If no **Z*** exists in the row of the **Z′**, stepnum = 5.

If a **Z*** exists, cover this row and uncover the column of the **Z***. Return to Step 4.1 to find a new **Z**.

Step5

Construct the "alternating sequence" of primed and starred zeros:
Z_0: Unpaired **Z′**from Step 4.2
Z_1: The **Z***in the column of Z_0
Z_{2N}: The **Z′**in the row of Z_{2N-1}, *if* such a zero exists
Z_{2N+1}: The **Z***in the column of Z_{2N}
The sequence eventually terminates with an unpaired $Z' = Z_{2N}$ for some **N**.
Unstar each starred zero of the sequence.
Star each primed zero of the sequence, thus increasing the number of starred zeros by one.
Erase all primes, uncover all columns and rows, stepnum = 3.

Step6

Let **h** be the smallest uncovered entry in the resulting cost matrix.
Add **h** to all covered rows.
Subtract **h** from all uncovered columns
Without altering stars, primes, or covers, stepnum = 4

Running the *Cost Matrix C* through the Munkres Assignment algorithm we find the final solution *Matrix X*. An example of the solution matrix is given below which shows a result of the *Assignment Solver* for a 3x3 cost matrix.

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

It shows that observation 1 is to be paired with prediction 2, observation 2 with prediction 3 and observation 3 with prediction 1. The pictorial illustration corresponding to this solution is given in Figure 27 which is self explanatory.

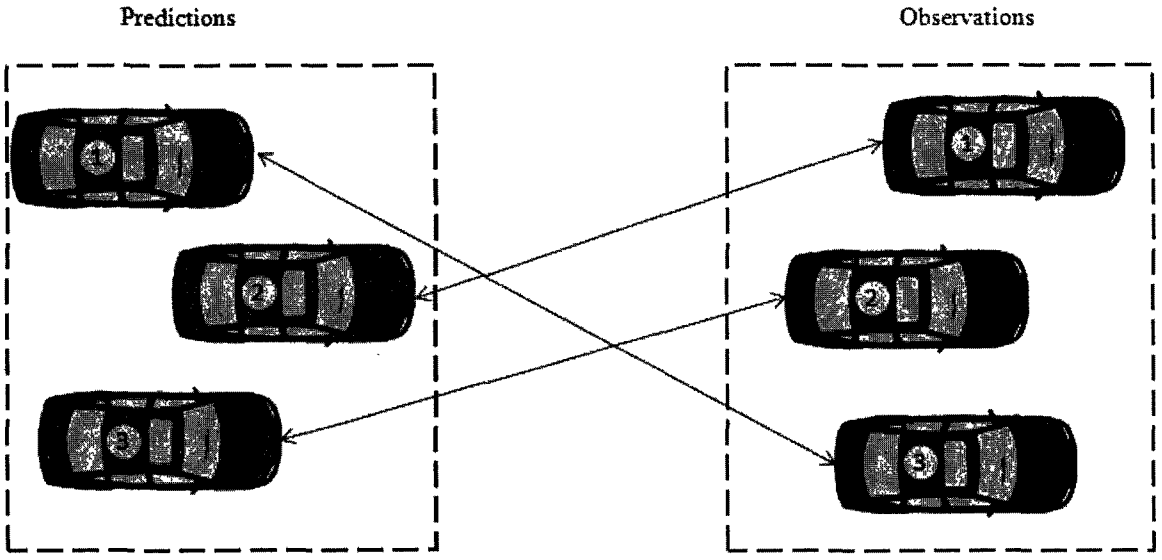


Figure 27: An example of the Assignment Solver Output

Now that we know which observation to pair with which prediction, we are ready to do the estimations. For this purpose the finalized pairs passed on to the Kalman filters which estimate the states of the concerned targets and predict their states for the next cycle.

3.8. Track Maintenance

All the steps described in sections 3.3through 3.7 are repeated indefinitely in the loop for every radar scan. However, there are certain cases where some additional steps have to be taken too. Together these steps are called *Track Maintenance*. The *Track Maintenance* sub-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 that are detected in the current radar scan which did not exist in the previous scans. On the other hand there would be situations where one or more of the targets being tracked would no more be in the radar range. In the first case we have to ensure if it is really a new target. The *New target Identification* sub-block 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* sub-block is responsible for dealing with such situations.

A new target is identified when an observation fails all the already established gates i.e. when all the elements of a row in the *Mask* matrix *M* are zero as shown by the matrix *M* in Figure 28. It shows that observation 2 failed all the existing three gates and hence it might be a new entry into the radar FOV.

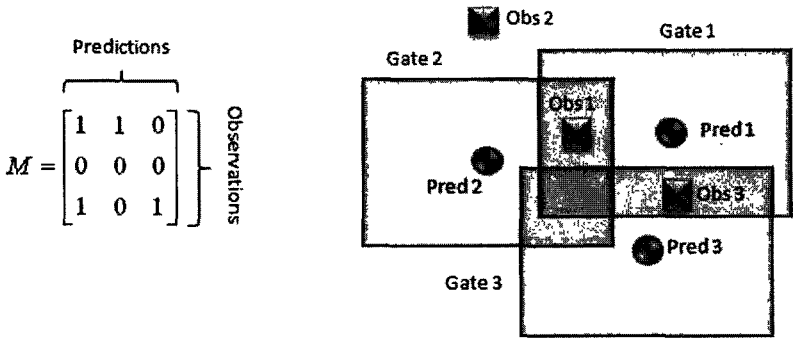


Figure 28: Observation 2 does not fall into any gate

Such observations are potential candidates for initiating new tracks after confirmation. The confirmation strategies we use in our work are based on empirical results cited in (44) and (46). In context of 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, in a scan, 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 as shown by the matrix M in Figure 29. This example shows that the gate around prediction 3 doesn't have any observation in it for the current scan.

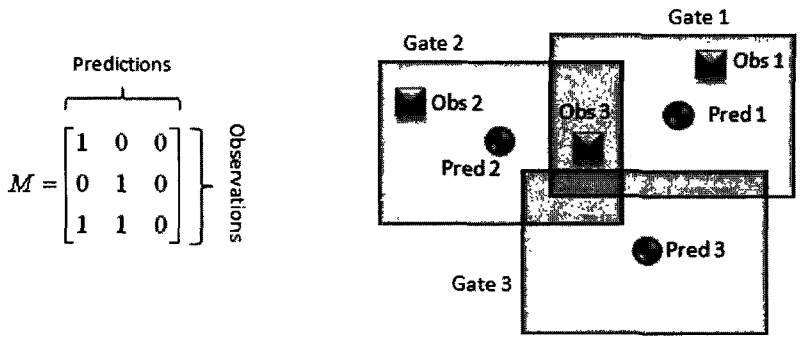


Figure 29: Gate 3 goes without any observation falling in it

The tracks for such targets have to be deleted after confirmation of their disappearance. The disappearance is confirmed if the concerned gate goes 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, the disappearance of the target is confirmed and its track is deleted from the system. The counter is reset every five scans.

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

4. Chapter Summary

In this chapter we focused on the application design and the mathematical modeling aspects of our embedded MTT system. We described the general concepts of tracking systems and explained the meaning of the terms and definitions used in the literature. We described the general principles of tracking by starting single tracking system (STT) and demonstrated how multiple target tracking is different and more complicated than the STT. We also explained the necessity to use MTT and not STT for automotive safety applications.

After discussing the salient features of a text-book MTT system, we moved on to the specifics of our own MTT application design. We explained how the application is broken down into individual components for modular implementation. We described the function of each module of the application and explained the interrelations among them. We developed a mathematical model for each module of the application. We used illustrations and flowcharts to make the models easily comprehensible. In the next chapter we look into the software development and software to hardware mapping aspects of the system.

4

Application to Architecture Mapping

In this chapter we discuss the hardware architecture for the MTT system. We discuss hardwired and MPSoC based system implementations and we argue that the MPSoC architecture is more suitable for our application than a hardwired architecture. After presenting some statistics about the profile of the application, we describe the application-to-architecture mapping and propose a preliminary MPSoC architecture for the system. The proposed architecture is not yet optimized and serves as a starting point for a more refined and optimized system that is the subject of the next chapter.

1. Introduction

In the previous chapter we discussed our approach to the design and modeling of the MTT application. After developing the application model we are now ready to move on to the architectural aspects of the co-design flow. This chapter focuses on the hardware aspects of the co-design flow. Figure 30 highlights the two stages of the co-design methodology that we investigate in this chapter. First we describe the architectural aspects of the system. Then we explore the mapping of various application tasks to the architectural elements of the system.

In section 2 we discuss the possible system implementation choices and their pros and cons. Section 3 describes the implementation platform, the design environment and the associated tools. In section 4 we present a brief introduction to the NiosII soft-core processor which is used as the main building block for the MPSoC architecture. The software development structure of the application is described in section 5. Section 6 gives a view the system software

used in the implementation. In section 7 the software profiling is presented where we lay emphasis on the application runtime statistics and memory requirements. This section plays an important role in determining the right processor configuration and memory subsystem for the application. Section 8 outlines the software to hardware mapping process. In section 9 the preliminary system architecture is presented. We summarize the chapter in section 10.

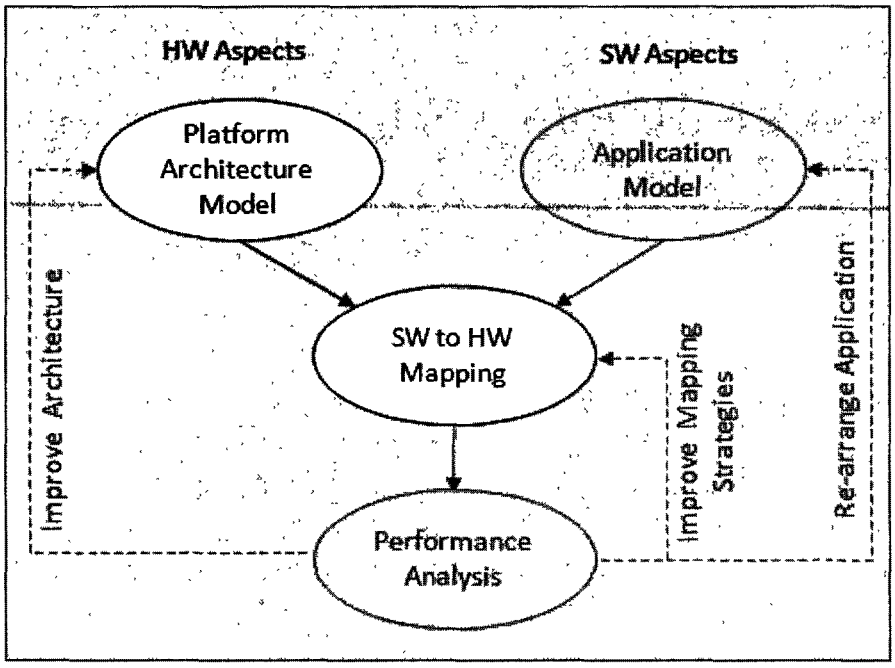


Figure 30: The Architectural Aspects of the Co-design Flow

2. Implementation Choices

In Chapter 1 and Chapter 2 we discussed the advantages of using FPGA as system implementation platform. When using FPGA as system implementation platform, we have two implementation choices. We can design a fully hardwired system using the FPGA fabric for the system hardware. Alternatively, we can code the application in software and map it to soft-core processors implemented in. Both these choices have their pros and cons.

Dedicated hardware for the entire application would process it with a high speed. While it is theoretically possible, there are some serious practical issues associated with such a move.

First, fully hardware implementations result in a large hardware size. Our experience with hardware implementation of a 2-state Kalman filter showed that it consumes about 29000 logic elements (LEs). This accounts for 48% of the logic available in the Stratix II 2S60 FPGA (56) that we use. Embedded systems need to be small sized, light and very efficient with respect to power consumption. FPGA hardwired systems are known for high power consumption proportional to their size and speed. Another demerit of the hardwired design is its lack of flexibility and scalability. A minor upgrade or debugging in the functionality of the application necessitates a complete revisit of the system hardware. The time required for dealing with such issues is prohibitively long and increases the system's time to market.

For implementing complex embedded applications, designers turn to soft-core processor based architectures. To meet the speed requirements of an application, multiple processors are often employed parallel execution of various application tasks. Hardware accelerators can be added up to share the load with the concerned processor for tasks where speed is absolutely critical. As a result, heterogeneous architectures incorporating functional units optimized for specific functions are commonly employed to design embedded systems. Power efficiency is another driving force having a strong impact on the architectures of embedded systems. The move to MPSoC design elegantly addresses the power issues faced on the hardware side. Using multiple processors that execute at lower frequency, results in comparable overall performance in terms of instructions per second while allowing designers to slow down the clock speed. Clock speed is a major constraint for low power designs (57). FPGA based MPSoC architectures are flexible and scalable. Hardware or software upgrades and modifications to the FPGA based MPSoCs are quick and with negligible NRE cost.

For the reasons stated above, we preferred to use an FPGA based MPSoC architecture using soft-core IPs to meet the performance and flexibility requirements of our system.

3. Implementation Platform and Design Environment

We used Altera's Nios II development kit (Stratix II 2s60 edition) as the implementation platform. This choice is driven by the fact that at the time when we started our work, this kit best matched the requirements our application. The Sratix II 2S60 FPGA (56) contains 48,352 ALUTs (60,000 Logic Elements) available to the designer for system implementation. The LE is the smallest unit of logic in Altera FPGA architecture.

Each LE features the following components (58):

- A four-input look-up table (LUT), which is a function generator that can implement any function of four variables
- A programmable register
- A carry chain connection
- A register chain connection
- The ability to drive all types of interconnects: local, row, column, register chain, and direct link interconnects
- Support for register packing
- Support for register feedback

An adaptive look-up table (ALUT) is the cell used in the Quartus II software for logic synthesis for Stratix II and later device families. It is equivalent to about 1.25 LEs (59) .

Altera provides its Nios II soft-core embedded processors and the accompanying Embedded Design Suite (EDS) with its development platforms. The EDS offers a user friendly interface for designing Nios II based processor systems. A library of ready to use peripherals and customizable interconnect structure facilitates creating complex systems quickly. The EDS also provides a comprehensive API for programming and debugging the system. The profiling tools included in the EDS help identify bottlenecks and critical points in an application.

Altera's embedded design suite (EDS) consists of the tools shown in Figure 31. The system design starts with creating a Quartus II project. Once the project is created, we invoke the SOPC Builder tool from within Quartus II. The SOPC Builder tool is used to instantiate pre-designed components and IPs and to interconnect these components. We can choose processors, memory interfaces, peripherals, bus bridges, IP cores, interface cores, common microprocessor peripherals and other system components from the SOPC Builder IP library. We configure the base addresses, the interrupt request priorities (IRQs), for the instantiated components.

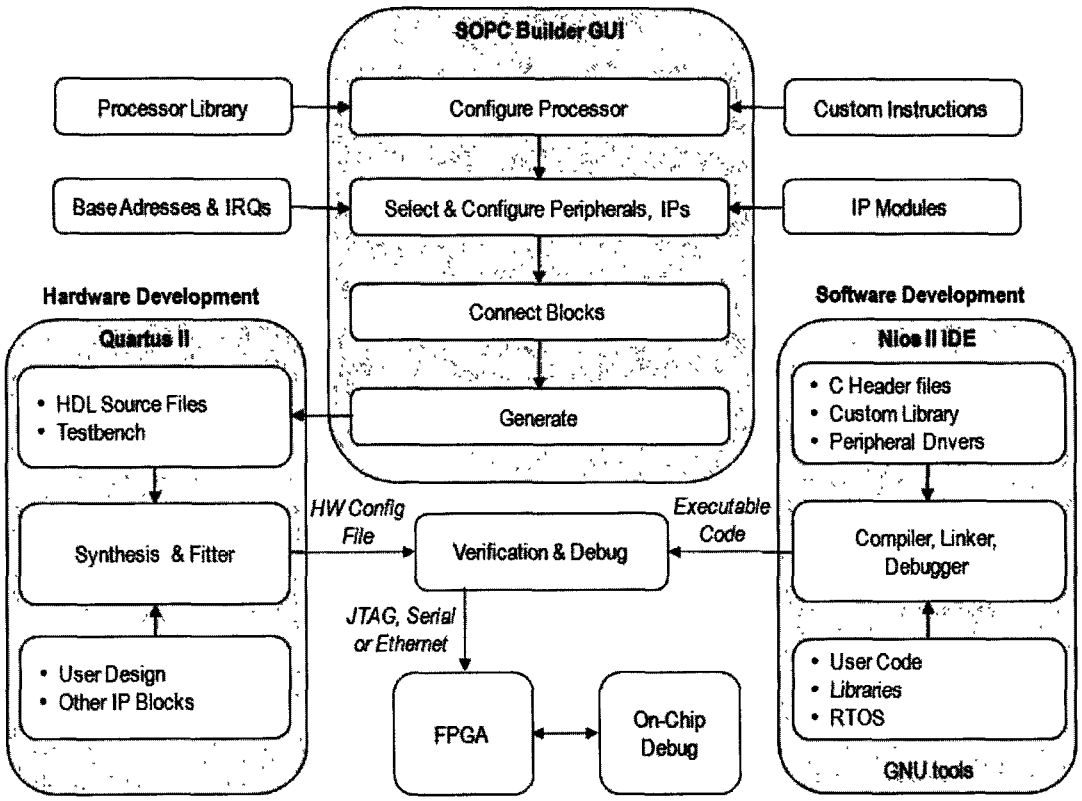


Figure 31: Altera's Embedded Design Suite (EDS)

The SOPC Builder produces a corresponding a software environment to develop applications. The software environment matches the target hardware including header files, custom libraries (peripheral routines), and design-specific operating system (OS) kernels etc. Once the system integration is complete, RTL code is generated for the system. The generated RTL code is sent back into the Quartus II project directory where it is synthesized, placed and routed and finally an FPGA is configured with system hardware.

After having the FPGA configured with a NiosII based hardware, the we download the relevant application tasks to the processor(s) in the system. This is done in the NIOS II IDE. The IDE manages NiosII C/C++ application and system library or board support package (BSP) projects. The system library includes all the header files and drivers related to the system hardware components.

4. The Nios II Processor

The Nios II (60) processor is a general-purpose RISC processor core. It is a configurable soft-core processor, as opposed to a fixed, off-the-shelf microcontroller. We can add or remove features on a system-by-system basis to meet performance goals. The main architectural features of the Nios II processor are illustrated in Figure 32.

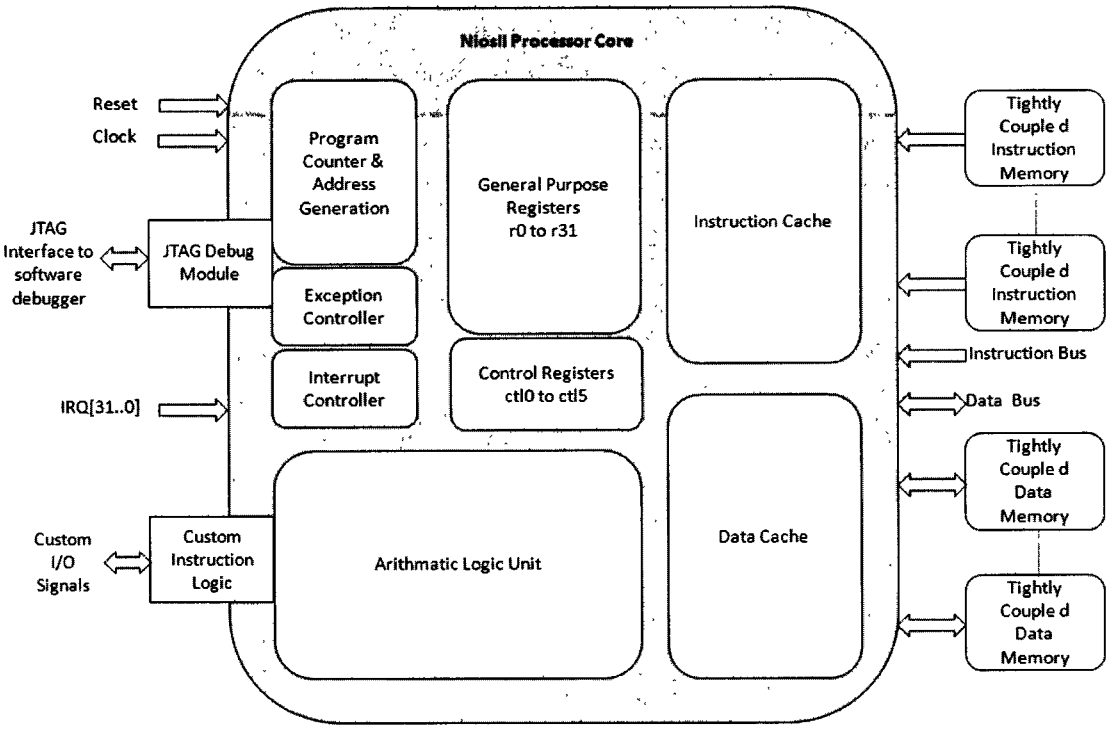


Figure 32: The Nios II Arcitecture

The processor architecture supports a register file, consisting of thirty two 32-bit general-purpose integer registers, and up to thirty two 32-bit control registers. There are no floating-point registers provided in the Nios II architecture. It supports single precision floating-point instructions as specified by the IEEE Std 754-1985. These floating-point instructions are implemented as custom instructions. NiosII custom instructions are custom logic blocks adjacent to the ALU in the processor's data path. Using custom instructions, the designer can reduce a complex sequence of standard instructions to a single instruction implemented in hardware. The NiosII software development tools recognize 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 compiles the code to use the custom instructions for floating-point operations, including addition, subtraction, multiplication, division and the newlib math library (61). 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 Nios II architecture supports separate instruction and data buses, classifying it as Harvard architecture. The data master port connects to both memory and peripheral components, while the instruction master port connects only to memory components. Both the Instruction Master Port and the Data Master Port can be cached. The caches are direct mapped; low address bits represent cache lines. The caches follow the write through policy, data is written into external memory as well as the cache.

The processor can prefetch sequential instructions and perform branch prediction. The instruction master port always retrieves 32 bits of data. It relies on dynamic bus-sizing logic contained in the system interconnect fabric. By virtue of dynamic bus sizing, every instruction fetch returns a full instruction word, regardless of the width of the target memory.

The NiosII processor comes in three customizable implementations. These implementations differ in the FPGA resources they require and the speeds they can achieve. NiosII/e is the slowest and consumes the least logic 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 speed and logic resource requirements. Table 1 shows the salient features of the three implementations of the NiosII core.

Table 1: Different NiosII implementations and their features

Processor / Description Features	Nios II / f Fast	Nios II / s Standard	Nios II / e Economy
Pipeline	6 Stage	5 Stage	None
H/W Multiplier and Barrel Shifter	1 Cycle	3 Cycle	Emulated in SW
Branch Prediction	Dynamic	Static	None
Instruction cache	Configurable	Configurable	None
Data Cache	Configurable	None	None
Logic Usage (Logic Elements)	1400-1800	1200-1400	600-700

The code written for one implementation of the processor runs on any of the two other implementations with a different execution speed. Hence switching from one processor implementation to another requires no modifications to the software code.

5. Structuring Application for Parallel Mapping

MPSoC architectures draw their strength from the parallelism in the application. Parallelism can be exploited at low level (fine-grain parallelism) or at high level (coarse-grain). Low level refers to parallelism at the operand level, i.e. where parts of a statement can be executed concurrently. High-level parallelism refers to different tasks being executed concurrently. To take advantage of the high level parallelism, the application is broken down to individual independent parts or tasks that can run in parallel. Then these tasks are mapped to the processors customized to their individual performance needs. It is important to balance the load on all the processors to ensure that all the processors work concurrently rather some working while others wait in an idle state. While dividing the application into tasks, the computational independence within the tasks must be taken into consideration. By computational independence we mean whether a task is self contained or it is dependent on other tasks for its computational needs. The former helps obtain better performance by overlapping computation and by minimizing the communication overhead. In the latter case, the task has to exchange data with other tasks and has to wait for inputs from them. Therefore such tasks cannot be executed in parallel because of the dependencies among them.

We divided the application into independent software routines which compute individual application tasks locally with minimal inter-task communication. The software structure follows directly from the system's mathematical model explained in the previous chapter. We first simulated the application in Matlab to verify that it works correctly. When the correct functionality of the application was verified, we coded it in ANSI C. Figure 33 shows software structure of the application and the interrelation among the tasks. It can be seen in the figure that most of the computation is contained within the tasks. There are no half-cooked results exchanged among the tasks to ensure minimum inter-task communication.

