

# Driver Assistance System Design and its Optimization for FPGA Based MPSoC

Jehangir Khan, Smail Niar, Mazen Saghir<sup>‡</sup>, Yassin El-Hillali and Atika Rivenq  
University of Valenciennes, France.

Email: {jehangir.khan, smail.niar, yassin.elhillali, atika.menhaj}@univ-valenciennes.fr

<sup>‡</sup>Texas A&M Engineering Building, Education City, Doha, Qatar  
Email: mazen.saghir@qatar.tamu.edu

**Abstract**—This paper discusses the design and optimization of an FPGA based MPSoC dedicated to Multiple Target Tracking (MTT) application. MTT has long been used in various military and civilian applications but its use in automotive safety has been little investigated. The system attains a desired processing speed and uses minimum FPGA resources. We profile the software to identify performance bottlenecks and then gradually tune the hardware and software to meet system constraints. The result is a complete embedded MTT application running on a multiprocessor system that fits in a contemporary medium sized FPGA device.

## I. INTRODUCTION

Adverse weather, low visibility conditions, misjudgment of delicate situations and physical or mental stress are among the reasons that put the lives of automotive drivers and their passengers in danger. Relieving the driver of stressful driving environment guarantees a decrease in road accidents. In this context, Driver Assistant Systems (DAS) can help drivers take correct and quick decisions in delicate situations. Use of Multiple Target Tracking (MTT) enhances the effectiveness of DAS's. We use radar as sensor in our application because it has the advantages of longer range as compared to camera based systems. It performs better in bad visibility conditions and has lower computational requirements [1]. We designed our MTT system as an application specific MPSoC implemented in FPGA that makes it easily evolvable and cost effective. Multiple processors with a lower frequency results in comparable overall performance while allowing to slow down the clock speed, which is a major requirement for low power designs [2]. Every module of the application is mapped onto a processor that best suits its performance requirements and performance critical code sections is placed in the fast on-chip memories. We also adjust the hardware features such that the system achieves the desired performance while using minimum configurable logic on the FPGA.

## II. THE MTT APPLICATION DESIGN

In the context of target tracking applications, a *target* represents an obstacle in the way of the host vehicle. Every obstacle has an associated *state* represented by a vector that contains parameters defining target's position and its dynamics in space (e.g. its distance, speed, azimuth or elevation etc). A concatenation of target states defining the target trajectory or movement history at discrete moments in time is called a *track*.

Target tracking deals with 3 quantities: 1) *The Observation*, which corresponds to the measurement of a target's state by a sensor (radar) at discrete moments in time. 2) *The prediction* of the target's state before the observation arrives. 3) Taking into account the observation and the prediction, an *estimate* about the true target state is made. In this paper, the term *scan* is used to specify the periodic sweep of radar field of view (FOV) giving observations of all the detected targets. A Multiple Target Tracking (MTT) system can broadly be divided into two main blocks namely Data Association and Tracking Filters as shown in Figure 1. The data association block is further divided into three sub-blocks: *Track maintenance*, *Observation-to-Track Assignment* and *Gate Computation* [3], [4].

We organized the application into sub-modules (Fig. 1). The functioning of the system is explained as follows. Assuming recursive processing as shown by the outer loop in Figure 1, tracks would have been formed on the previous radar scan. When new observations are received from the sensor, the processing loop is executed [5], [6]:

A) **Gate Checker**: Incoming observations are first considered by the *Gate Checker* for updating the existing tracks. Gate checking determines which *observation-to-track* pairings are probable.

B) **Cost Matrix Generator**: This module associates a cost with every possible observation-prediction pair. The cost  $c_{ij}$  of associating an observation  $i$  with a prediction  $j$  is the statistical distance  $d_{ij}^2$  between the observation and the prediction when the pairing is probable. The cost matrix is solved by the assignment algorithm to obtain a one-to-one coupling between predictions and observations.