The Kalman Filter (KF1) uses Matrix Multiplication (Mat Mul), Matrix Addition (Mat Add), Matrix Subtraction (Mat Sub), Matrix Inversion (Mat Inv) and Matrix Transposition (Mat Trans) sub-functions. All the operands in these functions are floating point numbers. For processing all the targets in parallel, as many filters are required as the number of targets being taken into account. In Figure 33, for processing 20 targets, 20 filters are shown in the software

model. The Kalman Filters provide predicted states and prediction error covariance to the Gating Module. The filters calculate the estimated target states as the system output.

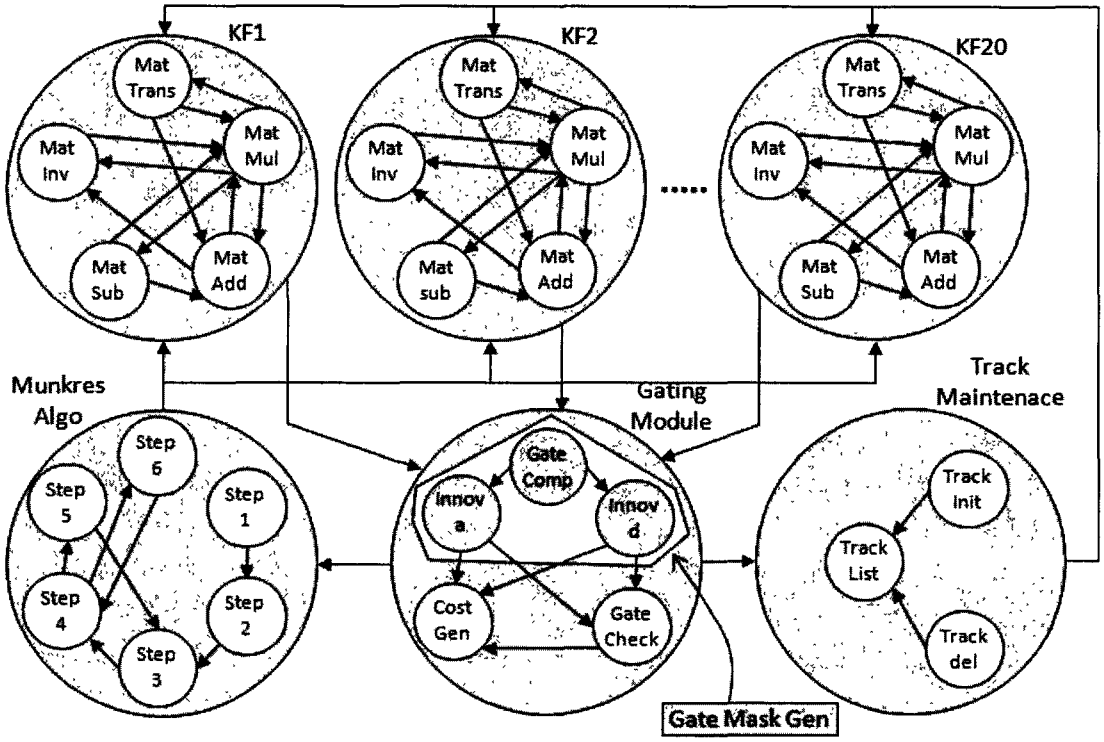


Figure 33: Application task structure

The Munkres Algorithm consists of six sub-functions (Step 1 through Step 6). Functions Step3 through Step6 are invoked iteratively for finding the final solution. The sub-functions exchange several floating point matrices and vectors between them. Every function evaluates the data passed on to it and performs certain operations on them before passing them on to the next function or back to the sending function depending on the conditions. The algorithm exits at Step 3 when the solution is found. The Munkres algorithm receives its input from the Gating Module and its output goes back to the Kalman filters.

The Gating Module outputs the Cost Matrix to the Munkres Algorithm and the Mask Matrix is tapped by the Track Maintenance module. The *Innov_d* and *Innov_a* calculators are two sub-functions used to calculate the distance and angle innovations (residuals) respectively, for all the observations in a scan. They involve subtraction operations on floating point numbers for every scan. The *Gate Computation* (Gate Comp) function though shown separately, is distributed between the Innov_d and Innov_a functions. The Gate Mask Generator code is also put into the same function as the two innovation calculators and the Gate Computation tasks. The

Gate Checker evaluates all the innovations for association with existing predictions and screens out the improbable associations. This involves comparison operations for the distance and angle measurements with their respective predicted values. The Cost Matrix Generator (Cost Gen) function is grouped with the abovementioned functions to form the upper level Gating Module.

The Track Maintenance (Track Maint) contains three sub functions; Track Initiate (Track Init), Track Delete (Track Del) and Track List Update (Track List). The Track Maintenance module sends the updated track list out to the Kalman filters.

6. System Software

Processor based systems use two types of software, the application software and the system software. We described the structural details of the application software above. In this section we present a brief description of the system software we use.

Many Nios II systems have simple requirements where minimal operating system or small footprint system software such as Altera Hardware Abstraction Layer (HAL) or a third party real-time operating system is sufficient. We use the HAL as the system software for our design, firstly because its memory footprint is small and secondly we prefer independence of any third party tools. Moreover an operating system not based on HAL requires a Memory Management Unit (MMU) to work correctly. An MMU can only be used with a Nios II/ F processor. This means that to use a full-featured real time operating system, every processor in the system has to be a Nios II/F each with an MMU. This would increase the size of the hardware unnecessarily.

The HAL, illustrated in Figure 34, is a lightweight runtime environment that provides a simple device driver interface for programs to communicate with the underlying hardware. The HAL application program 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 resistant to changes in 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 (62).

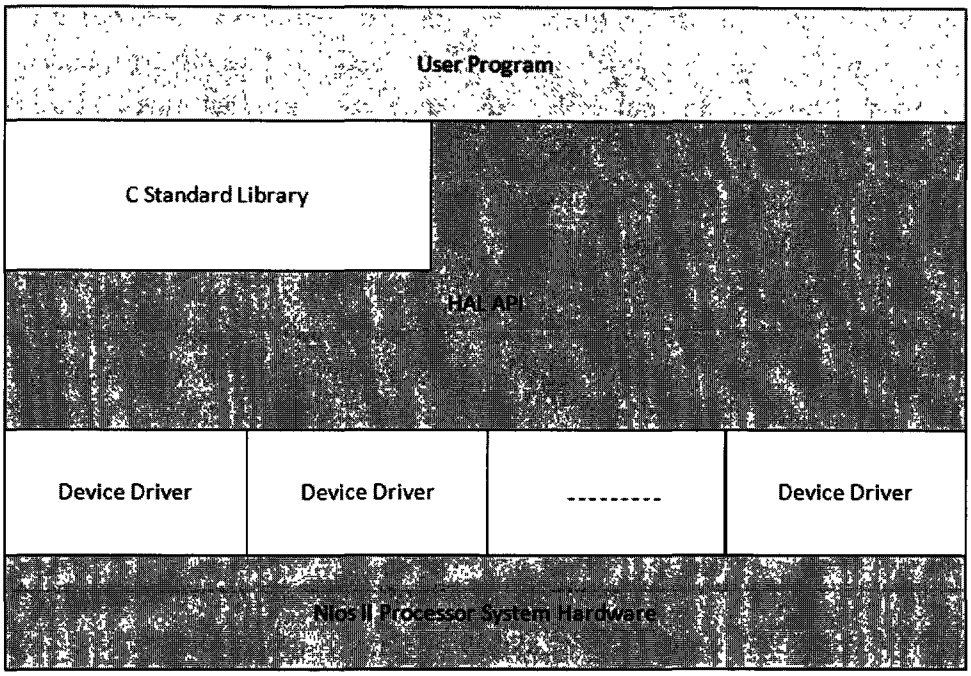


Figure 34: The Hardware Abstraction Layer (HAL) Structure

7. Application Profiling

To map the software to the hardware in a systematic way and to be able to optimize the resulting architecture latter on, the application must be profiled. Before deciding to allocate processing resources to the application modules, we profiled the application to measure the latencies, resource requirements of each software module and dependencies among the modules. Two tools namely the *GProf* and Altera's *Performance Counter* (63) peripheral are provided by Altera for profiling applications. Both these two tools have their merits and demerits.

The Gprof profiler tool called *nios2-elf-gprof* can be used without making any hardware changes to the Nios II system hardware. This tool directs the compiler to add calls to the profiler library functions into the application code. The profiler provides a crude overview of the run-time behavior of the entire system and also reveals the dependencies among application functions. However adding instructions to each function call for use by the GNU profiler affects the code's behavior in numerous ways. Each function becomes bulkier because of the additional function calls to collect profiling information. Collecting the profiling information increases the entry and exit time of each function. Pulling the profiling function into instruction cache memory

generates more instruction-cache misses than are generated by the original source code. The *Gprof* can be used to profile the entire system; the user cannot designate specific sections of the code for profiling. The *Gprof* tool is more suited for understanding interdependencies among the functions in the application rather than the knowing the exact runtimes of the individual functions.

The second method of profiling the application is the use of a performance counter peripheral provided in Altera's IP library. The 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 upto seven code sections. A pair of counters tracks each code section. A 64-bit time (clock-tick) counter counts the number of clock ticks during which 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 the designated sections of C/C++ code. Performance counters are best suited for measuring the latencies of the application modules more accurately.

We used both of the tools to have a thorough understanding of the behavior of an application. We used the *GProf* primarily for determining interdependencies among the application tasks and the *Performance Counter* peripheral for measuring accurately the execution time of each task. The relevant results of the profiling are discussed in the following sections.

7.1. Application Runtime

The runtimes of the various application tasks obtained through profiling are summarized in Table **2** through Table 4. All the latencies, cited in these tables, are obtained by running the application modules on a NiosII/s processor (cf. Table 1) with 100MHz clock, 4KB instruction cache and using an off-chip SDRAM device. We use this configuration as a baseline for executing all the modules of the application. This allows us to observe the relative speeds of the modules on a *fixed platform*. Later on, we adapt the configurations individually for each of the processors according to the requirements of the respective modules they are running. The latencies shown here offer an intuitive insight into the runtime behavior of the application and from the basis of our mapping and optimization strategies.

Table **2** shows the breakdown of the execution time taken by the Kalman filter. The top level function is indicated by *KALMAN* and the other functions are those called by the *KALMAN* for the filtering process. The main contributor to the execution time is the *Matrix Multiplication* sub function. There are two reasons for this; first, because this function involves multiplication operations on the floating point numbers and second, it is invoked eleven times for a single iteration of the filter. The global runtime for the filter is 15.146 ms. In its present non-

optimized form, for processing 20 targets in parallel we replicate the filter 20 times to meet the realtime requirements of the application.

Table 2: Runtimes of the Kalman Filter Functions

The Kalman Filter	
Function	**Runtime in ms**
KALMAN (top level Function)	15.146
Matrix Addition	1.060
Matrix Multiplication	11.871
Matrix Subtraction	0.824
Matrix Transposition	0.841
Matrix Inversion	0.376

Table 3: Runtimes of the Munkres Algorithm Functions

The Munkres Algorithm	
Function	**Runtime in ms**
Call to Mukres (top level Function)	147.197
Step 1	4.266
Step 2	1.396
Step 3	8.519
Step 4	61.221
Step 5	18.986
Step 6	51.850

Table **3** lists the runtimes for the Munkres algorithm and its sub-functions. The two sub functions that contribute the most to the overall execution time are *Step4* and *Step6*. These two sub functions are called by the top level function repeatedly in a nested loop. Step5 is the third most costly sub function with respect to execution time. We shall concentrate on these sub functions in the optimization process discussed in the next chapter.

The *Gating Module* runtime and those for its sub-functions are given in Table 4. The *Cost Mat Gen* (Cost Mat Gen) represents the runtime of the whole Gating Module. The other functions are sub-functions called repeatedly by the Cost mat Gen to construct the cost matrix. The *Innov_d* and *Innov_a* calculators involve operations on floating point numbers which are repeated 400 times for every scan. Hence they take 135ms and 113 ms respectively to complete these operations. The *Gate Checker* involves only comparison operations for the distance and angle measurements hence the relatively shorter execution time. The *Gate Mask Generator* makes up the higher level function for the two *innovation calculators* hence it includes the time taken by them. The *Cost Matrix Generator's* execution time 291.818 ms, is roughly the sum of the execution times for the *Gate Checker* and the *Gate Mask Generator*.

Table 4: Runtimes of the Gating Module Functions

The Cost Matrix Generator	
Function	**Runtime in ms**
Cost Mat Gen (top level Function)	291.818
Gate Mask Generator	258.356
Innov_d calculator	135.592
Innov_a calculator	113.178
Gate Checker	30.989

These sub functions are regular and repetitive which makes their behavior predictable. In the next chapter we shall see that these characteristics make these sub functions tractable and easy to optimize.

The contents of the three tables discussed above show the runtimes of the application tasks and their sub functions. The breakdown of the runtimes helps us concentrate on the most time consuming sub functions of the tasks during the optimization process. The above discussion

also highlights the intricate interrelations among the sub functions of the tasks. This is the reason we keep these sub functions together within the mentioned tasks. Processing resources are allocated to the tasks according to their needs. Memory allocation, discussed next, is one of these needs.

7.2. Memory Requirements

To be able to appropriately allocate memory to various application modules, we estimated their memory requirements. Although these estimates were obtained using the specific Nios II platform, they are not necessarily dependent on that platform. The memory requirements will still be the same if we use another platform. We also estimated the memory used by various sections of the code for each module. We determined the amount of memory required by the *.text* section, *.data* section, the *stack* section and the *heap* section of the program. Typically, the .text section is the part of the object module that is reserved for the program instructions. The .data section contains initialized static data e.g. in C, initialized static variables, string constants etc. The heap section is used for dynamic memory allocation. The stack section is used for holding the return addresses (program counter) when function calls occur. As we shall see in the next chapter, the memory sizes of these sections play an important role in the optimization process of the system. Table 5 summarizes the memory requirements of various modules.

Table 5: Memory requirements of various application modules

Code Section	Memory Foot Print
Kalman Filter	
Whole Code + Initialized Data	81 KB
.text Section	69.6 KB
.data Section	10.44 KB
The stack Section	approximately 2 KB
The heap section	approximately 1 KB

Munkres Algorithm	
Whole Code + Initialized Data	62 KB
.text Section	52.34 KB
.data Section	10.44 KB
The stack Section	approximately 2KB
The heap section	approximately 1 KB
Gating Module	
Whole Code + Initialized Data	63 KB
.text Section	51.81 KB
.data Section	8.61 KB
The stack Section	approximately 2 KB
The heap section	approximately 1 KB

The memory footprint of the application plays an important role not only in the execution speed of the application but also in the size of the system hardware and consequently in the power consumption. Furthermore, understanding the memory requirements helps design an optimum memory subsystem. Knowing the memory requirements allow us to decide on issues like memory system topology, dedicated or shared memory, the amount of on-chip and off-chip memory etc. These considerations will be explored in the next chapter.

8. Software to Hardware Mapping

With latencies and memory requirements of the application tasks known, we can now assign the tasks statically to different processing resources. Figure 35 shows the preliminary mapping scheme.

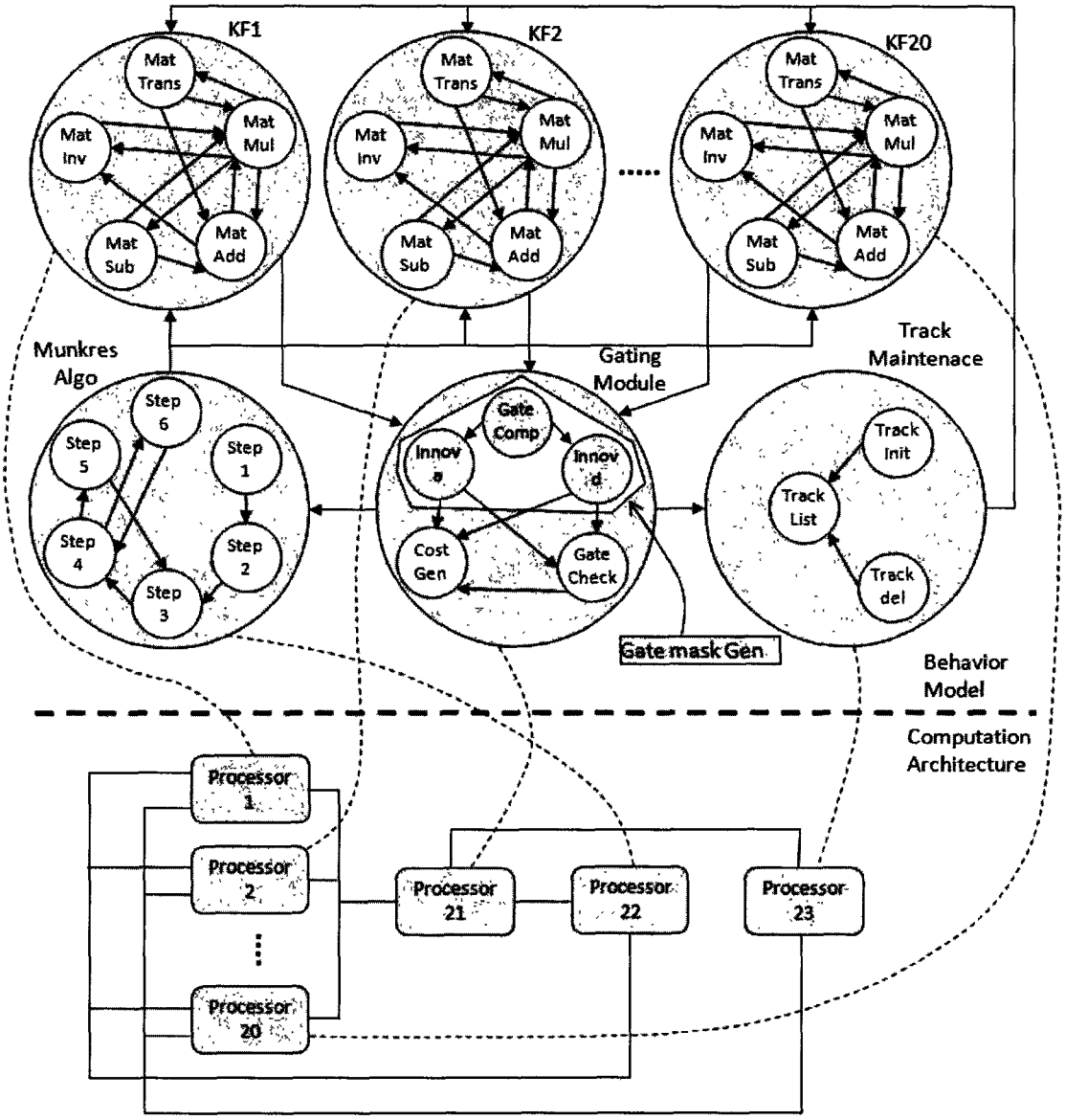


Figure 35: Software-to-Hardware Mapping

This choice of mapping is motivated by several factors. We want to run the application tasks on separate processors to ensure as much task level parallelism as possible. We keep the sub functions in the tasks together to minimize inter-processor communication. However, we don't want some of the processors overloaded while others are under-loaded. To ensure the load balance among the processors we optimize the architecture as well as the application as discussed in the next chapter.

Our initial analysis (cf. Table 2) shows that it takes 15.146 ms to execute the filter on a standard Nios II processor with 4KB instruction cache. To take a maximum of N targets into account, the Kalman filter is replicated N times. All the N filters process the targets in parallel. Assuming, for the moment that a standard Nios II would run the filter, we allocate N identical processors executing the Kalman filters in parallel in the preliminary architecture.

Obviously this proposition has to be optimized to be viable. We will discuss the optimization strategies for the filter in the next chapter.

The Munkres algorithm consists of six distinct iterative steps. It takes 147.197 ms (cf. Table 3) on a standard Nios II processor to loop through these steps for all the targets. Hence the Munkres algorithm cannot be combined with any of the other functions. In the preliminary architecture we assign the Munkres algorithm to a separate processor. Here again we have to accelerate the processing to achieve higher speed.

The sub functions of the Gating Module (innov-a, innov_d, gate mask gen, gate check) are always in constant communication with one another. So we group these three blocks together and map them onto a single processor to minimize inter-processor traffic and communication delays. Avoiding excessive inter-processor communication is desirable for reducing hardware and software complexity as well as for energy conservation.

The three blocks of the *Track Maintenance* sub functions (track init, track del, track list) individually don't demand heavy computational resources, so we grouped them together for mapping onto a processor.

Table 6: Summary of the preliminary architectural components

Module Name	Number of Processors	Processor Type	I-Cache Size	D-Cache Size	Local Memory	Run time
Kalman Filters	20	NiosII/s	4KB	0	81KB	15.146 ms
Gating Module	1	NiosII/s	4KB	0	63KB	291.81 ms
Munkres Algorithm	1	NiosII/s	4KB	0	62KB	147.19 ms
Track Maintenance	1	NiosII/s	4KB	0	0	1.5 ms

Table 6 summarizes the assumed contents of the preliminary architecture. At this stage the number of processors, processor types, the cache sizes and local memory sizes are only assumed based on the initial analysis presented above. In the next chapter we shall present a summary the finalized architecture after the optimization process.

One of our mapping requirements is to make the system scalable so that the level of performance is maintained as the number of processors and the problem size are increased. We believe that tracking 20 targets at a time are more than enough for automotive applications. However the need for tracking more targets can be easily incorporated into the system provided the sensor can handle more than 20 targets. All the application modules are parameterized so that increasing or decreasing the number of targets is as simple as modifying one parameter in the application header file. At the architecture level, all we need to do is to increase or decrease the number of filters accordingly. Since the filters run in parallel, changing the number of filters would not affect the overall system latency.

9. Preliminary System Architecture

The proposed multiprocessor architecture includes different implementations of the Nios II processor and various peripherals as system building blocks. We distributed the application over different processors as distinct functions communicating among them in producer-consumer fashion as shown in Figure 36. The architecture uses several Nios II processors as distinct function units for execution the main tasks of the application.

The system consists of Nios II 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. As can be seen in this figure, every processor has an Instruction cache (I-cache), a Data cache (D-cache), a local memory.

The local on-chip memories are intended for storing the performance critical sections of the code. The sizes of these memories depend on the requirements of the particular application module a processor is executing. The total memory requirement must be less than the available memory in the target FPGA. On-chip memories support dual port accesses, allowing two masters to access the same memory concurrently. The size and width of the memory are user-configurable. Since the on-chip memories are connected to the data master of the Nios II processors, we set the data width of the on-chip memory to 32 bits, the same as the data-width of the Nios II data master.

Since the execution time of the individual functions and their latencies to access a shared memory connected to a common bus are not identical, dependence exclusively on common 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 on-chip memory for every processor and a large

off-chip memory device shared by all the processors for non critical sections of the application modules. As a result the individual processors have lower latencies for accessing performance critical sections of the code located in theirs their local memories.

A Nios II processor core might include one, both, or neither of the cache memories. In the final system, the caches will be either I-cache and D-cache or D-cache only. The sizes of the caches and the local memories are not known at this stage. In the next chapter we shall demonstrate how to systematically determine the optimal sizes of these caches and the local memories. Besides, we will also show which processors must have both I-cache and D-cache and which ones need only D-cache in a latter section.

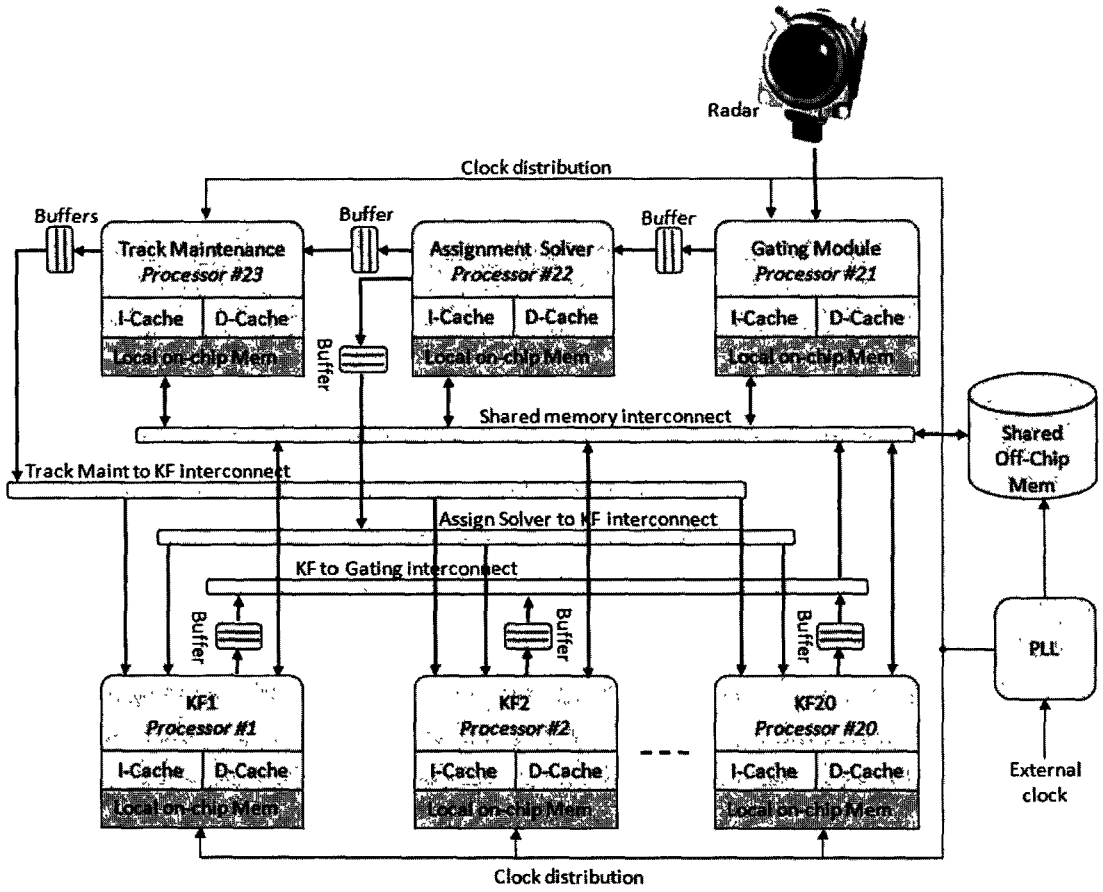


Figure 36: The Non-Optimized Initial Architecture

Every processor communicates with its neighboring processors through buffers allocated out of the on-chip memory blocks in the FPGA. This arrangement forms a system level pipeline among the processors. At the lower level, the processors themselves have a pipelined

architecture. 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 doesn't affect the accuracy because the data sampling frequency, as set by the radar PRT, is high enough to compensate for this minor loss.

The interconnect links shown in Figure 36, are not traditional buses. Four separate interconnects are used to ensure parallel processing. Using a single interconnect in a multiplexed fashion would serialize the inter-processor communication. The interconnect links are based on Avalon switch fabric. The Nios II 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. Slave-side arbitration moves the arbitration logic close to the slave where it determines which master gains access to the slave in the event that multiple masters attempt to access it at the same time. The slave-side arbitration scheme minimizes the congestion problems characterizing the traditional bus (64). Both data memory and peripherals are mapped into the address space of the data master port of the Nios II processors.

The architecture also includes an off-chip 32MB DDR SDRAM as shared memory connected to all the processors through the system-interconnect fabric. Access to the non critical sections of the code and data located in the off-chip shared memory is routed through an SDRAM controller core. Simultaneous accesses to the shared memory by more than one master are overseen by the slave-side arbitration logic.

The external oscillator on our board provides a 50 MHz clock. The PLL (phase-locked loop) is used for generating 100 MHz clock signals, distributing clock signals to different devices in the design and reducing clock skew between devices.

The architecture presented in Figure 36 not a definitive one. It will undergo modifications to reduce its hardware size, improve the processing speeds of various modules and optimize its on-chip memory requirements. In its current form its overall processing speed is too slow to meet the application realtime requirements. The cache sizes, local on-chip memories have not been optimized yet. After applying the optimizations discussed in the next chapter, the system is accelerated to meet the application realtime deadlines while at the same time its hardware size is reduced. Reduced hardware size is one of the main characteristics of embedded systems. Reducing the hardware size reduces the system energy consumption. Although the system is

expected to be powered up by the vehicle power supply, yet the consumed energy is dissipated in the form of heat which shortens the service life of the system.

10. Chapter Summary

This chapter dealt with the hardware aspects of the hardware/software co-design flow for system design. We reasoned that the MPSoC implementation is preferable over the hardwired implementation for our application. We presented a brief view of the Nios II soft-core processor which we are using as the processing element in the MPSoC. We described the FPGA platform, the design environment and the associated tools used for the Nios II based MPSoC.

We presented the software development model of the application and emphasized that application is divided into independent parallel software routines. Then we summarized application profiling tools and some presented selected results of the profiling. Application software profiling led us to the initial software to hardware mapping scheme. We proposed an initial architecture for the system built around the Nios II cores. In the preliminary architecture all the processors are supposed to be the standard Nios II/s implementations. The standard Nios II (Nios II/s) is the medium version of the implementations. The initial design assumes 23 Nios II processors in the architecture.

We proposed locally connected on-chip memories are provided to store the performance critical sections of the code for each processor. The optimum sizes of the on-chip local memories will be determined and discussed in the next chapter after optimization.

We proposed instruction and data caches for the processors. The caches sizes will be tailored up to the requirements of the particular needs of the application task a processor is executing. Some of the processors will have either one of the two caches or none at all in the finalized design. The non-critical sections of the code are stored in the off-chip shared SDRAM.

We discussed that the processors are connected to each other in a producer-consumer ring and they synchronize themselves through buffers. We described the interconnection structure of the architecture. The masters and slaves in the architecture are connected by the Avalon interconnect fabric. The fabric is a partial cross-bar structure where only those slaves are connected to a master to which it makes transactions. In case of simultaneous access by more than one masters to a shared slave, the interconnect fabric employs a slave-side arbitration logic to deal with congestion and contention.

The architecture presented here is not yet definitive. It will undergo improvements and modifications based on optimization results. Details of these optimizations are given in the next chapter.

5

Analysis and Optimization

Embedded system in general and FPGA based embedded systems in particular must have the minimum possible hardware sizes. Minimizing the hardware size is primarily desirable for accommodating the systems in a smaller FPGA thereby reducing the total system cost. Power consumption is also proportional to the hardware size of the system. However the system must also meet the application realtime deadlines. Therefore in the process of reducing the hardware size, the system processing speed must not be compromised beyond tolerable limits. The optimization strategies that we discuss in this chapter, aim at finding the right balance between the system hardware size and its processing speed.

1. Introduction

In the last chapter we discussed the application software structure and its mapping to the hardware architecture. We proposed a provisional architecture for the multiprocessor system. In this chapter we explain the final steps of the co-design flow (43). As depicted in Figure 37, we discuss the performance analysis and optimization aspects of the flow. In section 2 we discuss the constraints we have deal with while planning our optimization strategies. In section 3 we first outline the optimization strategies and then explain their significance one by one. Here we formulate strategies for speeding up the system while using the minimum possible computational resources. We apply these strategies in succession to all the modules of the application until we get the desired results. We deal with application tasks one by one and scrutinize them for potential

improvements. Section 4 we explain how the optimizations are applied to Kalman filter module. We clarify the reasons for continuing the optimization process even when the runtime of the filter is less than the radar PRT. In section 5 we use the optimization steps to customize the configuration of the processor for executing the *Gating Module*. Section 6 passes the Munkres algorithm through the optimization steps and introduces a new strategy for arriving at the final goal. Section 7 explains why the Track Maintenance module does not need to be optimized. In section 8 we conclude the chapter by illustrating the final optimized system architecture. We also summarize the final execution times of all the application modules and the architectural components of the complete system. We end the chapter by presenting the amount of FPGA resources used by each component of the architecture.

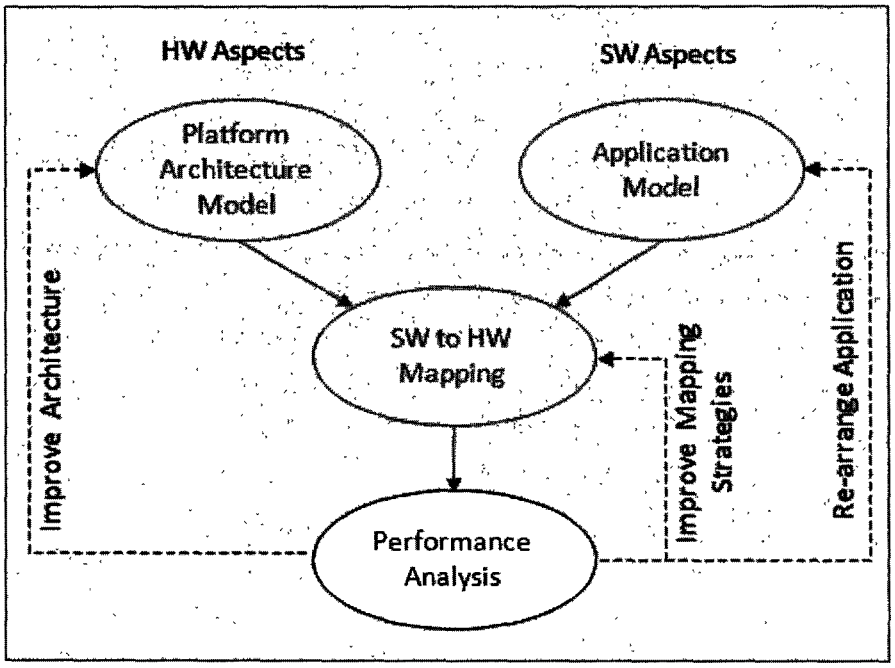


Figure 37: Performance Analysis and Optimization

2. Constraints

Using the profiling tools discussed in the preceding chapter, we analyzed the application to obtain necessary statistics about its performance. Since we require the system to be as compact

as possible, we aim to keep the sizes of the NiosII cores and their peripherals to the minimum and thereby minimize the usage of FPGA resources. These resources include the on-chip memory, the processor cache sizes and the configurable logic (LEs) etc. The initial performance analysis of the application modules shows that the realtime deadlines of the application cannot be met without optimization of the architecture, especially in the case of the Gating Module and the Assignment Solver. To be able to successfully optimize the system, we need to be aware of the constraints to be observed. The main constraints that we have to adhere to are the following.

- We need the overall response time of the system to be less than the radar PRT. The radar PRT for the radar unit we are currently using is 25ms. Since the application modules as discussed in the preceding chapter run in parallel, this means that the slowest application module must have less than 25ms of response time. Hence the first objective is to meet this deadline.

- The on-chip memory and the caches are allocated out of the memory blocks available on the FPGA. Both the on-chip local memory and the cache sizes have a tremendous impact on the execution speed of the system. However the StratixII EP2S60 FPGA (56), we are using for this system contains a total of 318KB 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. Consequently we have to find the right sizes of the on-chip local memories and the caches to optimize the hardware size as well as the execution speed. Accordingly we have to be vigilant not to overuse the on-chip memory and to keep the total on-chip memory requirement of the application below the 318KB limit. We can use off-chip memory devices but they are not only very slow in comparison to 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 acceptable limits nor the system becomes too slow to cope with the time constraints.

- We must choose our hardware components carefully to minimize the use of the programmable logic on the FPGA. The size of the system must not exceed the 48,352 LUTs (60,000 Logic Elements) available in the FPGA. Small hardware size is still desirable even if we have the liberty to use a bigger FPGA. Excessive use of

programmable logic not only complicates the design and consumes FPGA resources but also increases power consumption. For these reasons we optimize the hardware features of the processors and leave certain options out when they are not absolutely essential for meeting the time constraints.

3. Optimization Strategies

To meet the constraints discussed above, we laid out our optimization plan as follows:

- Select the appropriate processor type for each module to execute it in the most efficient way.
- Identify the optimum cache configuration for each module and customize the concerned processor accordingly.
- Explore the needs for custom instruction hardware for each module and implement the hardware where necessary.
- 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.
- Look for redundancies in the code and remove them to improve the performance.
- Investigate if the C2H compiler hardware can effectively accelerate certain functions. If so, use the C2H compiler.

The abovementioned optimization steps entail exploration of the "space" of all possible configurations of the MPSoC architecture and choosing the one that best meets the constraints. To reduce the time for the exploration, we tackle the hardware configurations of the processors individually one after the other. In the following pages we explain these strategies and their impacts on the system performance.

3.1. Choice of NiosII Implementation

The NiosII processor comes in three customizable implementations. These implementations differ in the FPGA resources they require and the processing speeds they can achieve. NiosII/e is the slowest and consumes the least logic 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 speed and logic resource requirements. However, the code written for one implementation of the processor will run on any of the other two implementations with a different execution speed. Hence switching from one processor implementation to another requires no modifications to the software code. The choice of the right implementations 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.

Another deciding criterion for selecting a particular implementation of the processor is the need for instruction and data caches. For example if we can attain the required performance for an application module without any cache, the NiosII/e would be the right choice for running that module. 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, we shall use NiosII/s for that module. The objective is to reach the desired speed with the least possible amount of hardware.

3.2. I-cache & D-cache

The NiosII architecture supports cache memories on both the instruction master port (instruction cache) and the data master port (data cache). The 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 need for higher memory performance (and by association, the need for cache memory) is application dependent. Many applications require the smallest possible processor core, and can trade-off performance for size. The cache memories are optional. 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. Both the Instruction and Data Cache sizes for NiosII/f can range from 0 to 64KB in discrete steps of 0, 2KB, 4KB, 8KB, 16KB, 32KB and 64KB.

Optimal cache configuration is application specific; therefore every processor has its own optimal instruction and data cache sizes. For example, if a 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 in a program is 2 KBytes, but the size of the instruction cache is 1 KByte, an instruction cache does not improve execution speed. In fact, an instruction cache may degrade performance in this situation (60).

In view of these assertions and the constraints discussed above we must choose the optimum instruction and data cache sizes that are necessary for achieving the desired performance for each application module. We analyzed the behavior of each module vis-à-vis the *Instruction Cache* and the *Data Cache* sizes. The objective here was to see how the performance changes with the changing sizes of the caches. We charted the runtime of each task with respect to different cache configurations to find where the bottlenecks lie.

3.3. Floating point custom instructions

Optimal caches can speed up the execution of an application module to a certain extent. Sometimes further speedup is required to meet execution time requirements of the module.

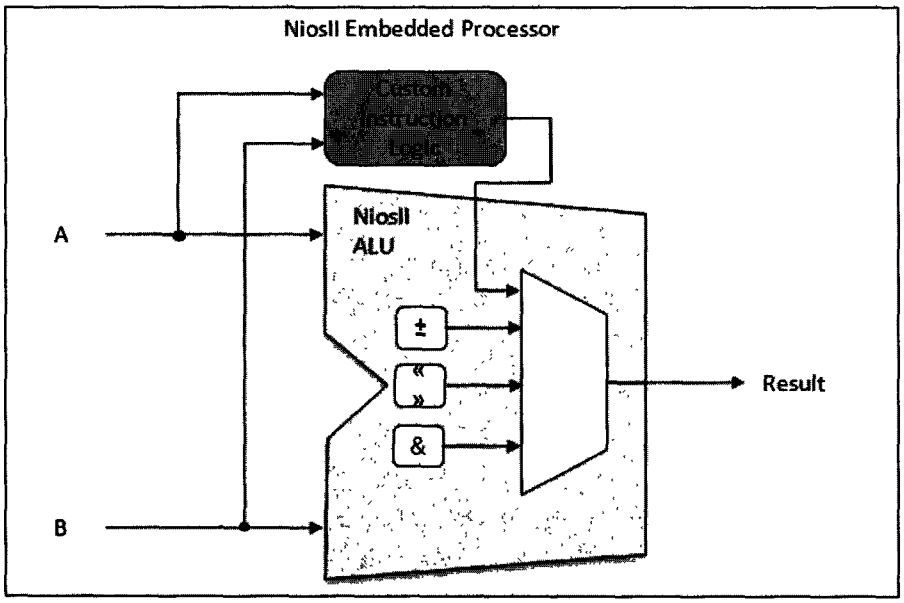


Figure 38: Nios II custom Instruction

Time-critical software algorithms can be accelerated by adding custom instructions to the NiosII processor instruction set. NiosII custom instructions are custom logic blocks adjacent to the ALU in the processor's data path. Using custom instructions, we can reduce a complex sequence of standard instructions to a single instruction implemented in hardware. The custom instruction logic connects directly to the NiosII arithmetic logic unit (ALU) as shown in Figure 38.

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 set of custom instructions is available on all NiosII cores. 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 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 compiles the code to use the custom instructions for floating-point operations, including addition, subtraction, multiplication, division and the *newlib* math library (61).

The best choice for a hardware design depends on a balance among floating-point usage, hardware resource usage, and 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, we rework the algorithms to minimize floating-point arithmetic.

3.4. On-chip Vs off-chip memory sections

In an ideal situation we would keep all the memory sections of a program in the fast on-chip memory. From Table 5 in Chapter 4, we can see that the memory footprints of the whole code and the *.text* sections for all the modules are too large to be accommodated in the on-chip memory. This implies that 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 which would further increase the memory access time. So, on 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 due to its scarcity. We have to strike a balance in our reliance on dedicated on-chip memory and the shared off-chip memory without compromising the performance too much.

By default, HAL-based systems are linked using an automatically generated linker script that is created and managed by the Nios II IDE. The linker script controls the mapping of code and data within the available memory sections. It creates standard code and data sections (*.text, .data,* and *.bss*), plus a section for each physical memory device in the system. Typically, the *.text* section is the part of the object module that is reserved for the program instructions. The *.data* section contains initialized static data e.g. in C, initialized static variables, string constants etc. The *.bss (Block Started by Symbol)* section defines the space for non initialized static data. The *heap* section is used for dynamic memory allocation e.g. when *malloc()* in C or *new()* in C++ are used. The *stack* section is used for holding the return addresses (program counter) when function calls occur.

In general, the Nios II design flow automatically specifies a sensible default memory partitioning. However, we wish to change the partitioning in special situations. For example, to enhance the performance we can place performance-critical code and data in the on-chip RAM with fast access time. In these cases, we have to manually specify which code belongs in which section.We can control the placement of *.text, .data, heap* and *stack* memory partitions by altering the NiosII *system library* or BSP settings. By default, the *heap* and *stack* are placed in the same memory partition as the *.rwdata* section. We can place any of these sections in the on-chip RAM if needed to achieve the desired performance. But due to the large footprint of the *.text* section we prefer it to be placed in the off-chip memory. However, by transferring the less memory-hungry sections into the on-chip memory, we can get considerable performance boost. For example if a certain module makes an abundant use of *malloc()* or *new()*, placing the *heap* section in the on chip memory can improve its speed by large margin. Similarly if a module makes frequent calls to other functions, putting the *stack* section in the on chip memory can help reach a higher execution speed for that module.

3.5. C2H Compiler

Altera provides a tool called C2H compiler (65) which is intended to transform C code into a hardware accelerator. 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 reasons discussed in section 6.5, we could use it only for the Munkres algorithm. But the tool can accelerate only a single function and that function too must not involve complex computations. So we could not use it for *step4* and *step6* of the algorithm where we needed it most. The tool simply stops working when we try to accelerate either of these two functions without producing any error message.

In case of small functions (like *step3* of the algorithm) 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 form it is far from its stated goals.

We applied the strategies described in sections 3.1 through 3.4, to all the modules of the application. In the following sections we discuss the results of these strategies.

4. Kalman Filter Optimization

Using the strategies outlined above for optimizing the Kalman filter, we first have to choose the right NiosII implementation for the filter. This choice depends on the cache requirements of the program. In the next subsection we discuss the cache requirements of the processors for executing the Kalman filter.

4.1. Cache Analysis

Figure 39 shows the influence of I-cache and D-cache sizes on the processor time of the Kalman filter algorithm running on NiosII/f processor with 100MHz clock and using off-chip RAM. It can be observed here that when D-cache size is increased from 0 to 2KB the execution time drops, but it remains almost unchanged beyond 2KB D-cache. The execution time drops slightly when the I-cache size is increased from 4KB to 8KB and it drops profoundly when the I-cache size is increased to 16KB. Beyond 16 KB I-cache, the execution time remains almost unchanged with increasing I-cache size. This indicates 16KB of instruction locality and 2KB of data locality in the filter code. Based on these observations, we can say that 16KB I-cache and 2KB D-cache are the optimum choices for the processors executing Kalman filters. However, for tracking a maximum of 20 obstacles we need 20 of these processors. Viewed in isolation, 16KB may not look much but we have to look at the global memory demands of the system. Compiling the system with a NiosII/f having 16KB I-cache and 2KB D-cache, we found that the total on-chip memory used by this configuration accounts for 7% of that available on our FPGA. Obviously replication of this system composition 20 times, is not feasible. Besides, we have to keep in mind that the other processors in the system also have on-chip memory requirements which have to be

met by the available on-chip memory. This calls for an optimization whereby we must reduce this excessive usage of on-chip memory.

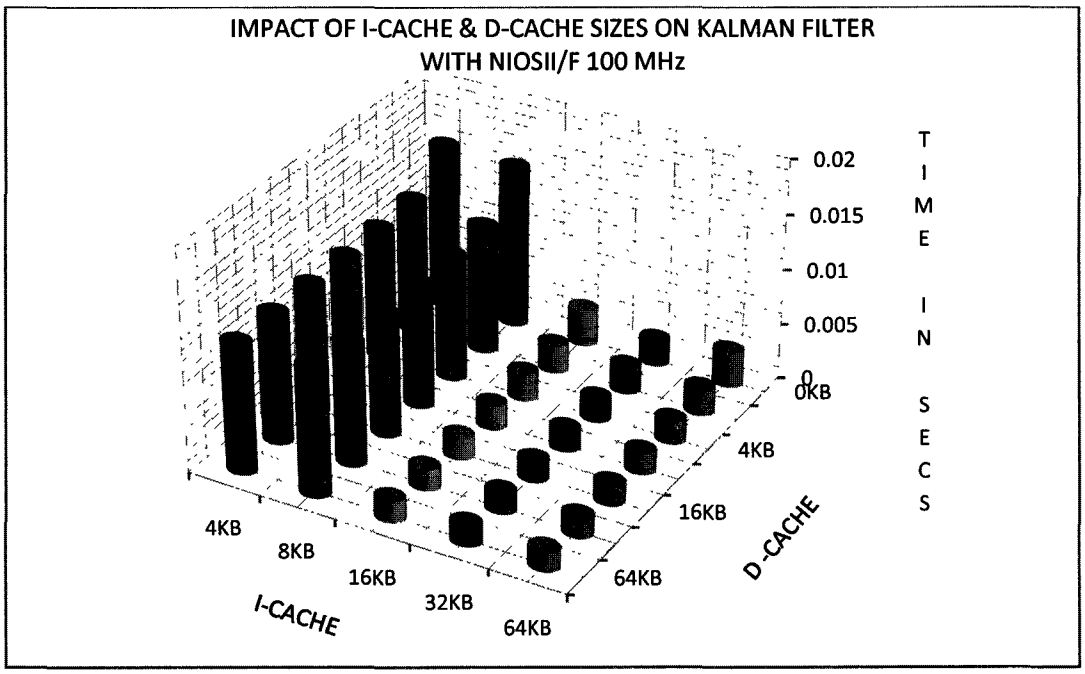


Figure 39: Cache behavior for Kalman Filter

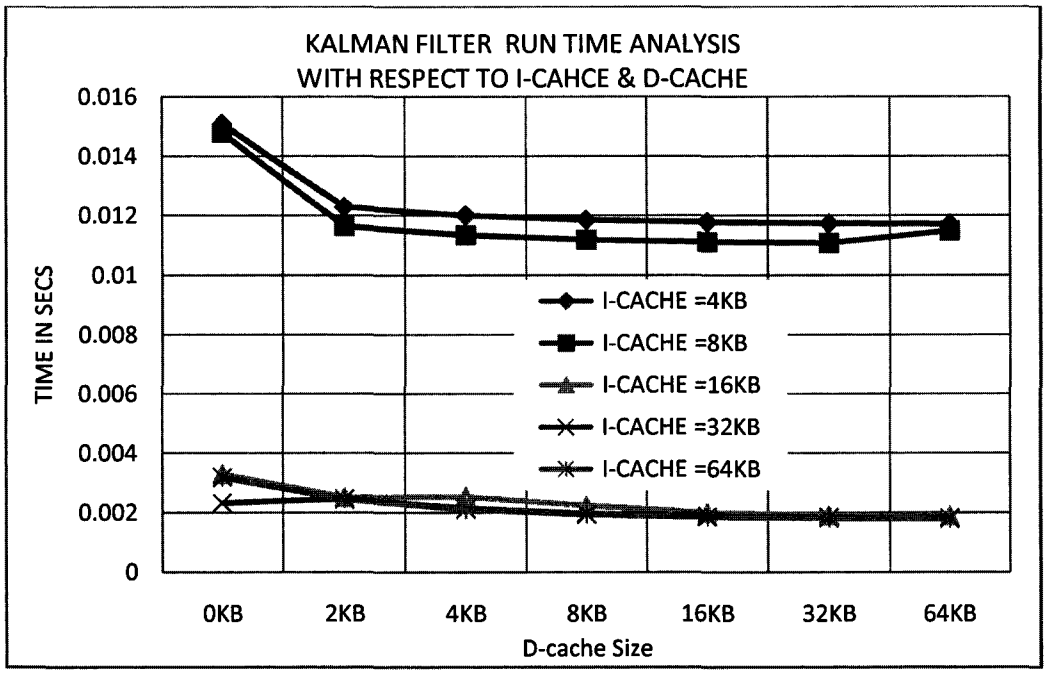


Figure 40: Kalman Filter Runtime Vs Cache Sizes

One way to bring about this optimization is to reduce the number of processors required to run the Kalman filters without reducing the number or targets being tracked. For this we need to speed up the processors so that more than one target can be treated by a single processor. The runtime behavior of the Kalman filter with respect to various cache configurations can be observed more clearly in Figure 40. Whatever the I-cache or D-cache size, the processor time never exceeds 15ms. Even with 4KB I-cache and no D-cache the processor time is below the 25ms threshold set by the radar PRT. On the contrary, if we use the optimal cache configuration the execution speeds up remarkably.