C) **Assignment Solver**: The assignment problem is stated as follows. Given a cost matrix of elements  $c_{ij}$ , 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:  $\sum_i x_{ij} = 1, \forall j$ , and  $\sum_j x_{ij} = 1, \forall i$

Here  $x_{ij}$  is a binary variable used for ensuring that an observation is associated with one and only one track and a track is associated with one and only one observation. The most commonly used algorithm for solving the assignment problem are the Munkres algorithm [8] and Auction algorithm [10]. We use the former in our application since it is inherently modular.

D) **Tracking Filters**: This block is particularly important as

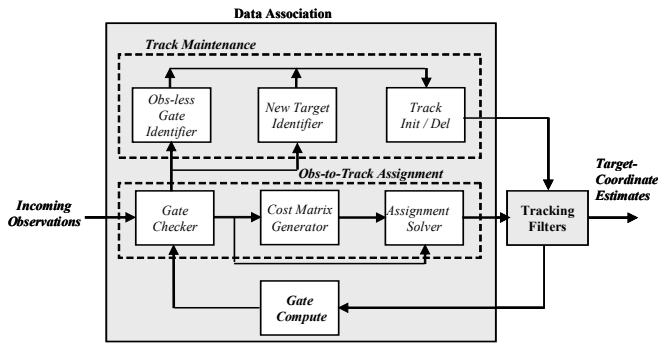


Fig. 1. The Proposed MTT implementation

the number of filters employed is the same as the maximum number of targets to be tracked. In our current work we have fixed this number at 20 because the radar we are using can measure the coordinates of a maximum of 20 targets. Hence this block uses 20 similar filters. We use Kalman filters for this block since it is considered to be the optimum recursive Least Square Estimator (LSE) for Gaussian systems [7].

E) **Gate Computation:** This sub-block receives state prediction and error covariance prediction from the tracking filters for all the targets. Using these two quantities the "Gate Compute" block defines the probability gates or windows which are used to verify whether an incoming observation can be associated with an existing track.

F) **Track Maintenance:** In real conditions new targets may appear and some of the targets being tracked would no more be in the radar FOV. The first case is the "New target Identification" whereas the latter one is the "Observation-less Gate Identification" case. A new target is identified when its observation fails all the already established gates. The case of Observation-less Gate indicates the disappearance of a target from radar FOV. The *Obs-less Gate Identifier* looks for consecutive *misses* in a given number scans to confirm the disappearance of a target. The *Track Init/Del* initiates new tracks or deletes existing ones when needed.

### III. SYSTEM ARCHITECTURE

The proposed multiprocessor architecture includes different versions of the NiosII [9] processor and various peripherals. We coded the application in ANSI C and distributed it over different processors as distinct functions communicating in a producer-consumer manner, Figure 2. The Kalman filter, as mentioned earlier, is a recursive algorithm looping around prediction and correction steps that involve matrix operations on floating point numbers. This makes the filter a strong candidate for mapping onto a separate processor. Consequently for tracking 20 targets at a time, we need 20 identical processors executing the filters. The assignment solver is an algorithm consisting of six distinct iterative steps [8]. Looping through these steps takes an execution time almost five times that for the Kalman filter, hence this function cannot be combined with any of the other functions. Thus the assignment solver is another strong candidate for mapping onto a separate processor.