Figure **41** shows the performance of the Kalman filter on NiosII/s with 16KB I-cache, 2KB D-cache, 100MHz clock and using off-chip memory exclusively. Even with all memory sections in the off-chip device and using no floating point custom instructions, the run time is around 2.2ms. Hence, with the optimal configuration (16KB I-cache and 2KB D-Cache cf. Figure **41**) it is possible to reduce the number of processors for Kalman filters and thus conserve valuable FPGA logic as well as the on-chip memory.

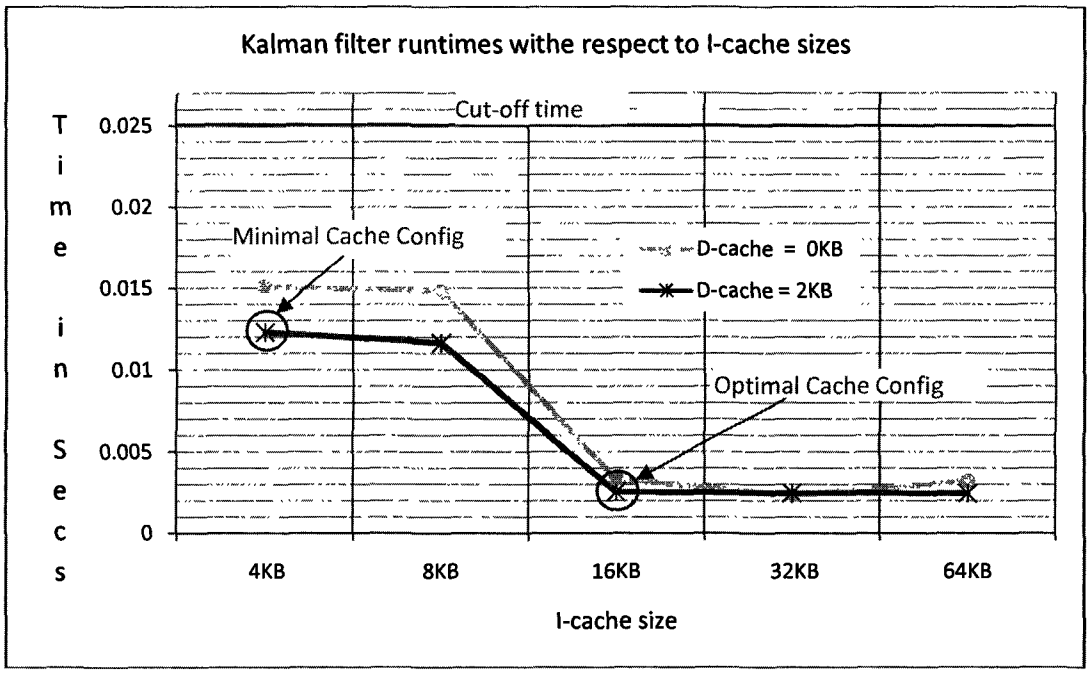


Figure 41: Kalman Filter performance with 16KB I-cache & 2KB D-cache

With 2.2ms processor time we can theoretically use 2 processors for 20 Kalman filters without exceeding 25ms time limit. But in practice we would have to add some further intelligence into the system to deal with this *housekeeping* of 10 filters on a single processor. This would, obviously, add some latency into the system and we would exceed the 25ms time limit for 10 filters. The hardware size per processor would increase but the gain in size due to the reduction in the number of processor will heavily outweigh this increase. Moreover, due to the reduced number of processors, a great deal of on-chip memory will be conserved. These issues merit further investigation and we do the same in the next two subsections.

4.2. Floating Point Custom Instructions

To be able to minimize the number of processors for executing the filters, we must accelerate the processing to the maximum possible limit. For this we tested the floating point custom instructions' impact on Kalman filter's performance. Figure 42 shows the results the tests.

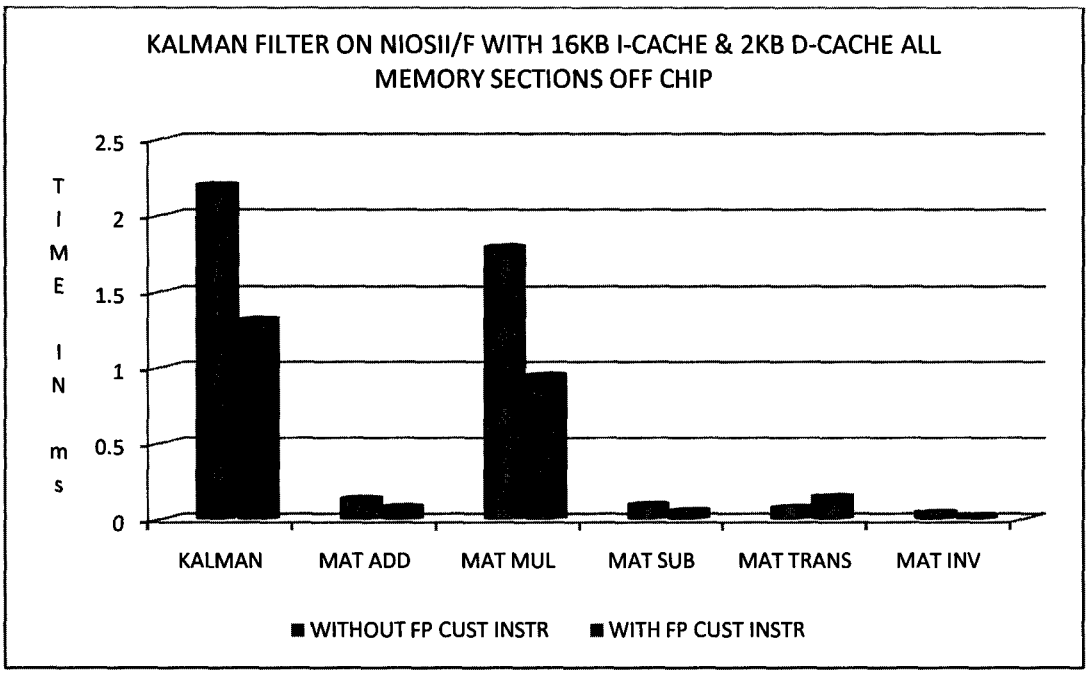


Figure 42: Kalman Filter performance with FP Custom Instructions

The overall runtime for the filter drops from 2.2ms to 1.3ms when floating point custom instructions hardware is used. This accounts for a speedup of 41% in comparison with the scenario where no floating point custom instructions are used. The most significant improvement is witnessed in case of the *Mat Mul* sub-function. This improvement can be attributed to two factors. One, *Mat Mul* relies heavily on floating point multiplication and second, it is invoked 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 speedup comes at the cost of a bulkier hardware. The usage of ALUTs increases by 8% when floating point custom instructions are used. With the 1.3ms runtime, we can now theoretically use 2 processors running 10 filters each in 13ms, ignoring the time for managing the *housekeeping* part. This multiplexing logic can become simpler if we could manage to use one processor for the filters instead of two. For this we would have to speed up the processing even further. In the following subsection we explain the procedure for achieving this speedup.

4.3. On-Chip Memory

We investigated the prospects of improving the runtime for the filter further through on chip memory placement. The outcome of this investigation is summarized in Figure 43. As before, *Kalman* represents the overall algorithm and the other bars are its constituent sub-functions. These results are obtained with 16KB I-cache and 2KB D-Cache.

Moving only the *stack* section to the on-chip memory reduces that runtime from 2.2ms to 1.9ms without using the floating point custom instructions and from 1.3ms to 1.01ms with the floating point custom instructions. Since the stack section of the memory requires only 2KB, the cost in terms of memory size for this speedup is nominal. We achieve this speedup by connecting 2KB of on-chip dedicated memory to the processors for the stack. Now we can use only one processor for 20 filters with time to spare for the *housekeeping* part of this set-up.

Since the housekeeping is not a computation intensive task, we use a NiosII/e processor for it alongside the NiosII/f processor running the filters. We call this processor the *Housekeeper* and its job is to feed the data for 20 targets in sequence to the filtering processor and to synchronize the information exchange with the *Gating Module* and the *Assignment Solver*. The housekeeper takes around 3ms to accomplish its task. Thus we reduce the number of processors two from the initially proposed 20 NiosII/s processors for the Kalman filters. The finalized architecture contains a NiosII/f processor for filtering and a NiosII/e for the *housekeeper*.

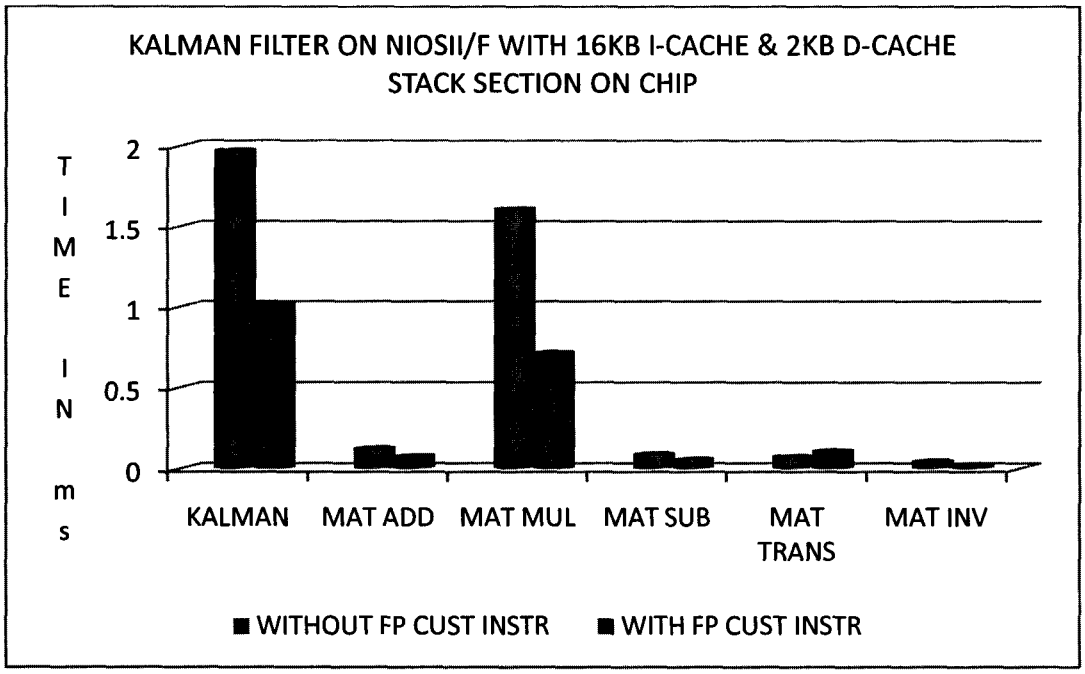


Figure 43: Effects of on chip and off chip memory sections on Kalman Filter

5. Gating Module Optimization

As in the case of the Kalman filter, the optimum cache configuration is necessary for choosing the right NiosII implementation for the *Gating Module* as well as for improving its speed. In section 5.1 we analyze the behavior of the *Gating Module* for various I-cach and D-cache combinations.

5.1. Cache Analysis

Figure 44 reveals the influence of I-cache and D-cache sizes on the processor time of the Gating module running on NiosII/f with 100MHz clock. A remarkable speedup is observed when I-cache size changes from 4KB to 8KB and again when it changes from 8KB to 16 KB. Beyond 16KB the speedup for the I-cache is insignificant. The D-cache size does not matter much as long as it is more than zero. Figure 45 gives a clearer view of the runtime of the Gating Module with respect to the I-cache and D-cache size variation. The overall processor run time is minimum (70ms) when I-cache size I 16KB and D-cache size is 2KB.

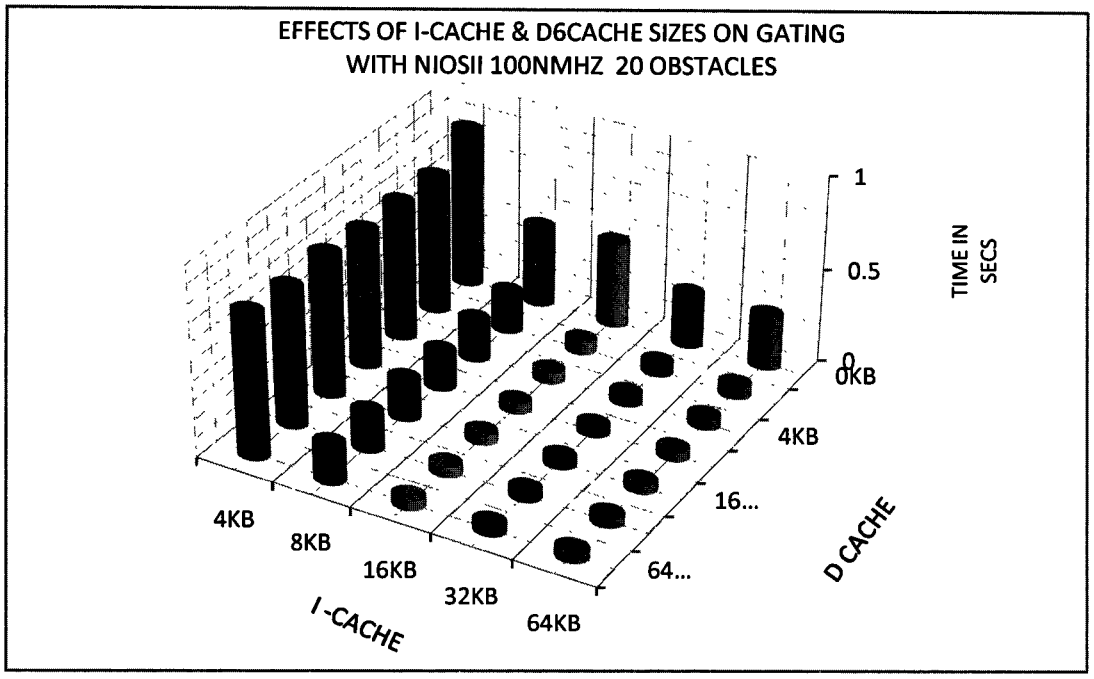


Figure 44: Cache behavior for Gating Module

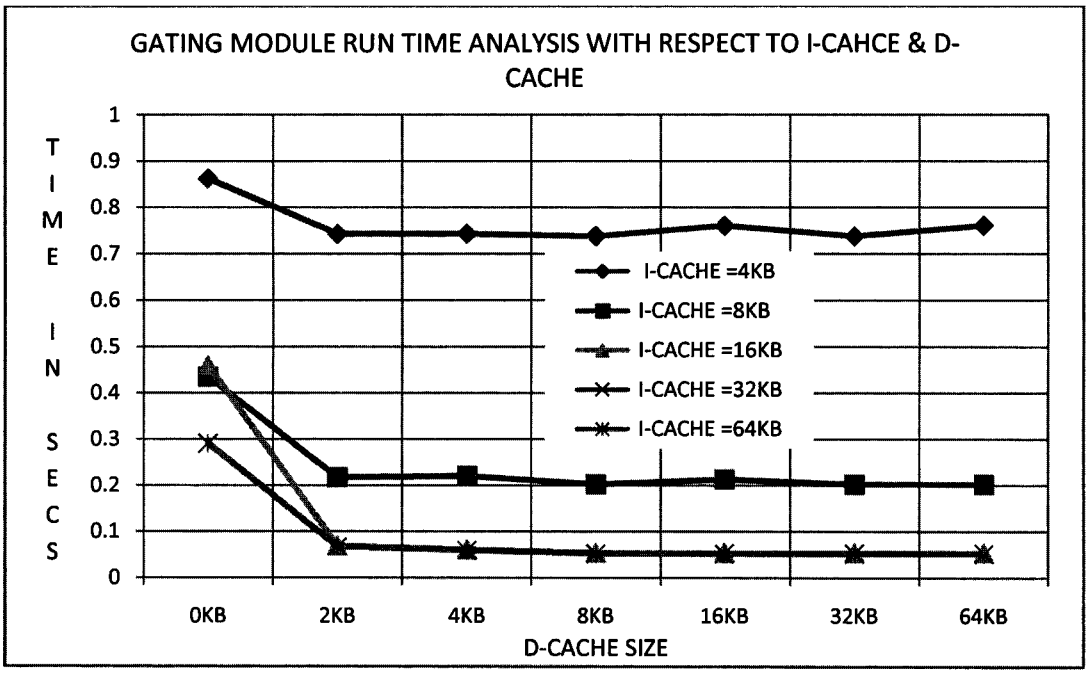


Figure 45: Gating Module Runtime Vs Cache Sizes

Based on these observations 16KB I-cache and 2KB D-cache are the optimum cache sizes for the gating module. The total on-chip block memory usage for this processor sums up to 8% of that available on the FPGA.

Using these cache sizes we charted the performance of the processor while varying number of obstacles from 2 through 20 as shown in Figure 46. 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 *Gate Mask Generator* functions are in turn called up by the *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*. This behavior of the gating module is observed when we use the off-chip SDRAM exclusively.

Although the overall run time for 20 obstacles is minimum (70ms) in the given circumstances, yet it is well above the 25ms mark we are aiming for. We have to improve the processor performance to get to the desired execution time of 25ms.

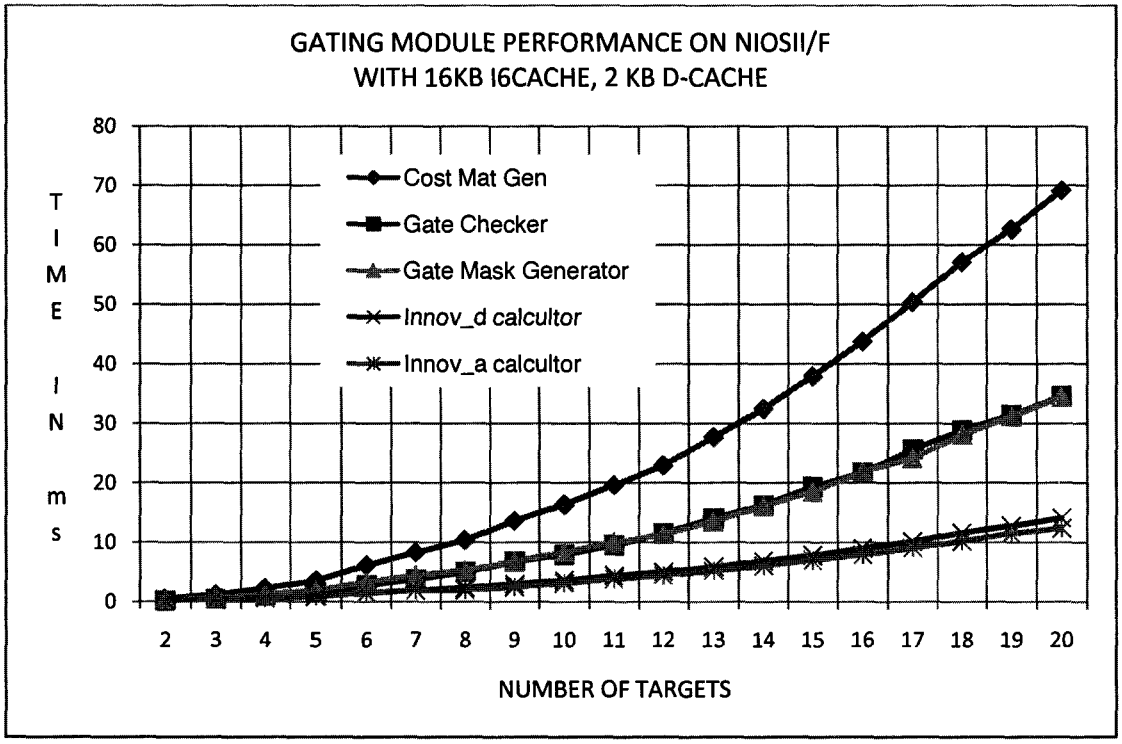


Figure 46: Gating Module Performance with 16KB I-cache & 2KB D-cache

5.2. Floating Point Custom Instructions

With the optimum cache configuration the Gating Module executes in 70ms on the NiosII/f processor. To accelerate the execution of the Gating Module, we inserted the floating point custom instruction hardware into the processor. Figure 47 shows the performance of the Gating module after the floating point custom instructions are added to the processor.

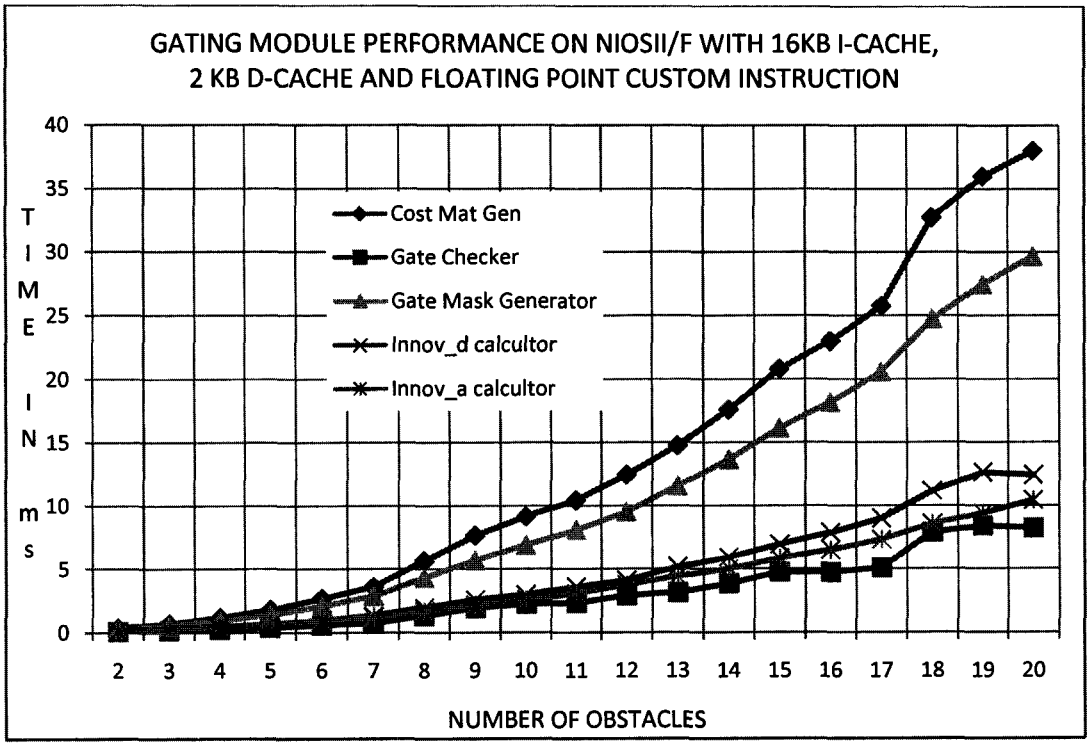


Figure 47: Gating Module performance with Floating Point Custom Instructions

Using the floating point custom instructions with NiosII/f processor for the Gating module improves the overall performance by approximately 50%. Comparing this figure with Figure 46 in section 5.1, we notice two interesting differences. The first, and the very obvious one, is the drop in the overall runtime from 70 ms to 37 ms for 20 obstacles. The second important difference is that the curve for the *Gate Checker*, which was earlier above *the Innov_a* and *Innov_d* curves, is now below them. This shift 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()* 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 don't use *sqrt()*. Although by using the floating point custom instruction we managed to bring the execution time from 70ms down to 37ms yet we are still above the desired 25ms threshold hence we need further improvement in the processor performance.