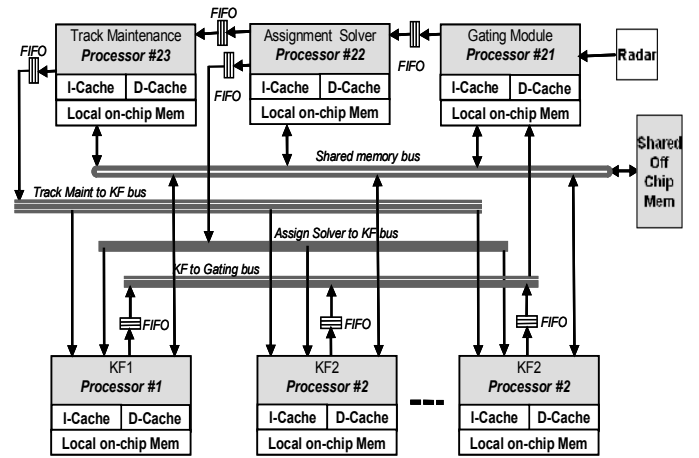


Fig. 2. The proposed MPSoC architecture

The "Gate Compute" block regularly passes information to "Gate Checker" which in turn, is in constant communication with "Cost Matrix Generator". So, we group these three blocks together, and collectively call them the "Gating Module" and map them onto a single processor to minimize inter-processor communication. The three blocks of the *Track Maintenance* sub-function don't demand heavy computational resources, so we group them together for mapping onto a processor.

### IV. CONSTRAINTS AND OPTIMIZATION STRATEGIES

We have to meet three main constraints in our system: the overall response time of the system, the limited amount of on chip memory and the amount of used configurable logic. The response time of the system must be less than 25ms (the Radar Pulse Repetition Time). Consequently, the latency of the slowest application module must be less than 25ms. The FPGA we are using, an Alteras stratixII, contains a total of about 500KB of configurable on chip block 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. Hence, 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 imposed by the radar. We must choose our hardware components carefully to avoid unnecessary use of the programmable logic on the FPGA. To meet these constraints, we have to determine the total memory requirements of the application, the optimum combinations of instruction and data caches, the processor types and the need for custom hardware etc. This implies exploring the "space" of all possible configurations of the MPSoC architecture and choosing the one that best meets the constraints. To reduce the time for this *design space exploration*, we tackle the hardware configurations individually one after the other. In the futur, heuristics could be used to avoid the exhaustif exploration of the configuration space. All the performance measurements cited in the rest of the paper are obtained through performance counters.

TABLE I  
MEMORY REQUIREMENTS OF THE APPLICATION MODULES

Section Name	Memory Foot Print
Kalman Filter	
Whole Code + Initialized Data	81KB
.text Section alone	69.6KB
.data Section alone	10.44KB
Gating Module	
Whole code + initialized data	63KB
.text section alone	51.81KB
.data section alone	8.61KB
Munkres Algorithm	
Whole code + initialized data	62KB
.text section alone	52.34KB
.data section alone	10.44KB

#### A) Choice of NiosII type processor and cache sizes:

The NiosII processor comes in three customizable micro-architectures. They differ in their performance and the amount of FPGA resources they require. The 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 in terms of speed and logic resource requirements [9]. Cache memory resides on-chip as an integral part of the Nios II processor core, and the optimal cache configuration is application specific. We experimented with various combinations of I-cache and D-cache sizes to determine the optimum cache sizes for each module.

**B) Floating point custom instructions:** The floating-point custom instructions, optionally available on the NiosII processor, implement single precision floating-point arithmetic operations in hardware. While the floating-point custom instructions speed up floating-point arithmetic, they increase substantially the size of the hardware design. Hence floating point custom instructions must be used with care.

**C) On-chip Vs off-chip memory sections:** The linker script generated by NiosII IDE, creates standard code and data sections (.text, .data, stack, heap and .bss). Typically the .data section is the part of the object module that contains initialized static data e.g. initialized static variables, string constants in C. The .bss section defines the space for non initialized static data. The heap section is usually used for dynamic memory allocation The stack section is used for storing the processor state when function calls occur. We can place any of these memory sections in the on-chip RAM if needed to achieve desired performance. However as the amount of the on-chip memory in the FPGA is very limited, we have to rely on the off-chip SDRAM or SSRAM. Table I summarizes the memory requirements of the application modules. Note that, each of the stack and heap sections requires respectively 2KB and 1 KB for each of the 3 software components. The memory requirements of the whole code and the .text sections for all the modules are too large to be accommodated in the on-chip memory. We performed experiments by placing the memory sections for the different modules in the off-chip and on-chip memories. These results are discussed in the following sections.