5.3. On-Chip Memory

To further improve the performance of the *Gating Module*, we placed various memory sections of the function in the on-chip RAM. Figure 48 shows the result of these memory placements.

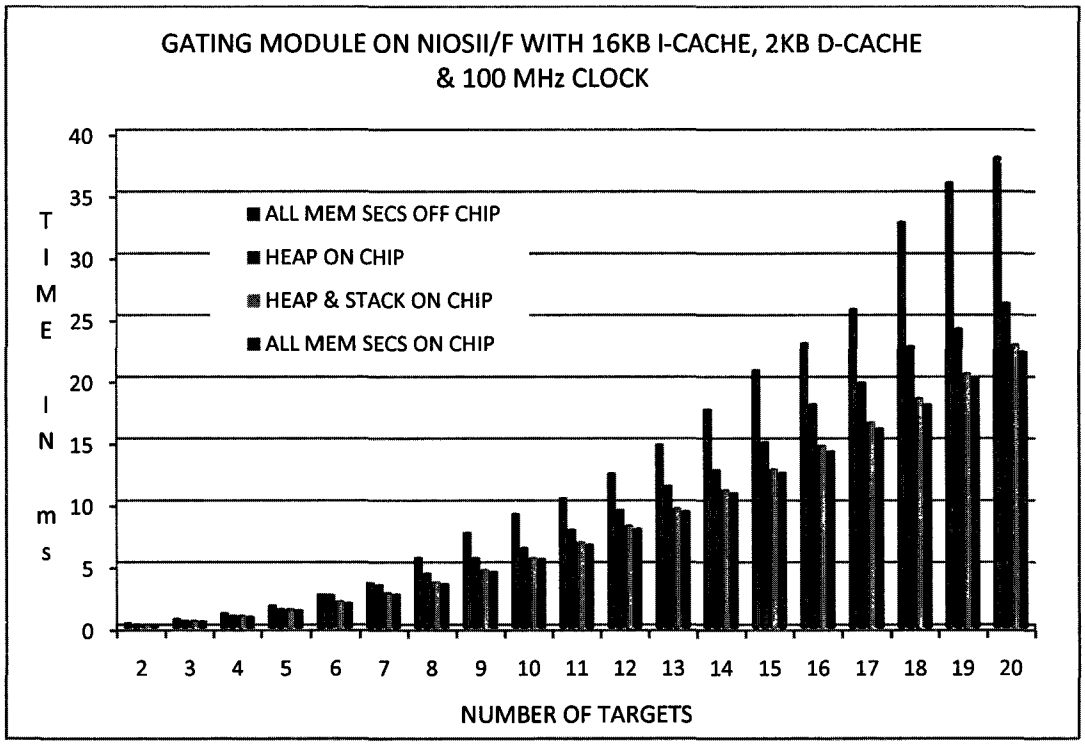


Figure 48: Effects of on chip and off chip memory sections on Gating Module

The lowest runtime of 22ms is achieved by keeping all the memory sections in the on-chip memory. But this would require the on-chip RAM to be at least 61KB. This combined with the on-chip memory taken up by the I-cache and D-cache sums up to 79KB. This is a high

requirement considering the limited on-chip memory availability. The next best solution, 23 ms, is obtained when we place the *stack* and the *heap* sections in the on-chip memory. There is a little speed loss as compared to the conditions when all the memory sections are on chip but in this case only 3KB of dedicated on-chip memory would be sufficient to get this speedup. This is clearly a huge gain in on-chip memory saving compared to the earlier requirement of 79KB. So the Gating module can operate adequately by using a NiosII/f processor with 16KB I-cache, 2KB D-cache, 3KB dedicated on-chip ram and floating point custom instructions.

6. Munkres Algorithm Optimization

Following the same methodology as for the preceding modules, we first determine the optimum cache configuration for the Munkres Algorithm and then make the choice of the suitable processor implementation. For further improvements we consider floating point custom instructions and on-chip placement of various memory sections.

6.1. Cache Analysis

Using a range of instruction and data cache sizes, Munkres algorithm manifested the behavior as presented in Figure **49**. The first observation here is that when the D-cache size is more than 0, the runtime drops deeply, whatever the I-cache size. Beyond 2KB the influence of the D-cache is not so striking. The runtime decreases gradually with increasing I-cache size. Looking more closely at the figure we can eliminate 4KB from the list of the competitors for the I-cache size. Figure 50 shows the runtime more clearly against the cache configurations. An 8KB I-cache along with 16KB D-cache offers the minimum execution time of 71.07ms. 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 51 shows the performance of the algorithm using this system composition for number of obstacles ranging from 2 to 20. *Step1* through *Step6* are the constituent sub-functions of Munkres algorithm. The *Call to Munkres* denotes the total runtime of the algorithm including the six sub-functions. We notice here that the two main contributors to the total runtime are *Step4* and *Step6*. This is because these two sub-functions contain nested loops and they are invoked multiple times during the solution finding process.

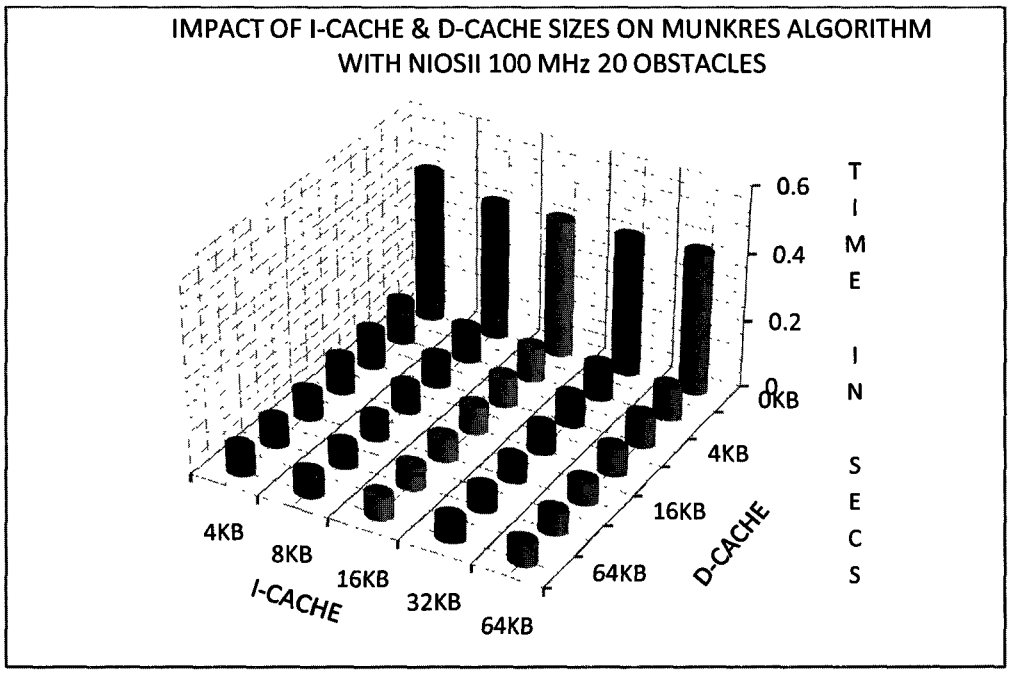


Figure 49: Munkres Algorithm I-cache and D-Cache Analysis

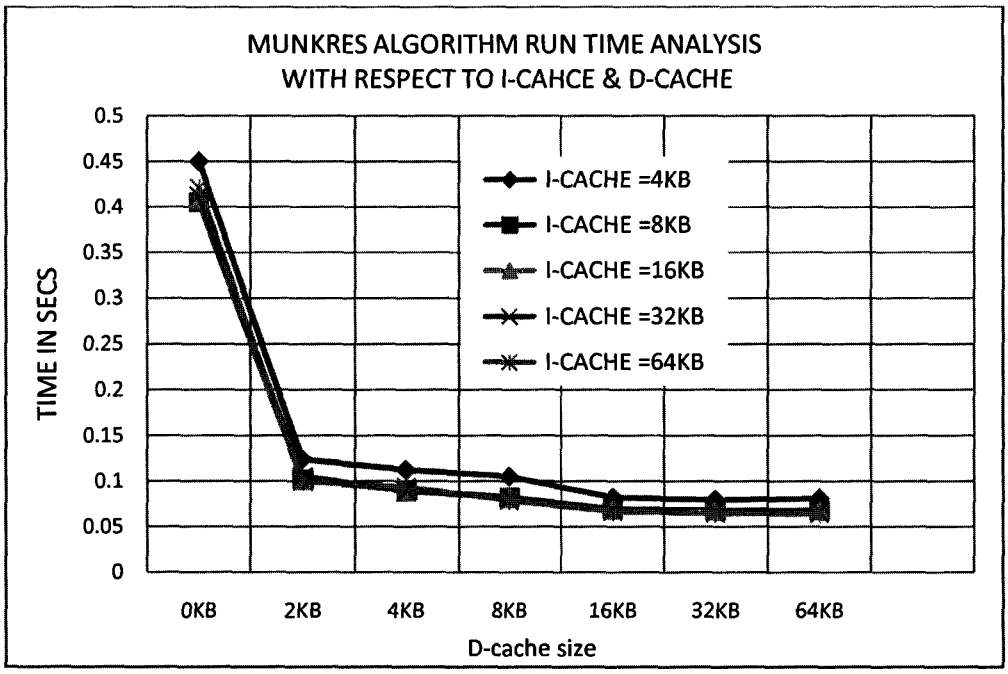


Figure 50: Mankres Algorithm Runtime Vs Cache Sizes

The overall runtime, for 20 obstacles, is 71ms which is higher than the 25ms bound. We need to further optimize the system to meet the speed constraint of the application. For this we use floating point custom instructions in the processor to accelerate floating point operations in the algorithm.

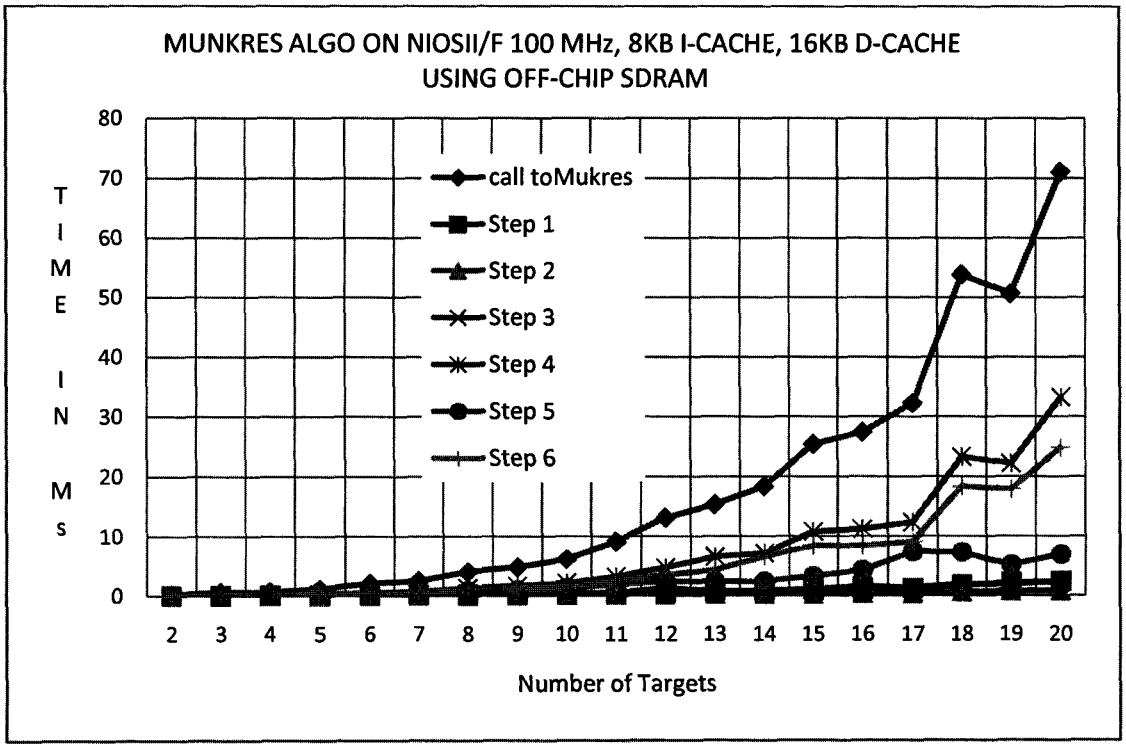


Figure 51: Munkres Algorithm performance with 8KB I-cache and 16KB D-Cache

6.2. Floating point Custom Instructions

Floating point custom instructions bring Munkres algorithm's execution time from 71ms down to 47 ms for 20 obstacles as shown in Figure 52. Although this is a 33.8% improvement over the previous performance, yet 47ms is almost twice the time we aim to attain i.e. 25ms. This urges us to look for other ways and means to improve this performance. In pursuance of this goal we try out several methods as explained in the following sections.

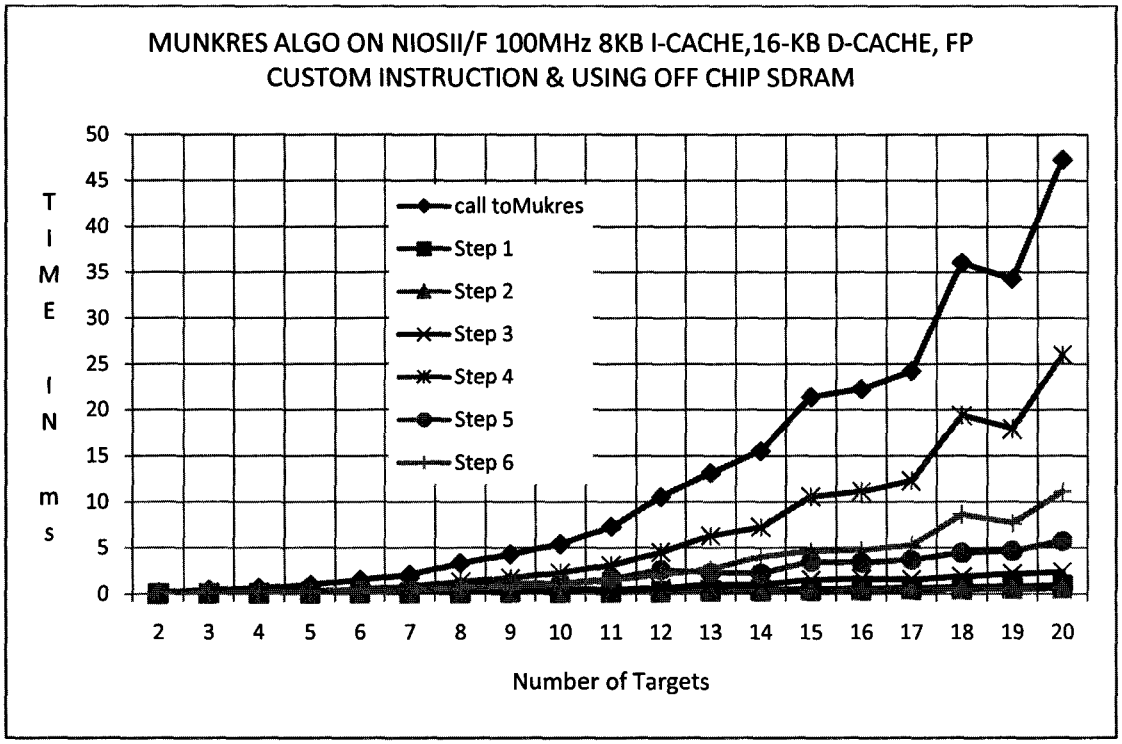


Figure 52: Munkres Algorith performance with Floating Point Custom Instructions

6.3. Munkres Algorithm and Memory Sections

Placing various memory sections on chip does not have a considerable influence on the Munkres algorithm performance, although there is some improvement as shown in Figure 53. We gain only 6ms if all the memory sections are put on chip. The next best gain is achieved by putting the *heap* section 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 25ms, the former is not even feasible given the memory footprint of the algorithm. We have to look elsewhere for a possible and viable solution. The following section explains our approach for dealing with this concern.

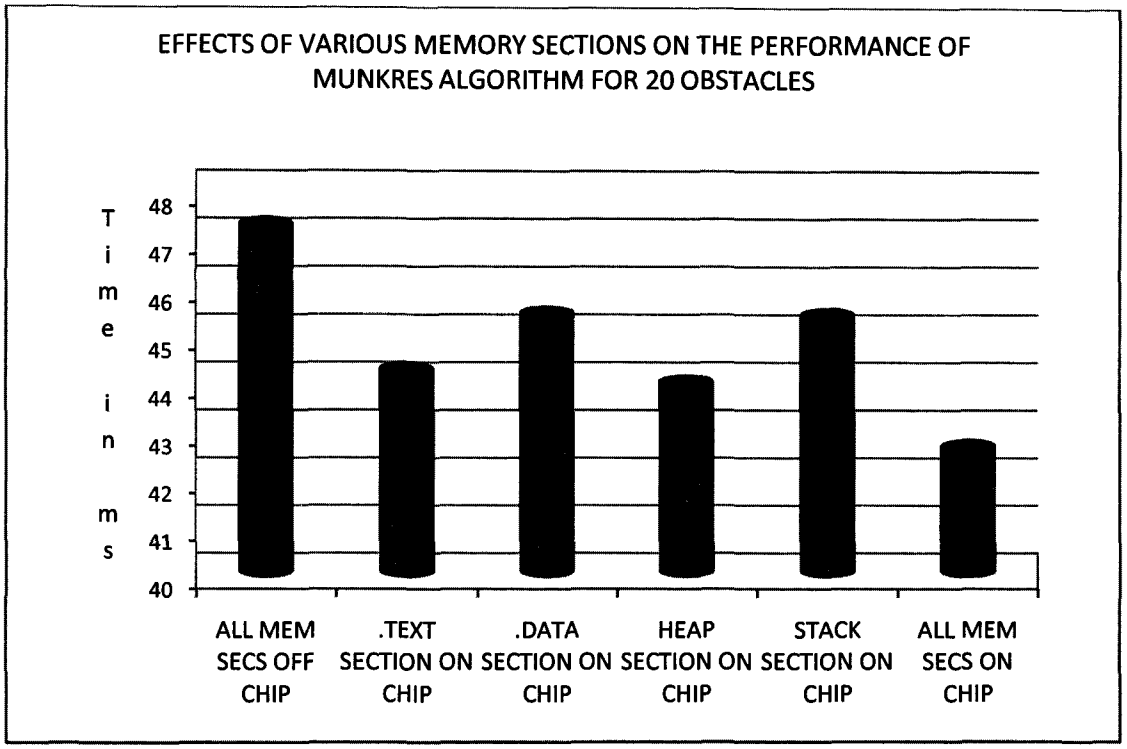


Figure 53: Effect of on-chip and off-chip memory sections on Munkres Algorithm

6.4. Floating Point Vs 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. This is demonstrated in Figure 54 which shows screen shots of the results generated by the algorithm for both types of cost matrices.

Hence we can replace the floating point cost matrix by a "representative" integer cost matrix without sacrificing the accuracy of the final solution. The advantage of this manipulation,

however, is that with 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 precludes the need for the floating point custom instruction hardware. Consequently the size of the processor is reduced by 8%.

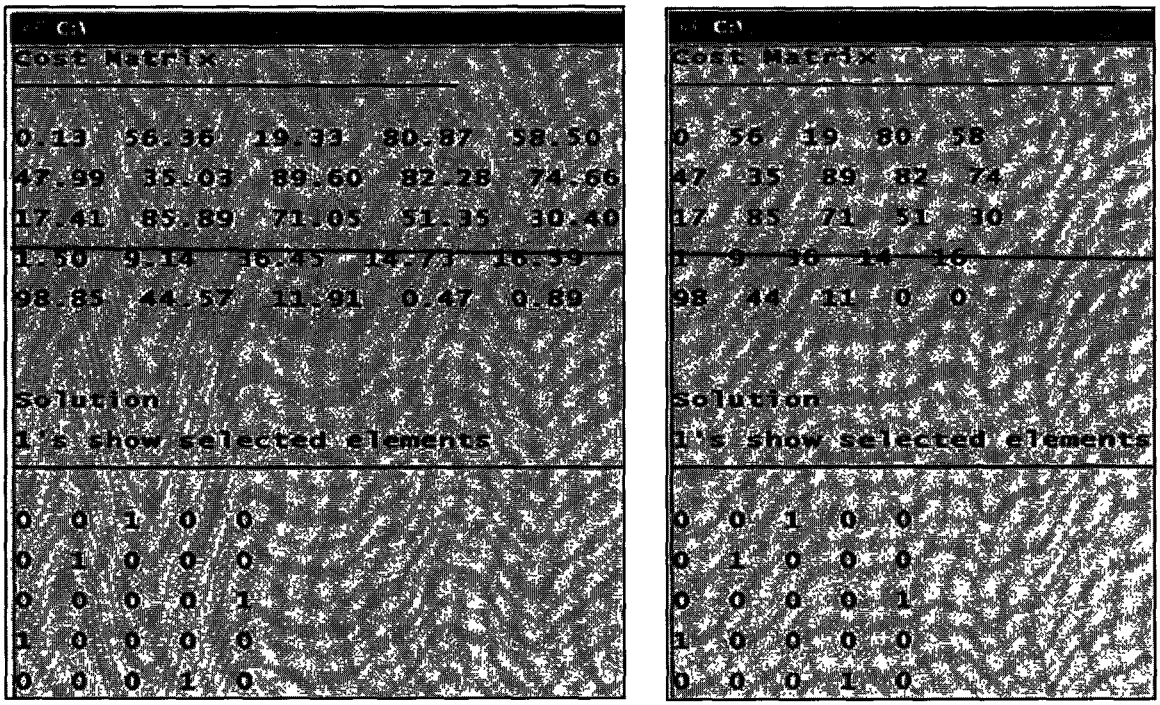


Figure 54: Floating point Cost Matrix versus Integer Cost Matrix

We made necessary modifications to the codes of Munkres algorithm to incorporate this rearrangement. A glimpse of the advantage of these transformations can be seen in Figure 55 which shows the optimal cache configuration for the integer version of Munkres algorithm. Certainly, 8KB I-cache and 16KB D-cache are still the best choices, the point worth noticing here is that with this cache configuration using an integer cost matrix, the runtime for the overall algorithm drops down to 24ms as opposed to the 82ms with floating point cost matrix. So the final solution to Munkres algorithm's defiance is to use a cost matrix with integer elements and map the algorithm to a NiosII/F processor with 8KB I-cache and 16KB D-cache.