## V. COMPONENT OPTIMIZATION

**A) Kalman Filter Optimization:** Figure 3 shows the influence of I-cache and D-cache sizes on the processor time of the Kalman filter algorithm running on NiosII/f with 100MHz clock using off-chip RAM.

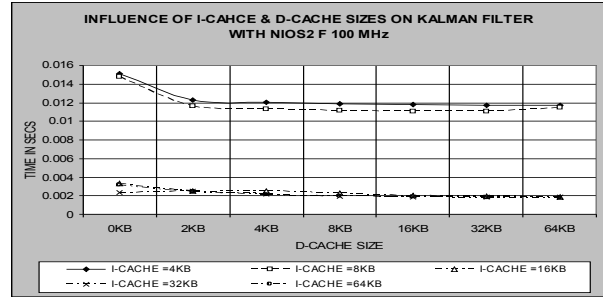


Fig. 3. Kalman Filter performances for different Caches Sizes

Whatever the I-cache or D-cache size, the processor time hardly exceeds 15ms. With a 4KB I-cache and no D-cache the processor time is below the 25ms threshold set by the radar PRT. Furthermore, using NiosII/s instead of NiosII/f help reduce the configurable resource consumption from 1800 to 1200 logic elements. The on-chip block memory usage in this case is only 3% of that exploitable on the FPGA. For 20 Kalman processors the total on-chip memory usage is 60% of that available to the user.

**B) Gating Module Optimization:** For the gating module a remarkable speed up is observed when I-cache size changes from 4KB to 8KB and again when it changes from 8KB to 16 KB. Beyond 16KB I-cache the speed up is insignificant. The D-cache size does not matter much as long as it is more than zero. The smallest overall processor run time (70ms) is obtained when I-cache size is 16KB and D-cache size is 2KB. The total on-chip block memory usage for this processor adds up to 8% of that available on the FPGA. The 70ms run time for 20 obstacles is much greater than the 25ms we are aiming for. Hence in the case of the "gating module" the use of floating point custom instructions is a necessity. Moreover, this module is run on one processor so we don't have to replicate the floating point custom instructions hardware either. The overall performance improves by approximately 50% after adding floating point custom instructions (figure 4). To improve the performance further, we experimented with placing various memory sections in the on-chip memory. Figure 4 highlights Gating module's performance for different memory placement experiments. Here we can deduce that the highest speed of 22ms can be achieved by keeping all the memory sections in the on-chip memory. But this would require the on-chip RAM to be at least 61KB. This combined with the on-chip memory taken up by the I-cache and D-cache sums up to 79 KB. The next best solution, 23 ms is obtained when we place the stack and heap sections in the on-chip memory. There is a very little speed loss but in this case only 3KB of dedicated on-chip memory is sufficient to get this speed up. So the Gating module can operate satisfactorily by using a NiosII/F processor with

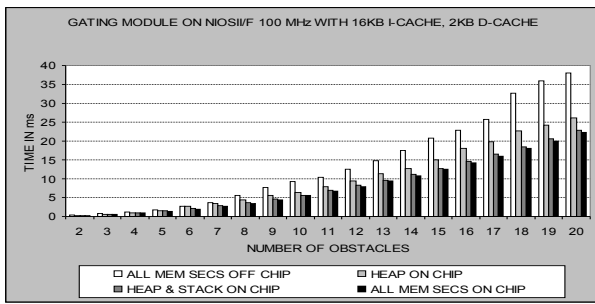


Fig. 4. Effects of on-chip and off-chip memory sections on Gating Module

16KB I-cache, 2KB D-cache, 3KB dedicated on-chip RAM and floating point custom instructions.