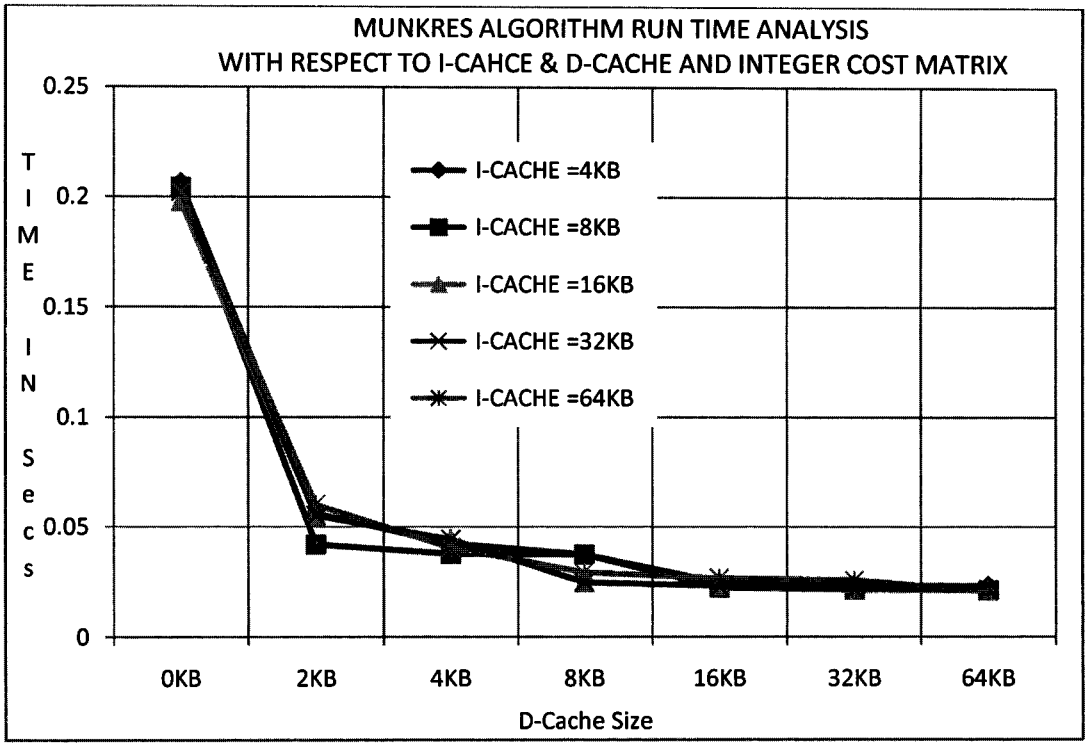


Figure 55: Cache Behavior for Munkres Algorithm with Integer Cost Matrix

6.5. Discussion

After the success of the last optimization of the *Munkres* algorithm discussed in section 6.4, we investigated the other modules for similar optimizations. We found out that this technique cannot be extended to all the modules of the application for the following reasons.

- The *Kalman* filter calculates the predicted states, prediction error covariance, estimated states and estimation error covariance for the targets. The variations 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 carry over to the left of the decimal point. Hence the integer parts of these floating point numbers remain unchanged for hundreds of scans. Nevertheless, these small variations play an important role not only in the filter itself but also in the operation of 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 only the integer parts of these 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.

- The predicted states and the prediction error covariance 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 i.e. *innov_d* and *innov_a* respectively. If we use only the integer parts 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. This would mean that all the targets remained stationary these scans, which is an unlikely proposition. 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.

- 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.

- For the Munkres algorithm (the Assignment Solver) the case is different. The Munkres algorithm is the last step of the application loop. By the time the application reaches this step, most of the floating point operations are already completed resulting in the *Cost Matrix*. The output of the Munkres algorithm is the *Matrix X* which has either 1s or 0s 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*. For these reasons replacing floating elements of the cost matrix with representative integers neither changes the output of the algorithm nor affects the accuracy of the overall application.

7. 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 very short processing time is required for this block. A simple NiosII/e processor executes this block in 8ms. In future we may even remove this processor and run the *Track Maintenance* block as a second task on one of the other processors.

8. Chapter Summary

In this chapter we presented the procedure we adopted for optimizing our application specific MPSoC architecture. Profiling the application helped us in allocating processing resources to the application modules and in planning 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.

For formulating our optimization strategies we also identified the constraints to be met. The constraints include the radar PRT 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. Determining 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 the cut-off time set by the radar PRT. We incorporated floating point custom instructions hardware in the relevant processors to accelerate them further. Floating Point custom instructions reduced the runtime from 70ms to 37ms (47% speedup) for the Gating Module and from 71ms to 47ms (34% speedup) for the Assignment Solver. To bring these times below the radar PRT, 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 23ms which is below the PRT cut-off time and hence we settled for it.

For the Assignment Solver (Munkres algorithm) we gained only 6ms 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. 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 treatment of the input matrix reduced the runtime for the algorithm below the PRT without compromising the accuracy of the final solution. It also allowed us to remove the floating point custom instructions and thereby save 8% of the processor hardware size.

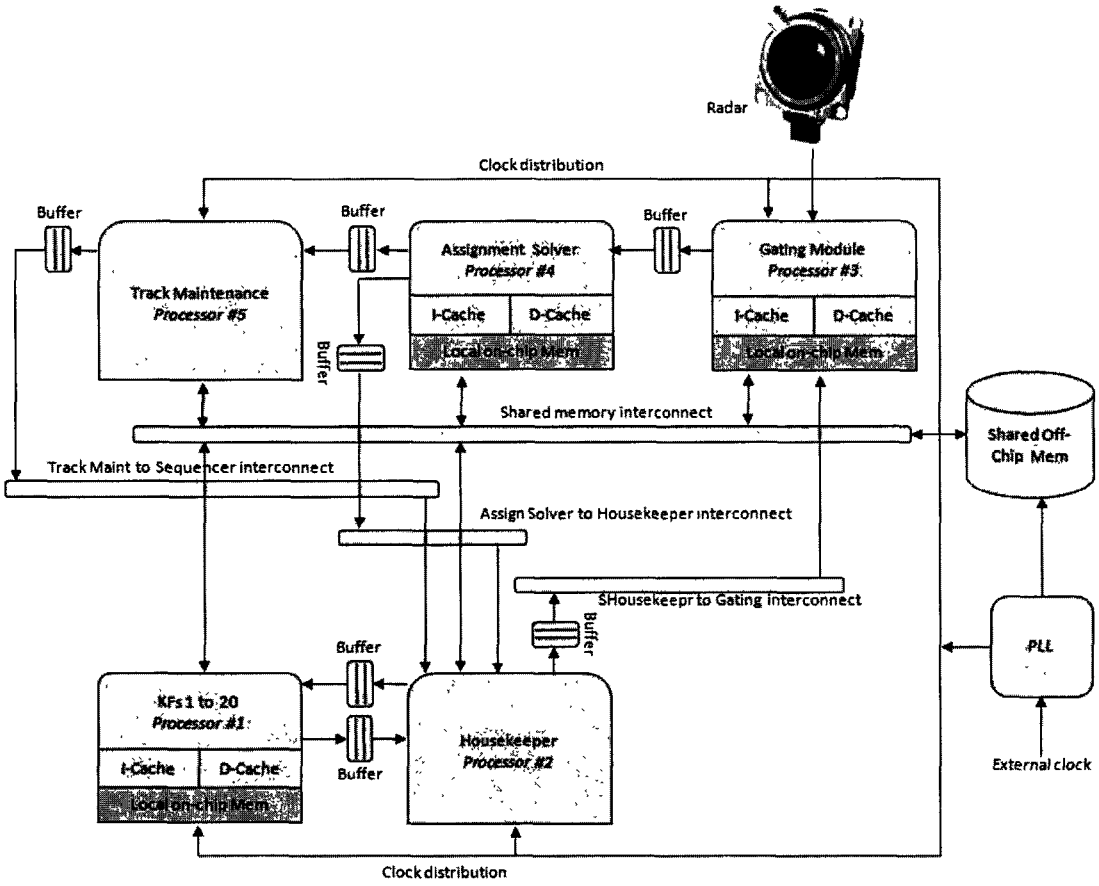


Figure 56: Finalized System Architecture

Processing speed was not the only objective in choosing the system components and the optimization strategies; we sought to keep the on-chip memory utilization and the hardware size in check too. We optimized the amount of processing resources to reduce the hardware size.

The optimized system architecture is shown in Figure 56. In comparison to the initial architecture presented in the Chapter 4, its hardware size is very low since the number of processors for the Kalman filters is reduced from 20 to only one. An additional NiosII/e processor, the *Housekeeper* is added to the architecture. The two NiosII/e processors have neither caches nor local memories. The rest of the processors use caches as well as local memories for to achieve the desired performance.

Table 7 summarizes the finalized system configuration. Note here that the 20ms runtime for the Kalman filters accounts for processing 20 targets one after the other by a single processor. The Munkres algorithm is the module with highest latency of 24ms yet it is lower than the time limit set by the radar PRT. The total on-chip memory usage adds up to 27% of that available in the FPGA. Thus there is enough memory left for future optimizations. The pie chart shown in Figure 57 summarizes the FPGA resources used by the different system components.

Table 7: Finalized System Component Summary

Module Name	Number of Processors	Processor Type	I-Cache Size	D-Cache Size	Local Memory	Memory used on FPGA	FP Custom Instructions	Run time
Kalman Filters	1	NiosII/f	16KB	2KB	3KB	8%	Yes	20ms
Gating Module	1	NiosII/f	16KB	2KB	3KB	8%	Yes	23ms
Munkres Algorithm	1	NiosII/f	8KB	16KB	2KB	9%	No	24ms
House Keeper	1	NiosII/e	0	0	0	1%	No	3ms
Track Maintenance	1	NiosII/e	0	0	0	1%	No	8ms

The system easily fits in the StratixII FPGA with enough space spared for the future evolution of the system.

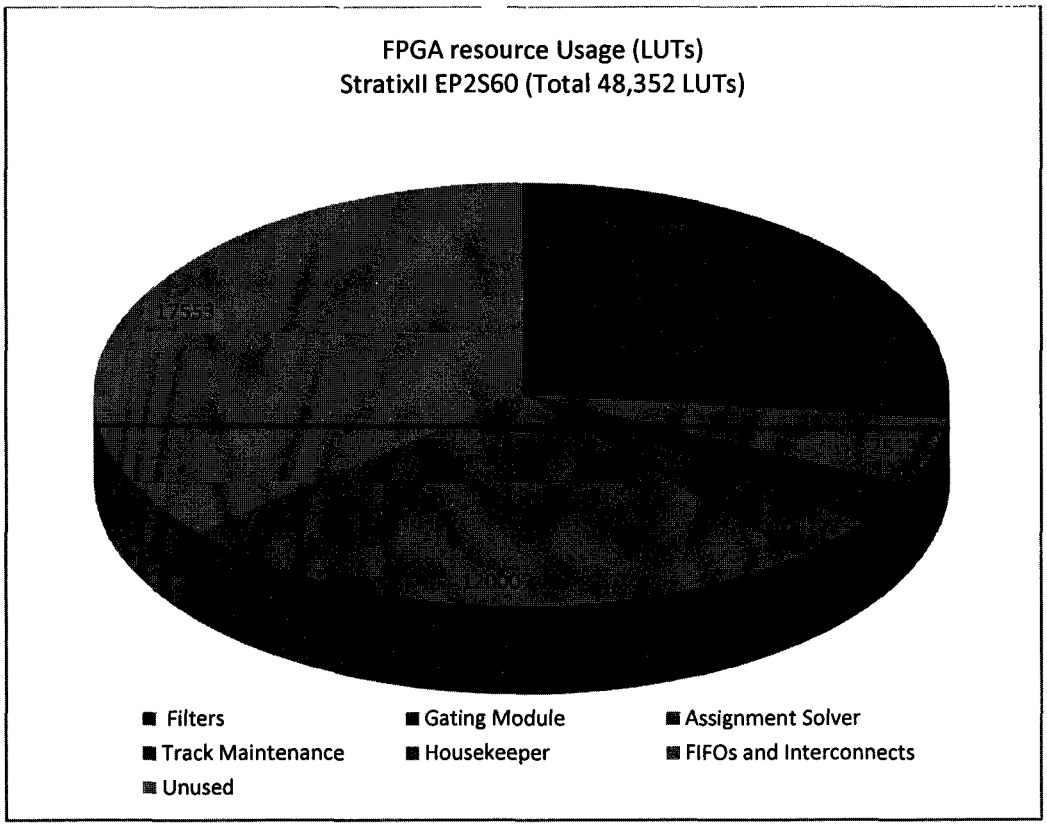


Figure 57: FPGA Resource Usage

6

Conclusion and Outlook

Scientific research and development flourish through the unending quest for finding better and better solutions to scientific problems. The end of one endeavor marks the start of another. However, to keep the interest alive and to quantify the progress, certain milestones are set to be achieved in a predefined timeframe. The objectives set for the three years of this work were largely accomplished. In this chapter we briefly recapitulate on the highlights of our efforts and their results achieved in the last three years. In the second half of the chapter we provide directions for the possible extensions of the work.

1. Conclusion

Road accidents are one of the most lethal problems the humanity is confronted with. The need for finding viable solutions to the problem is becoming more and more imperative with every passing day. Researchers, automotive manufacturers and government authorities around the world are continuously looking for answers to this problem. The conventional safety systems, onboard a vehicle, are mostly intended for post-crash damage minimization rather than accident avoidance. Research has shown that half of the accidents can be reduced if a driver is alerted to

an impending collision a fraction of a second in advance. As a result various kinds of systems have been proposed for warning the driver of an approaching danger. These systems are collectively called Driver Assistance Systems (DASs). Most of the existing DASs, though helpful, are too limited in their functionality to be fully effective. Moreover, in most cases they are too costly to be employed on a large scale. The lack of efficiency of these systems can be attributed to the design principles and implementation techniques of these systems. The implementation technology is also one of the causes of the exorbitant costs of these systems.

In this work our objective was to provide an efficient, low cost and evolvable solution to the road accident problem. To propose an efficient solution, we analyzed the principal scenarios of road accidents. Examining accident statistics we found that a great majority of the vehicle crashes result from frontal collisions. So minimizing frontal collisions would significantly decrease road accidents. To predict a frontal collision sufficiently in advance, the obstacle must be detected from a distance. Moreover, for the DAS to be really effective, an imminent collision must be sensed in all circumstances, especially in poor weather where the DAS is needed most. A radar sensor fulfills both the prerequisites of long range obstacle detection and all-weather operation. However, only detecting obstacles can be useful to a certain extent. To establish whether an obstacle is on a collision course with the host vehicle, its trajectory must be foreseen before it comes close to the host vehicle. Determining the trajectory of a moving object requires its dynamic behavior to be monitored over a period of time. In a real traffic scenario more than one obstacle can pose a danger to the host vehicle. Hence trajectories of multiple objects have to be monitored simultaneously. An apparatus which is capable of performing such functions is called a Multiple Target Tracking (MTT) system.

In Chapter 3 of this document we explained the fundamental concepts of the MTT application. The principles of MTT have been used in aviation applications like Air Traffic Control (ATC) etc. However, the specific operating environment of automotives is very different than that of airborne vehicles. Therefore, an MTT application designed for aviation use cannot be directly applied to automotives. Hence we specifically designed an MTT application for use in automotives. We modeled the application mathematically and then structured it into easily manageable software functions and sub-functions.

While theoretically an MTT system offers one of the best answers to the road accident problem, its practical implementation is not a trivial task. It involves complex computations and consequently, needs a long processing time. However to alert a driver to an approaching danger in real time, the computations must be performed very rapidly. We use multiple processors to share the computation load and thereby reduce the processing time. Multiple processors running in parallel not only speed up computation but also address the power consumption issues of the embedded systems. Due to these advantages many multiprocessor platforms have been developed on commercial scale, for various applications. Commercially available multiprocessor

platforms have fixed hardware architectures making it impossible to customize them for an evolving application. Apart from the rigidity of their architectures, the commercially available multiprocessor platforms are very costly too. Modern day embedded systems must be flexible enough to evolve according to the demands of the end user and they must have low cost.

We took all these aspects into consideration while designing the architecture of our system. In Chapter 4, we presented the mapping of our MTT application to the multiprocessor architecture of the system. We took care to group the application sub-functions in such a way that the dependency of computations, among various groups, is minimized. This way the communication among the groups of functions is as low as possible. We label these groups as modules and map these modules to individual processors to form an interim architecture for the system.

We chose FPGA as the implementation platform for our multiprocessor system. FPGAs offer the flexibility needed for the ever evolving embedded systems and they are very cost effective. A multiprocessor system implemented in an FPGA makes its architecture flexible and reconfigurable while the processors in the architecture can be reprogrammed when needed. Thus FPGA based multiprocessor systems guarantee flexibility in hardware as well as in software. The hardware flexibility of our multiprocessor architecture owes its existence, mainly to the use of soft-core processors. Soft-core processors are IP cores defined in a Hardware Description Language (HDL) in which many architectural parameters can be customized at system design time. The architectural parameters are chosen according to the requirements of the application to be executed on the processor. Thus their hardware size, processing speed and configuration can be optimized at design time. Optimization of the architecture leads to the reducing the system hardware size and hence its energy consumption. Moreover, reducing the hardware size allows us to fit the system in a smaller FPGA to reduce the cost of the system.

In Chapter 5, we described the optimization process for improving our preliminary architecture. Our goal was to minimize the hardware size while maximizing the speed within possible limits. These seemingly contradictory objectives were accomplished by using various optimization techniques. The strategies included optimizing processor configurations like their cache sizes, size of dedicated local memories, custom instructions and placements of various memory sections in the on-chip or off-chip devices. These strategies were successful enough to reduce the number of processors to five from the initially proposed 23. We also managed to bring the overall processing time to below the radar PRT (Pulse Repetition Time). The whole system fits in a contemporary medium size FPGA leaving enough space for future enhancements and functionalities.

2. Outlook

While the work accomplished the envisaged objectives set at the start, several extensions are possible for future continuation. Future work can be divided into two categories; the first category is related to the application aspects and the second the category concerns the architectural aspects of the system.

In the application related areas we foresee the following venues for exploration.

- In the current form of the application we use the Kalman filter for the prediction and estimation of the target states. The Kalman filter performs with sufficient speed and accuracy for road safety applications. Nevertheless, other estimators may be explored for better speed and precision performance, especially in cases where obstacles are expected to exhibit nonlinear dynamic behavior e.g. in hilly regions. One of the algorithms that may be used in such situations is the Extended Kalman Filter (EKF) which takes into account the nonlinearities in the prediction and estimation processes. Another algorithm used mainly in robotics, is the Mean-shift algorithm. This algorithm may be investigated for computational resource saving.

- In the current work we solve the assignment problem in the data association part of the application by using the Munkres algorithm. As we saw in Chapter 5, this algorithm takes the longest of all the application modules after optimization. We see two opportunities of deeper study in this respect. The first possibility is to examine the Auction algorithm for solving the assignment problem in the data association function. The auction algorithm has been successfully used in fields like network resource allocation.
  The second venue of research in this regard can be the incorporation of the radar cross-signature of the obstacles into the data association process. The radar signature depends on the size, the form and the angle of approach of an obstacle. If the signature of an obstacle, in the current radar scan, matches with that of a previously detected obstacle, the association between the two can be confirmed with little further screening. This functionality can be implemented by a pattern matching algorithm. Used in conjunction with the any of the assignment solver algorithms, it would highly improve the accuracy of the data association process. At the same time, it would simplify the gating process as well as the assignment solver due to a reduced cost matrix.

On the architectural front, we foresee several tracks for future research.

- The first one concerns the mapping of the three most time-consuming sub-functions of the Munkres algorithm (*step4* through *Step6*) onto individual processors while the rest of the algorithm is mapped onto a separate processor. This would form a cluster of four processors locally communicating in a synchronized fashion while the cluster would communicate asynchronously with the outer world. Such types of architectures are known as GALS (Globally Asynchronous Locally Synchronous) architectures. This would accelerate the execution of the algorithm while its internal operations would be isolated from the rest of the system. In the same manner the Gating Module can be distributed over three processors; one each for *innov_a* and *innov_d* and the third for the rest of the module.

- As an alternative, the floating point operations may be converted into fixed point operations. In fixed point format, the computational components become smaller hence the application may be implemented as hardwired circuitry in the FPGA. Hardwired circuits operate at very high speed as compared to the software running on processors. However, sufficient accuracy must be maintained during the conversion from floating point to fixed point operations.

- Another approach can be to integrate dynamic reconfigurability into the system for adapting its architecture to the operating conditions at runtime. This involves in-depth investigation into the application as well as the operating environment of the system.

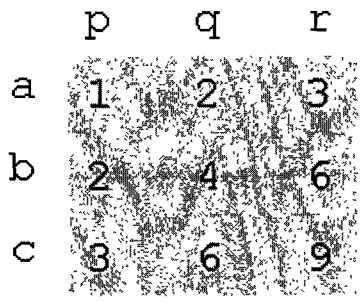
Appendices

A. An example assignment problem solved by Munkres Algorithm.

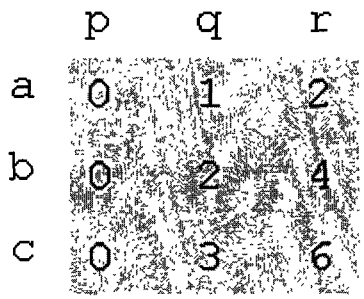
Observations = { a, b, c}
Predictions = {p, q, r}

Cost of assigning prediction j to observation i is

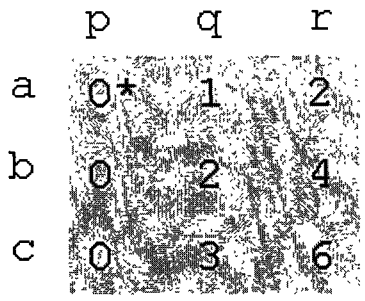
$$C(i,j) = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$



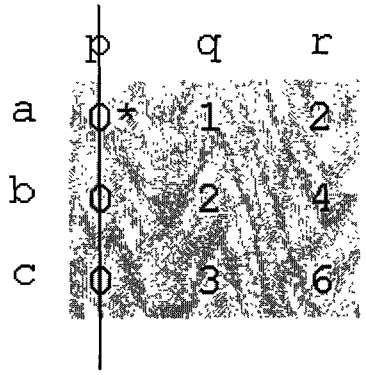
1. Step 0



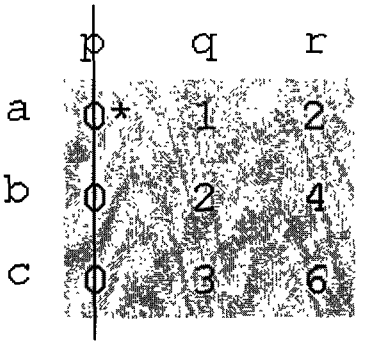
2. Step 1



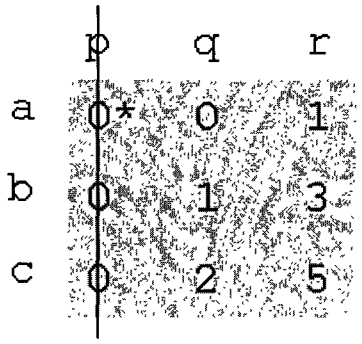
3. Step 2



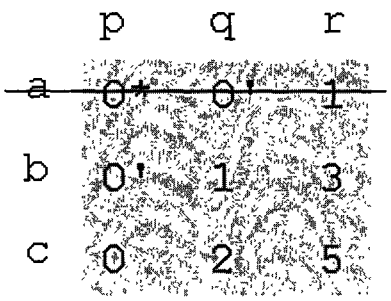
4. Step 3



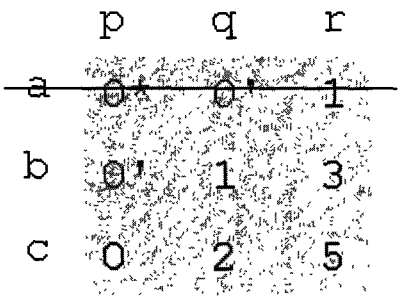
5. Step 4



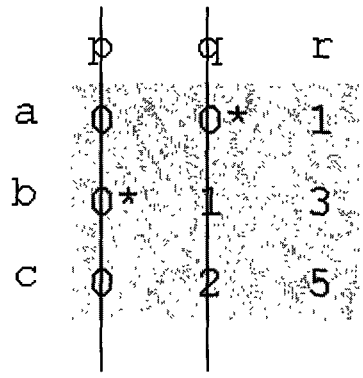
6. Step 6



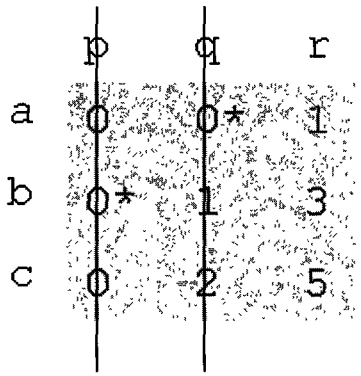
7. Step 4



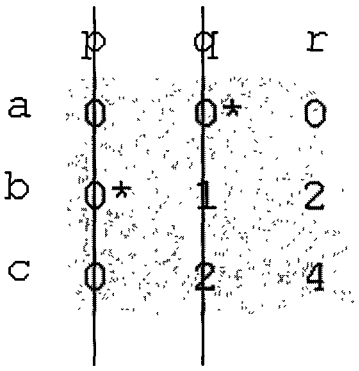
8. Step 5



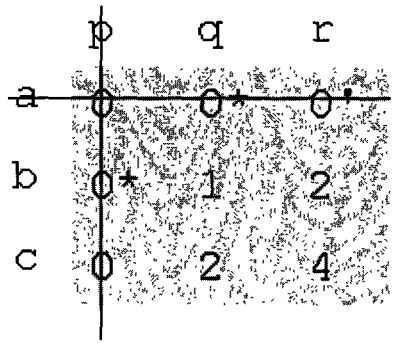
9. Step 3



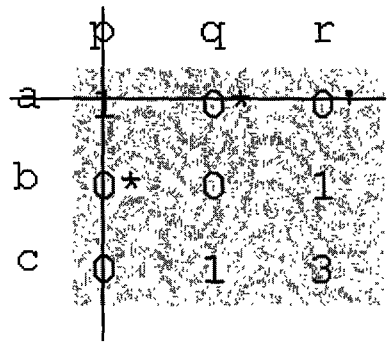
10. Step 4



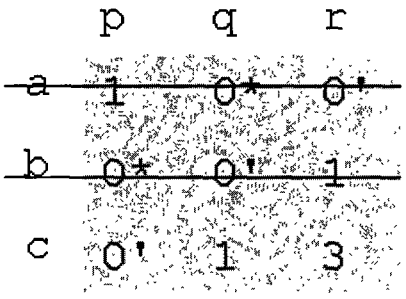
11. Step 6



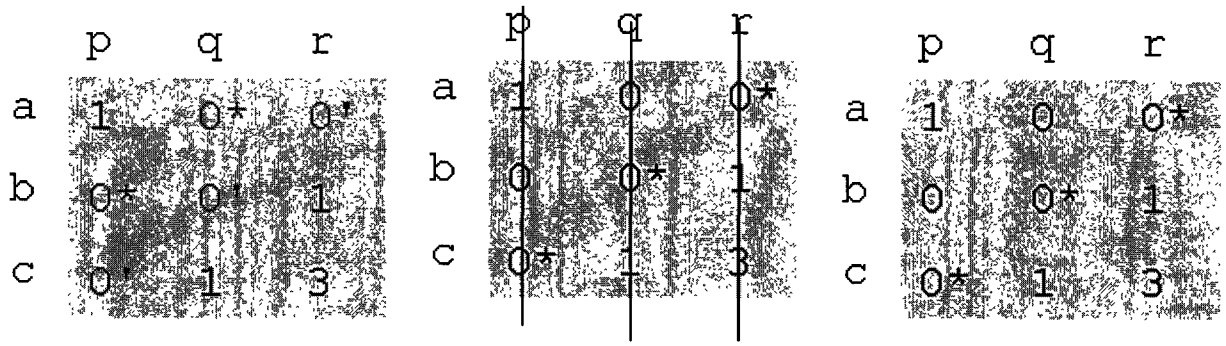
12. Step 4



13. Step 6



14. Step 4



15. Step 5

16. Step 3

17. Done

B. Some MPSoC Architectures

I. Lucent Daytona

The first known MPSoC according to (66), is the Lucent Daytona shown in **Figure 58**. Daytona was designed by Lucent Technologies (a Bell Labs subsidiary) for wireless base stations. It contains four SPARC V8 CPUs connected to a high speed bus in a symmetric architecture. The CPUs internally contain a 32-bit RISC, 64-bit SIMD, a 16 x 64 register file and a controller each. The four SPARC CPUs are enhanced with 16 × 32 multiplication, division step, touch instruction, and vector coprocessor. Each CPU a reconfigurable L1 cache and they snoop to ensure consistency. The CPUs share a common address space in memory. The high speed system bus supports split transactions.

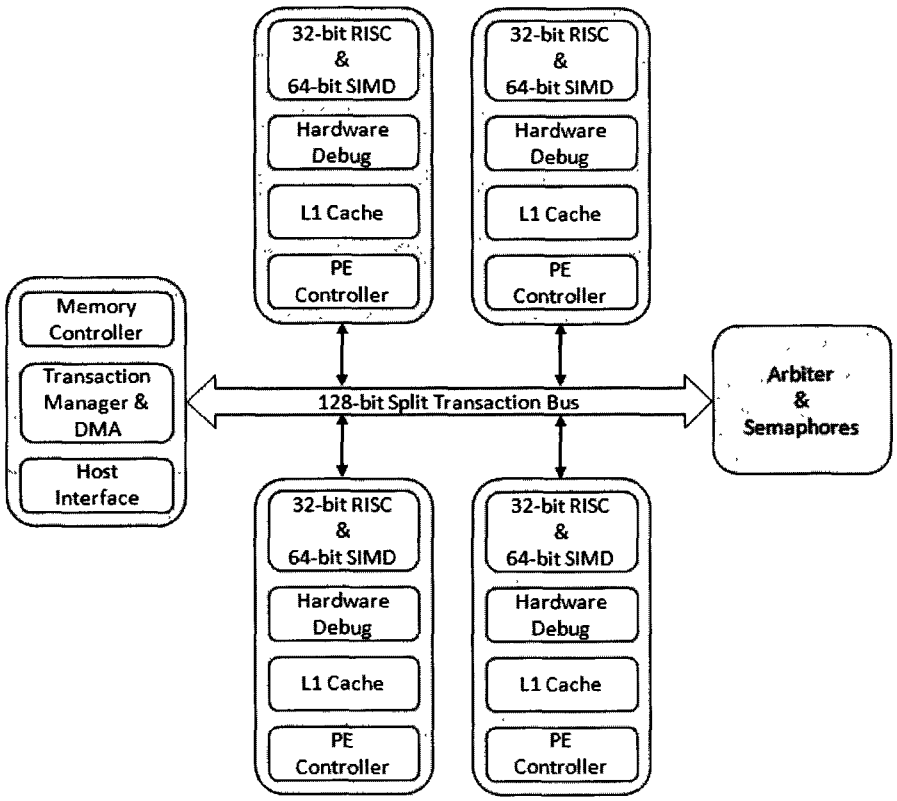


Figure 58: Lucent Datona Architecture

II. Nomadik

The ST Microelectronics Nomadik shown in **Figure 59** is also aimed for mobile multimedia applications. It contains a combination of an ARM926 core and two programmable accelerators based on MMDSP+. The ARM is used as the host processor while one of the accelerators runs audio related functions and the other is dedicated to video processing. The architecture is built on classic bus. The architecture supports 16-bit, 24-bit fixed point and 32-bit floating point operations.

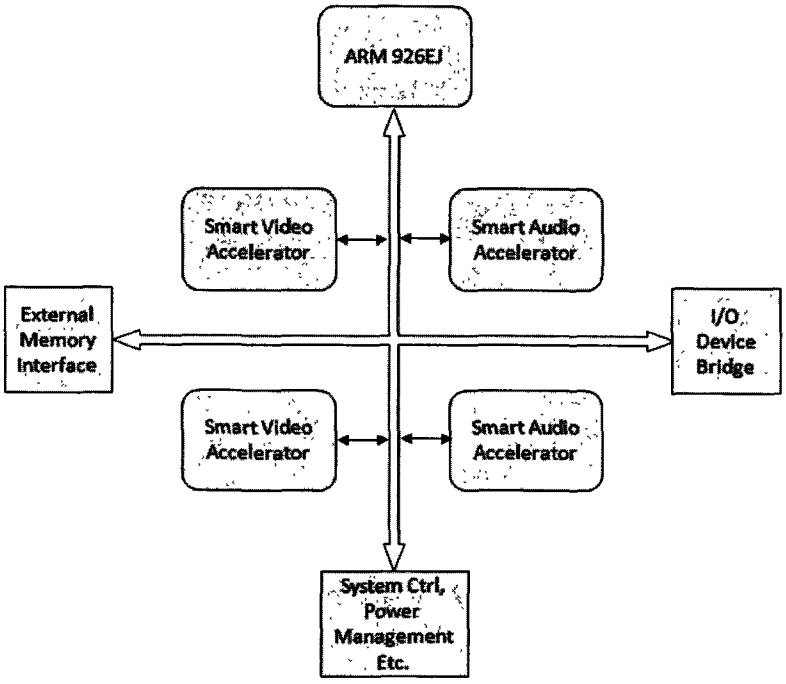


Figure 59: ST Microelectronics NOMADIK

III. IXP2855

The Intel IXP2855 is a network processor. It contains 16 multi-threaded micro-engines organized into two clusters to for data processing and an Intel XScale core for control functions. In addition, the IXP2855 integrates two cryptography blocks that provide hardware acceleration of encryption and data integrity algorithms. The two cryptography blocks utilize the same bus structures and communication processes as the micro-engines.

IV. 1AX with AMBA Bus

The 1AX (67) architecture as illustrated in **Figure 60**, is composed of an ARM7 processor, a configurable XTENSA processor and a global memory of 256MB. Both the processors are also equipped with their private local memories. The ARM processor is used to execute the control functions of the application, while the Xtensa processor is used for r processing data-intensive algorithms. The Xtensa processor can be customized to the target application functions with an automatic instruction set generator called XPRES (Xtensa Processor Extension Synthesis). All

the components are interconnected using an AMBA bus. The processors communicate via mailboxes.

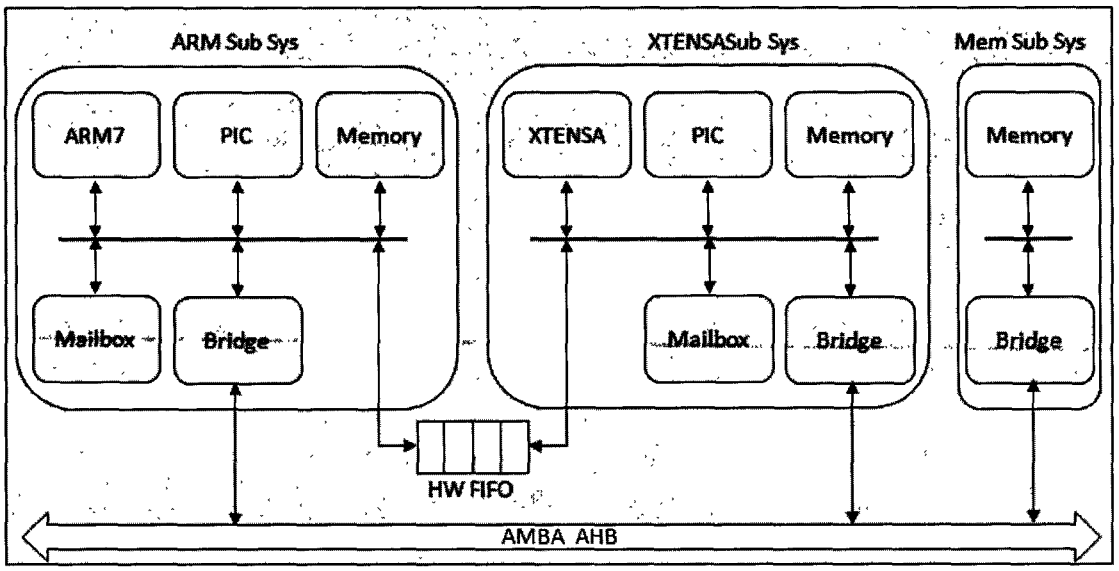


Figure 60: 1AX Architecture with AMBA Bus

The global memory is accessible to both processing units through a bridge between the AMBA bus and the memory. The architecture also contains a hardware FIFO (HWFIFO) directly connected to the local buses of the two processors. The HWFIFO is used for synchronization.

The memory address space of a processor subsystem is divided into two parts: 3 MB for local memory and 1 MB for peripheral memories. Bus transactions with addresses lower than 4 MB (0x00400000) are treated as accesses to local components, while those with addresses higher than 4MB are forwarded to the global AMBA bus via the bridge component of the processor subsystem.

This architecture allows two types of communication schemes between the processors: using the global memory and using the hardware FIFO. In the first communication scheme, one processor can deliver data to other processor though a global shared memory and send a synchronization event via a mailbox between different processors. The second possible communication scheme between the two processors is based on the hardware FIFO. The HWFIFO is a point-to-point communication between two processor subsystems. Besides the data transfer, the HWFIFO also implements the synchronization mechanism of the processors. The HWFIFO provides an alternative path for data transfer instead of using the shared memory and global network. Thus, it can decrease the required bandwidth of the global memory and network and speed up the communication.

V. Diopsis RDT

The Shapes (68) MPSoC architecture shown in **Figure 61**, is a multi-tile architecture based on a Diopsis tile also called D940. The Diopsis tile is a triple core system integrating an ATMEL mAgicV VLIW DSP an ARM 9 RISC microcontroller and a distributed network processor (DNP). The local memories of the DSP and RISC can be accessed by both processing units. Additionally, a distributed external memory (DXM) can be used to share data among all the processors. The data transfer between these processors can follow different paths using an AMBA bus, e.g. the DSP can read/write data to the local memory of the ARM by using a DMA transfer or bypassing the DMA. The ARM includes the processor core and local memories: SRAM for data and ROM for program code. The DSP includes the DSP core, data memory (DMEM), program memory (PMEM), control and data registers (REG), direct memory access engine (DMA), programmable interrupt controller (PIC) and the mailbox as synchronization component for the communication between the two processors.

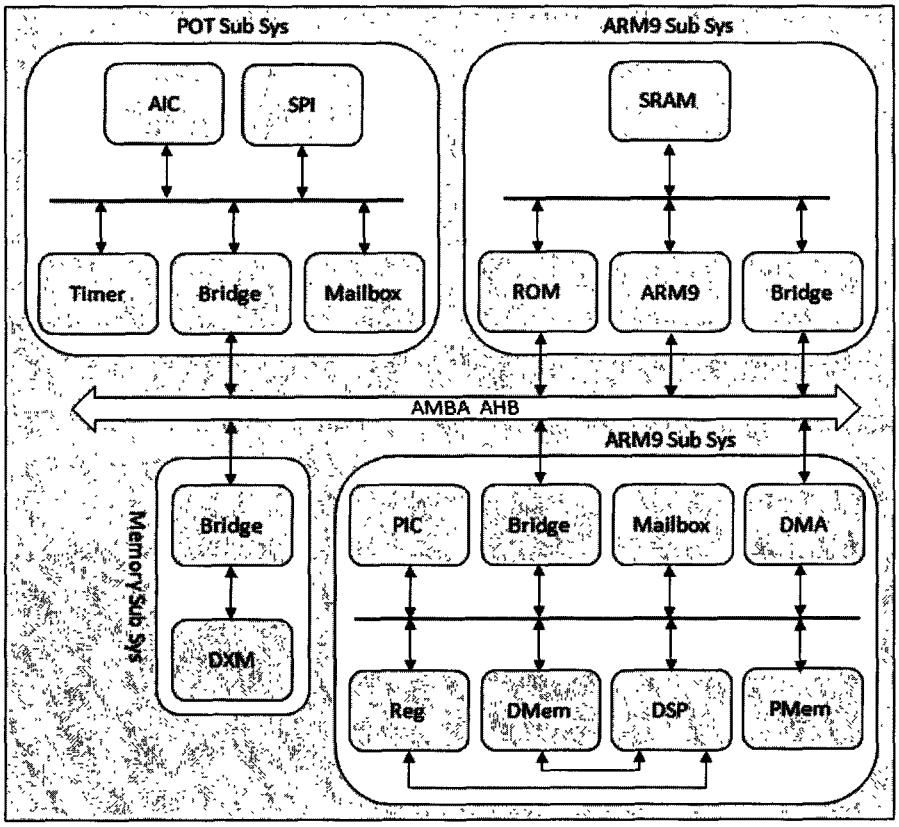


Figure 61: Diopsis RDT Architecture

The hardware nodes consist of distributed external memory subsystem (DXM) and peripherals on tile (POT) subsystem. The distributed external memory subsystem includes a global memory shared by the processors. The POT includes the system peripherals of the RISC processor, e.g. timer, advanced interrupt controller (AIC), but also the I/O components of the tile such as the serial peripheral interface (SPI).

The interconnection between these software and hardware subsystems is made via the AMBA bus. Hence, all the subsystems contain a bridge component to interface with the AMBA bus and a local bus for the local components interconnection.

The ARM processor can access directly the data memory and control/status registers of the DSP processor via the AMBA slave interface of the DSP subsystem. In the same way, the DSP core can read/write directly on the local memory of the RISC processor by initiating a DMA transfer. Moreover, the processors can store and load data to/from DXM connected to the AMBA bus. Therefore, this architecture allows different kinds of communication mapping schemes between the processors characterized by different performances.

C. The Nios II Processor Deatails

Processor Architecture

The Nios II architecture describes an instruction set architecture (ISA). The ISA in turn necessitates a set of functional units that implement the instructions. A Nios II processor core is a hardware design that implements the Nios II instruction set and supports the functional units.

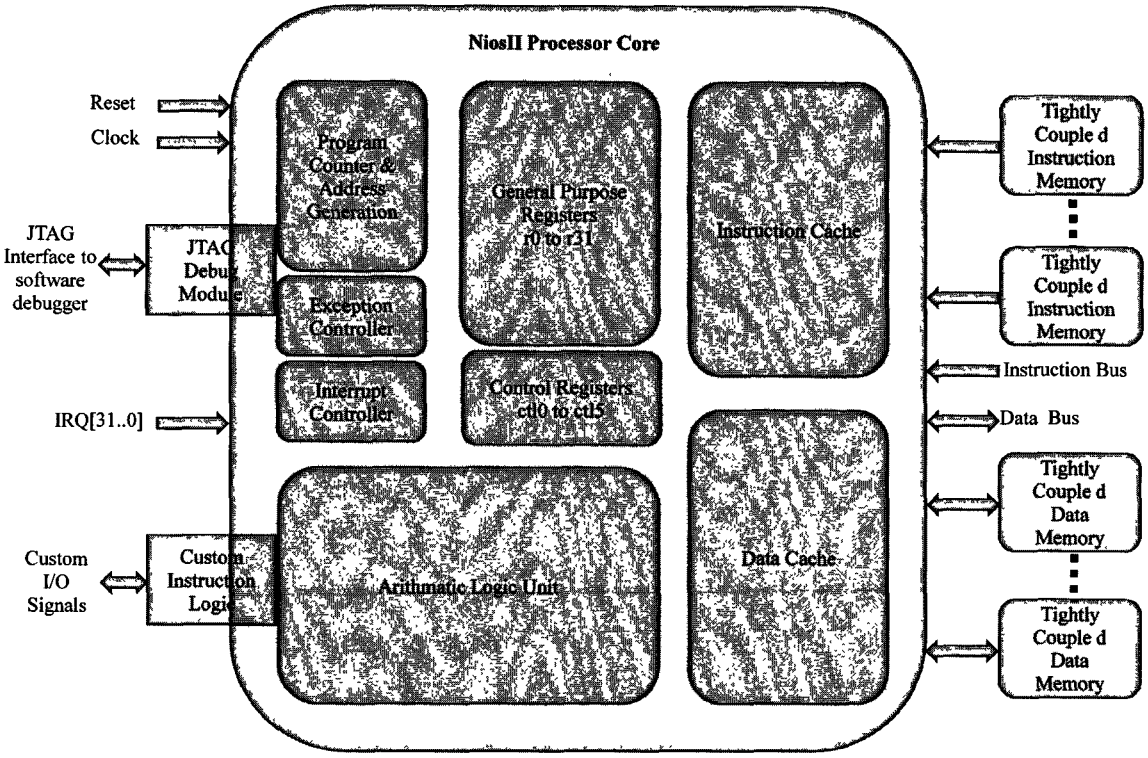


Figure 62: The Nios II Softcore Processor

A functional unit can be implemented in hardware, emulated in software, or omitted entirely. Every implementation achieves specific objectives, such as smaller core size or higher performance. Implementation variables generally fit one of three trade-off patterns: more-or-less of a feature; inclusion-or-exclusion of a feature; hardware implementation or software emulation of a feature.

- More or less of a feature—For example, to fine-tune performance, the designer can increase or decrease the amount of instruction cache memory. A larger cache increases execution speed of large programs, while a smaller cache conserves on-chip memory resources.

- Inclusion or exclusion of a feature—For example, to reduce cost, the designer can choose to omit the JTAG debug module. This decision conserves on-chip logic and memory resources, but it eliminates the ability to use a software debugger to debug applications.

- Hardware implementation or software emulation—for example, in control applications which rarely perform complex arithmetic, the designer can choose for the division instruction to be emulated in software. Removing the divide hardware conserves on-chip resources but increases the execution time of division operations.

Register File

The Nios II architecture supports a register file, consisting of thirty two 32-bit general-purpose integer registers, and up to thirty two 32-bit control registers. Currently there are no floating-point registers provided in the Nios II architecture.

Arithmetic Logic Unit

The Nios II ALU operates on data stored in general-purpose registers. ALU operations take one or two inputs from registers, and store a result back in a register. To implement any other operation, software computes the result by performing a combination of the fundamental operations.

Some Nios II processor core implementations do not provide hardware to support the entire Nios II instruction set. In such a core, instructions without hardware support are known as unimplemented instructions. The processor generates an exception whenever it issues an unimplemented instruction so the exception handler can call a routine that emulates the operation in software. Therefore, unimplemented instructions do not affect the programmer's view of the processor but they take longer to execute than the implemented instructions.

The Nios II architecture also supports user-defined custom instructions. The Nios II ALU connects directly to custom instruction logic, enabling the user to implement in hardware operations that are accessed and used like native instructions. Like custom peripherals, custom instructions allow the user to increase system performance by augmenting the processor with custom hardware. The soft-core nature of the Nios II processor enables the user to integrate custom logic into the arithmetic logic unit (ALU). Similar to native Nios II instructions, custom instruction logic can take values from up to two source registers and optionally write back a result to a destination register.

From the software perspective, custom instructions appear as machine-generated assembly macros or C functions, so programmers do not need to understand assembly language to use custom instructions.

The architecture supports single precision floating-point instructions as specified by the IEEE Std 754-1985. 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. These floating-point instructions are implemented as custom instructions.

Exception and Interrupt Controllers

The Nios II architecture provides a exception controller to handle all exception types. Every exception, including hardware interrupts, causes the processor to transfer execution to an exception address. An exception handler at this address determines the cause of the exception and dispatches an appropriate exception routine.

All exceptions are precise which means that the processor has completed execution of all instructions preceding the faulting instruction and not started execution of instructions following the faulting instruction. Precise exceptions allow the processor to resume program execution once the exception handler clears the exception.

The Nios II architecture supports 32 external hardware interrupts. The processor core has 32 level-sensitive interrupt request (IRQ) inputs, irq0 through irq31, providing a unique input for each interrupt source. IRQ priority is determined by software.

Instruction and Data Buses

The Nios II architecture supports separate instruction and data buses, classifying it as a Harvard architecture. Both the instruction and data buses are implemented as Avalon-MM master ports. The data master port connects to both memory and peripheral components, while the instruction master port connects only to memory components.

Both data memory and peripherals are mapped into the address space of the data master port. The Nios II architecture is little endian. Words and halfwords are stored in memory with the more-significant bytes at higher addresses. The address map for memories and peripherals in a Nios II processor system is design dependent. The user specifies the address map at system generation time.

Programmers access memories and peripherals by using macros and drivers. Therefore, the flexible address map does not affect application developers.

Typically, Nios II processor systems contain a mix of fast on-chip memory and slower off-chip memory. Peripherals typically reside on-chip, although interfaces to off-chip peripherals also exist.

Instruction Master Port

The Nios II instruction bus is implemented as a 32-bit Avalon-MM master port. The instruction master port performs a single function: it fetches instructions to be executed by the processor. The instruction master port does not perform any write operations. It is a pipelined Avalon-MM master port. The instruction master port can issue successive read requests before data has returned from prior requests. The Nios II processor can prefetch sequential instructions and perform branch prediction to keep the instruction pipe as active as possible.

The instruction master port always retrieves 32 bits of data. It relies on dynamic bus-sizing logic contained in the system interconnect fabric. By virtue of dynamic bus sizing, every instruction fetch returns a full instruction word, regardless of the width of the target memory. The Instruction Master Port can be cached.

Data Master Port

The Nios II data bus is implemented as a 32-bit Avalon-MM master port. The data master port reads data from memory or a peripheral when the processor executes a load instruction and it writes data to memory or a peripheral when the processor executes a store instruction. Byte-enable signals on the master port specify which of the four byte-lane(s) to write during store operations.

The Data Master Port can be cached. When the Nios II core is configured with a data cache line size greater than four bytes, the data master port supports pipelined Avalon-MM transfers. When the data cache line size is only four bytes, any memory pipeline latency is perceived by the data master port as wait states. Load and store operations can complete in a single clock-cycle when the data master port is connected to zero-wait-state memory.

JTAG Debug Module

The Nios II architecture supports a JTAG debug module that provides on-chip emulation features to control the processor remotely from a host PC. PC-based software debugging tools communicate with the JTAG debug module and provide facilities, such as the following features:

- Downloading programs to memory
- Starting and stopping execution
- Setting breakpoints and watchpoints
- Analyzing registers and memory
- Collecting real-time execution trace data

The debug module connects to the JTAG circuitry in an FPGA. The soft-core nature of the Nios II processor allows the user to debug a system in development using a full-featured debug core, and later remove the debug features to conserve logic resources.

The System Interconnect Fabric for Memory-Mapped Interfaces

To effectively exchange information, the IP cores in the system have to be connected among them by a reliable and efficient communication medium. Components of a Nios II based system communicate among them by means of a communication infrastructure called the *"system interconnect fabric"*.

The System interconnect fabric for memory-mapped interfaces implements a partial crossbar interconnect structure that provides concurrent paths between master and slaves. System interconnect fabric consists of synchronous logic and routing resources inside the FPGA.

In the path between master and slaves, the system interconnect fabric might introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by the specific interfaces.

The System interconnect fabric can connect any combination of components. It consumes minimal logic resources, provides greater flexibility, and higher throughput than a typical shared system bus.

The system interconnect fabric logic provides the following functions:

- Address Decoding
- Datapath Multiplexing
- Wait State Insertion
- Pipelined Read Transfers
- Arbitration for Multi-master Systems
- Burst Adapters

- Interrupts
- Reset Distribution

Arbitration for Multi-master Systems

The system interconnect fabric supports systems with multiple master components. The system interconnect fabric provides shared access to slaves using a technique called *slave-side arbitration*. If a system contains multiple masters (e.g. two processors or a processor and a direct memory access (DMA) peripheral), SOPC Builder automatically generates slave-side arbitration technology to optimize multi-master system performance. Slave-side arbitration moves the arbitration logic close to the slave where it determines which master gains access to a specific slave in the event that multiple masters attempt to access the same slave at the same time. The connection between a master and a slave exists only if it is necessary and is specified in by the designer. If a master never initiates transfers to a specific slave, no connection is necessary, and therefore no logic resources are wasted to connect the two ports. It eliminates unnecessary master-slave connections.

Slave-side arbitration allows multiple masters to transfer data simultaneously. Unlike traditional *host-side arbitration* architectures where each master has to wait until it is granted access to the shared bus, multiple masters can simultaneously perform transfers with independent slaves with slave-side arbitration scheme. Arbitration logic stalls a master only when multiple masters attempt to access the same slave during the same cycle.

The interconnect fabric provides configurable arbitration settings, and arbitration for each slave is specified independently. For example, the user can grant one master more arbitration shares than others, allowing it to gain more access cycles to the slave. The arbitration share settings are defined for each slave independently.

Slave-Side Arbitration VS Traditional Shared Bus Architectures

In traditional bus architectures, one or more bus masters and bus slaves connect to a shared bus, consisting of wires on a printed circuit board or on-chip routing. A single arbiter controls the bus (that is, the path between bus masters and bus slaves), 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.

Figure 63 illustrates the bus architecture for a traditional processor system. Access to the shared system bus becomes the bottleneck for throughput. Only one master has access to the bus at a time, which means that other masters are forced to wait and only one slave can transfer data at a time.

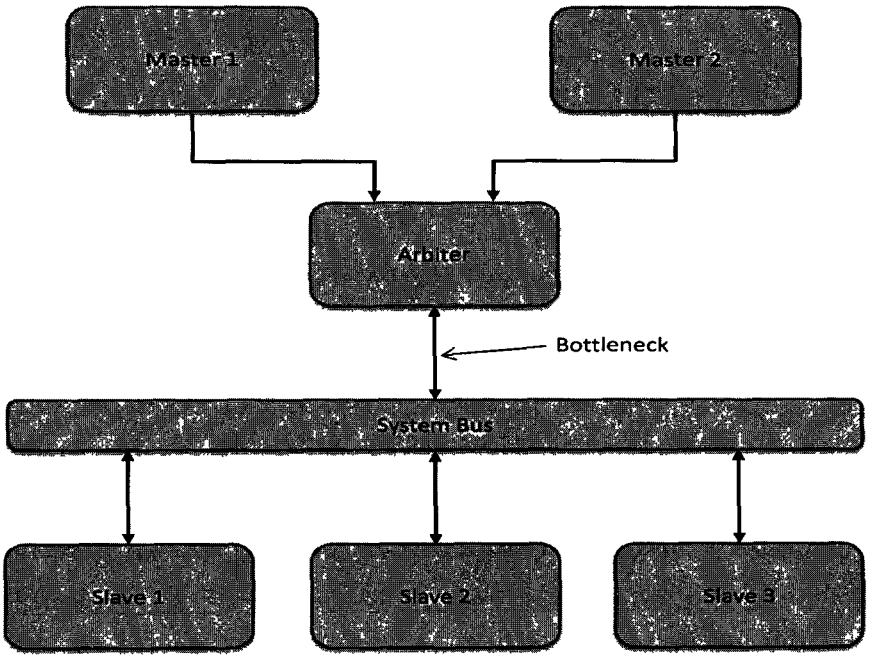


Figure 63: Common Bus Multi-Master Connection with Host Side Arbitration

With the slave-side arbitration the system interconnect fabric uses multimaster architecture to eliminate this bottleneck. Multiple masters can be active at the same time, simultaneously transferring data with independent slaves. For example, **Figure 64** demonstrates a system with two masters sharing a slave. Arbitration is performed at the slave. The arbiter dictates which master gains access to the slave if both masters initiate a transfer with the slave in the same cycle. The arbiter logic multiplexes all address, data, and control signals from a master to a shared slave.

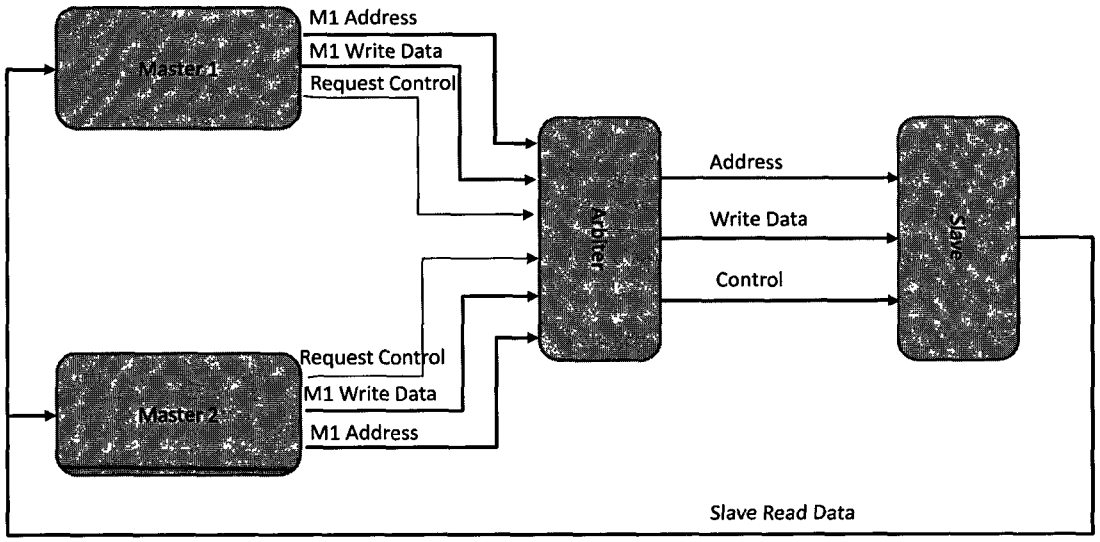


Figure 64: System Interconnect Multi-Master Connection with Slave Side Arbitration

The arbiter logic evaluates the address and control signals from each master and determines which master, if any, gains access to the slave next. It grants access to the chosen master and forces all other requesting masters to wait. The arbiter uses multiplexers to connect address, control, and datapaths between the multiple masters and the slave.

Arbitration Rules

To allocate access intervals (shares) to different masters, the arbiter logic uses a fairness-based arbitration scheme. Each master pair has an integer value of transfer *shares* with respect to a slave. One share represents permission to perform one transfer. For example, assume that the designer has assigned three shares to Master 1 and four shares to Master 2. In this case if two masters continuously attempt to perform back-to-back transfers to a slave, the arbiter grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. **Figure 65** demonstrates this case, showing each master's transfer request output, wait request input (which is driven by the arbiter logic), and the current master with control of the slave.

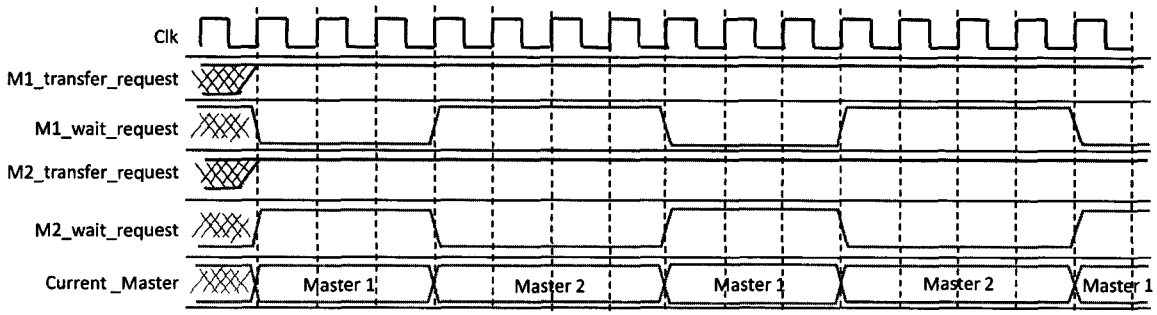


Figure 65: Arbitration of Continuous Transfer Requests from Two Masters

If a master stops requesting transfers before it exhausts its shares, it forfeits all its remaining shares, and the arbiter grants access to another requesting master as illustrated in **Figure 66**. After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbiter grants access back to Master 1, which gets a replenished supply of its shares.

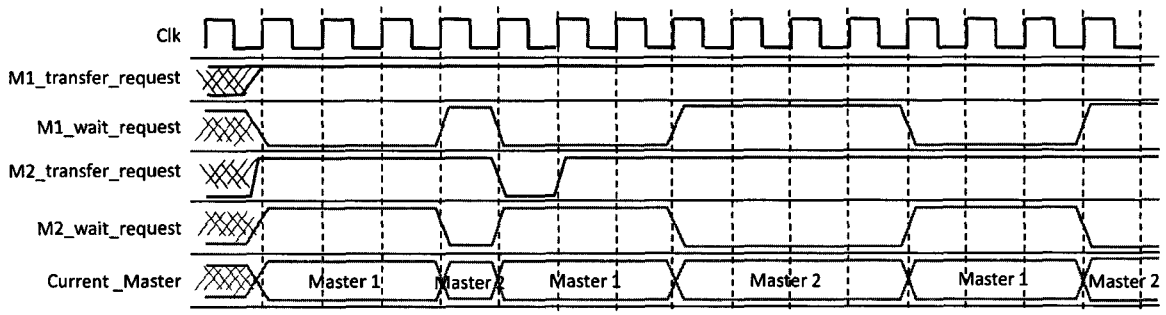


Figure 66: Arbitration of Two Masters with a Gap in Transfer Requests

When multiple masters contend for access to a slave, the arbiter grants shares in round-robin order. At every slave transfer, only requesting masters are included in the arbitration.

Bibliography

1. **ST Microelectronics.** *STMicroelectronics and Mobileye Deliver Second-Generation System-on-Chip for Vision-Based Driver Assistance Systems.* Geneva : http://www.webwire.com/ViewPressRel.asp?aId=65141, 2008.

2. **ITF.** *International Transportation Forum Statistics.* : http://www.internationaltransportforum.org/shorttermtrends/, . .

3. **Rediff Report.** *India tops chart in road fatalities.* : http://www.rediff.com/news/2004/jan/03road.htm, 2004. .

4. **EU Commission Report.** *Toward Fair and Efficient Pricing in Transport, Green Paper.* Brussels : COMMISSION OF THE EUROPEAN COMMUNITIES, 1995. COM(95) 691 final.

5. **R. PARASURAMAN, P. A. HANCOCK and O. OLOFINBOBA.** *Alarm effectiveness in driver-centred collision-warning systems.* : Taylor & Francis Ltd , 1997. ERGONOMICS, 1997, VOL. 40, NO. 3, 390-399.

6. **Automotive DesignLine.** *Tech Tutorial: Driver Assistance Systems, an introduction to Adaptive Cruise Control .* : http://www.automotivedesignline.com/189600772?printableArticle=true, 2006. .

7. **E. Farber and M. Paley.** *Using freeway traffic data to estimate the effectiveness of rear end collision countermeasures.* Washington, DC : Third Annual IVHS America Meeting, IVHS America, 1993. .

8. **R. R. Knipling et al.** *Assessment of IVHS countermeasures for collision avoidance: rear-end crashe.* Washington DC : Third Annual IVHS America Meeting, National Highway and Traffic Safety Administration, 1993. NTIS No. DOT-HS-807-995.

9. **A. Iioshi.** *Driver Assistance System (Lane Keep Assist System), Honda HiDS.* Geneva : Presentation WP-29 ITS Round Table, 2004.

10. **K. Ch. Fuerstenberg et al.** *Development of a Pre-Crash sensorial system – the CHAMELEON Project.* s.l. : http://www.ibeo-as.com/english/downloads/publications/VDI_2001_Fuerstenberg_Wolfsburg.pdf, 2003.

11. **NEC Web Magazine.** *The IMAPCAR parallel processor for image recognition and its contribution to the realization of pre-crash safety solutions for automobiles.* : http://www.necel.com/magazine/en/vol_0063/vol_0063_1.html, 2006.

12. **PReVENT Project Final Report.** *http://www.prevent-ip.org/PR-04000-IPD-080222-v15_PReVENT_Final_Report.* 2008.

13. **K. Fuerstenberg, B Roessseler.** *Intersection Driver Assistance Systems - Results of the EC-Project INTERSAFE.* Istanbul : IEEE Intelligent Vehicle Symposium Istanbul, Turkey, 2007.

14. **SEAT.** *http://www.auto-innovations.com/actualite/actu-z032securite.html, Prototype SEAT Alhambra ADAS (Advanced Driver Assistance Systems).* Madrid, 2003.

15. **Project SARI, PREDIT.** *http://www.sari.prd.fr/index.html, Automatic Road Condition Monitoring to Provide Information to Drivers and Road Managers.* , 2006.

16. **N. Kaempchen , K. Fuerstenberg et al.** *Advanced Microsystems for Automotive Applications, Chapter: Sensor Fusion for Multiple Automotive Active Safety and Comfort Applications* . Berlin : Springer Berlin Heidelberg, 2004. 978-3-540-20586-9 (Print) 978-3-540-76989-7 (Online).

17. **J. Miura, M. Itoh and Y. Shirai.** *Towards Vision-Based Intelligent Navigator... Its Concept and Prototype.* s.l. : IEE transaction on Intelligent transportation Systems,Vol.3, No.2, pp136-146, 2002.

18. **J. Miura, T. Kanda, and Y. Shirai.** *An Active Vision System for Real-Time Traffic Sign Recognition.* Dearborn, MI : Proc. 2000 IEEE Int. Conf. on Intelligent Transportation Systems, pp 52-57, 2000.

19. **Z. Salcic et al.** *Scalar-based direct algorithm mapping FPLD implementation of Kalman Filter.* : IEEE Transactions on Aerospace and Electronic Systems, 2000. .

20. **Z. Salcic and C-R Lee.** *FPGA-Based Adaptive Tracking Estimation Computer.* : IEEE transactions on aerospace and electronic Systems, 2001. .

21. **P. Konstantinova et al.** *A study of Target Tracking Algorithm Using Global Nearest Neighbor Approach.* : CompSysTech, 2003. .

22. **Y. Boismenu.** *Etude d'une carte de tracking radar, Thse de doctorat.* : Universit de Bourgogne, France, 2000. .

23. **A. Goransson and B. Sohlberg.** *Tracking Low Velocity Vehicles from Radar Measurements.* Cancun, Mexico : Proceedings Circuits, Signals and Systems (CSS), 2003.

24. **G. Chen and L. Guo.** *The FPGA implementation of Kalman filter.* Malta: Proceedings of the 5th WSEAS International Conference on Signal Processing, Computational Geometry & Artificial Vision, 2005. ISBN:960-8457-35-1.

25. **NEC Electronics.** *NEC electronics and NEC introduce imapcar image processor with advanced parallel processing capabilities for automotive safety systems new processor adopted in lexus ls460 precrash safety system. Press release,.* s.l.: http://www.nec.co.jp/press/en/0608/2501.html, 2006.

26. **M. Ooishi.** *New Toyota Lexus Detects Pedestrians, Applies Brakes.* s.l.: Nikkei Electronics Asia, on line journal, 2006.

27. **NEC Electronics.** *NEC IMAPCAR2 Image Processor.* s.l.: ElectronicsWeekly.com, 2008.

28. **P. Leteinturier.** *Multi-Core Processors: Driving the Evolution of Automotive Electronics Architectures.* : Infineon technologies, Embedded.com, 2007. .

29. **Freescale Semiconductor.** *MPC5561: 32-bit PowerPC Microcontrollers.* s.l.: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC5561&nodeId=016246 8rH3bTdG06C10325&tid=tevpr&tid=tApr5561, 2009.

30. **Texas Instruments Incarporated (TI).** *TMS570F MCUs with dual ARM Cortex-R4F CPUs reduce complexity and time-to-market for braking, steering, advanced driver assistance and chassis control* . Houston : www.ti.com/tms570f, 2008.

31. **Freesacle Semiconductor.** *C-5 Network Processor Architecture Guide.* : http://www.freescale.com/files/netcomm/doc/ref_manual/C5NPD0-AG.pdf?fsrch=1, 2001. .

32. **PHILIPS.** *Home Entertainment Engine, Nexperia pnx8500.* : http://www.nxp.com/acrobat_download/other/nexperia/pnx8500-1000.pdf, PHILIPS, 2000. .

33. **T. Spits and P. Werp.** *OMAP Technology Overview, White Paper SWPY001.* , 2000.

34. **ARM.** *ARM11 MPCore - ARM Processor.* : http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html, 2009. .

35. **N. Blachford.** *Cell Architecture Explained Version 2.* : http://www.blachford.info/computer/Cell/Cell0_v2.html, 2005. .

36. **A. Jerraya, W. Wolf.** *Multiprocessor Systems-on-Chips* . : Morgan Kaufmann, 2004. ISBN-10: 012385251X.

37. **Altera.** *Why Use FPGAs in Embedded Designs?* : http://www.altera.com/technology/embedded/fpgas/emb-why-use.html, 2009. .

38. **C. Maxfield.** *The Design Warrior's Guide to FPGAs, Devices, Tools and Flows.* : Elsevier, 2004. ISBN: 0-7506-7604-3.

39. **J. Jussel.** *Survey Shows Hidden Market for ESL and FPGA; Electronic System Level Design and FPGAs Are the Big Winners in This Year's Worldwide Survey on Designer Trends.* Abingdon, England : http://www10.edacafe.com/nbc/articles/view_article.php?section=CorpNews&articleid=202531 &interstitial_displayed=Yes, 2005. .

40. **J.Turley.** *Survey: Who uses custom chips Embedded Systems Design.* s.l. : Embedded Systems design, http://www.embedded.com/columns/surveys/166404172?_requestid=379134, 2005.

41. **J. Turley.** *Survey says: software tools more important than chips.* : Embedded Systems Design, http://www.embedded.com/columns/surveys/160700620?_requestid=380127, 2005. .

42. **E. Heikkila , J. Eric Gulliksen.** *A White Paper On: Multi-Core Computing in Embedded Applications.* : Venture Development Corporation, 2007. .

43. **A. Jantsch et al.** *A NOC Architecture and Design Methodology.* Pittsburgh, PA, USA : IEEE Computer Society, Annual Symposium on VLSI, 2002. ISBN 0-7695-1486-3.

44. **S. Blackman and R. Popoli.** *Design and analysis of modern tracking systems* . : Artech House Publishers , 1999. ISBN 1-58053-006-0.

45. **TRW AC20.** *TRW Autocruise AC20 radar.* s.l. : Elsevier Ltd., ScienceDirect, 2004. doi:10.1016/S0961-1290(04)00812-9 .

46. **E. Brookner.** *Tracking and Kalman Filtering Made Easy.* : John Wiley, 1998. ISBN: 978-0-471-18407-2.

47. **G. Welch and G. Bishop.** *An introduction to the Kalman Filter.* : http://www.cs.unc.edu/ welch/kalman/, 2001. .

48. **V. Nedovi.** *Tracking moving video objects using meanshift algorithm, project report.* : http://staff.science.uva.nl/ vnedovic/, 2004. .

49. **R. E. Kalman.** *A New Approach to Linear Filtering and Prediction Problems.* : Transaction of the ASME–Journal of Basic Engineering, 1960. .

50. **D. P. Bertsekas, D. A. Castaon.** *A Forward/Reverse Auction Algorithm for Asymmetric Assignment Problems.* : Springer Link Journal of Computational Optimization and Applications, 1992. ISSN: 0926-6003 (Print) 1573-2894 (Online).

51. **M. M. Zavlanos, L. Spesivtsev and G. J. Pappas.** *A Distributed Auction Algorithm for the Assignment Problem.* Cancun, Mexico : 47th IEEE Conference on Decision and Control, 2008. .

52. **James Munkres.** *Algorithms for Assignment and Transportation Problems .* : Journal of the Society for Industrial and Applied Mathematics Volume 5, Number 1, 1957. .

53. **F. Burgeois and J.-C. Lasalle.** *An extension of the Munkres algorithm for the assignment problem to rectangular matrices.* : Communications of the ACM, 1971. 142302-806.

54. **S. Eckel.** *USC Health Science Course. Biostatistics II: Biostatistical Modelling .* : http://www-hsc.usc.edu/~eckel/biostat2/, 2008. .

55. **R. Burkard, M. Dell'Amico, and S. Martello.** *Assignment Problems.* Philadelphia : SIAM Books, 2009. ISBN: 978-0-898716-63-4 .

56. **Altera Corporation.** *Nios Development Board Stratix II Edition.* : Altera Caroporation, Nios Development Board Stratix II Edition, 2007. .

57. **F. Schirrmeister.** *Multi-core Processors: Fundamentals, Trends, and Challenges.* San Jose, California : Embedded Systems Conference, 2007. .

58. **White Paper Altera.** *FPGA Architecture.* : Altera Corp., 2006. .

59. **Altera.** *Nios II Performance Benchmarks.* : Altera, 2009. .

60. **Altera Corporation.** *Nios II Processor Reference Handbook.* : Altera Corporation,http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf, 2009. .

61. **Altera Corporation.** *Using Nios II Floating-Point Custom Instructions.* : Altera Corporation, http://www.altera.com/literature/tt/tt_floating_point_custom_instructions.pdf, 2006. .

62. **Altera Corporation.** *AN 459: Guidelines for Developing a.* : Altera Coroporation, http://www.altera.com/literature/an/an459.pdf, 2008. .

63. **Altera Corporation.** *Application Note 391, Profiling Nios II Systems.* s.l. : Altera Corporation, http://www.altera.com/literature/an/an391.pdf, 2008.

64. **Altera Corporation.** *Avalon Switch Fabric.* : Altera Corporation, http://www.ee.ryerson.ca/~courses/coe718/Data-Sheets/sopc/AVALONBUS_qii54003.pdf, 2006. .

65. **Altera Corporation.** *Nios II C2H Compiler User Guide.* : Altera Corporation, http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf, 2008. .

66. **W. Wolf, A. A. Jerraya and G. Martin.** *Multiprocessor System-on-Chip (MPSoC) Technology* . : IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems , 2008. .

67. **Intel Corp.** *Product Brief, Intel IXP2855 Network Processor.* : On line Ducument, . .

68. **K. M. Popvici.** *Environnement de Programmation Multi Niveau pour Architectures Hétérogènes MPSoC* . : thèse pour obtenir le grade de docteur de L'INP Grenoble, France, 2008. .

69. **V.A.W.J. Marchau , R.E.C.M. van der Heijden, E.J.E. Molin.** *Desirability of advanced driver assistance from road safety perspective: the case of ISA.* s.l. : www.sciencedirect.com, 2005. Safety Science 43 (2005) 11–27.