C) **Munkres Algorithm Optimization:** This module manifested the behavior presented in Figure 5. When D-cache size

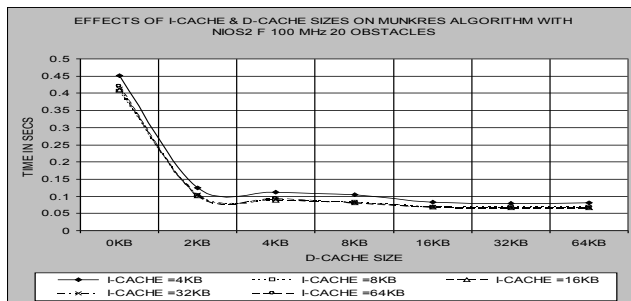


Fig. 5. Cache performances for Munkres Algorithm

is more than 0, whatever the I-cache size the run time drops down profoundly. An 8KB I-cache along with 16KB D-cache offers the minimum execution time i.e. 71.07ms. Hence this is the optimum I-cache/D-cache configuration for this module. A NiosII system with these cache sizes uses 9% of the on-chip block memory available on the FPGA. Floating point custom instructions bring Munkres algorithm's execution time from 71ms down to 47 ms for 20 obstacles. However, when the floating point cost matrix is replaced by a "representative" integer cost matrix, we don't sacrifice 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. Additionally, using an integer cost matrix obviates the need for the floating point custom instructions consequently 8% of the FPGA logic is conserved. (Figure 6). Step1 through Step6 are the constituent sub functions of Munkres algorithm. The Call to Munkres signifies the total run time of the algorithm including the six sub functions. The point worth noticing here is that with integer cost matrix, the run time for the overall algorithm drops down to 24ms as opposed to the 71ms with floating point cost matrix. So the final solution to Munkres algorithm's defiance is the use of a cost matrix with integer elements and mapping the algorithm to a NiosII/F processor with 8KB I-cache and 16KB D-cache. Note here that it is also possible to implement the the Munkres algorithm in

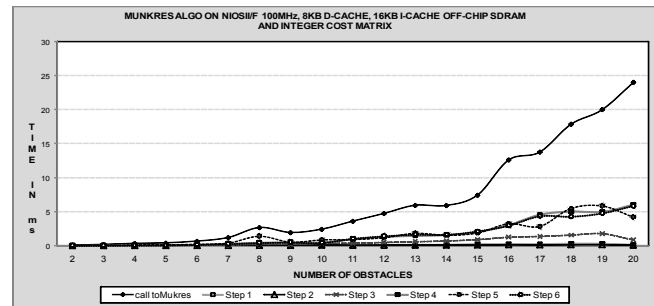


Fig. 6. Cache Behavior for Munkres Algorithm with Integer Cost Matrix

hardware. However, our experiments show that this solution can effectively accelerate the execution but the cost in term of hardware logic is prohibitively high.

## VI. CONCLUSION

We described the general view of the MTT application and then we presented our own approach to the design and development of the application. Next we outlined the preliminary architecture of the system without going into the minute details. To plan our optimization strategies, we first identified the constraints to be respected. Looking for potential bottlenecks in the application we profiled the application. Using hardware counters, several hardware and software optimization have been experimented: cache size tuning, floating point custom instruction integration, floating point to integer conversion, segment memory mapping, etc. We traded speed for FPGA resource economy where we could afford it. Guided by the results of our experiments, we finalized the components and their respective features. The whole system fits in a single FPGA and meets the time constraints enforced by the radar used in the system. The system achieves the overall performance of 25 ms for tracking a maximum of 20 obstacles and uses 81% of the total available logic elements (60K) of the Altera StartixII 2s60.

## REFERENCES

- [1] Z. Salcic and C.R. Lee, FPGA-Based Adaptive Tracking Estimation Computer, IEEE transactions on aerospace and electronic systems, 2001.
- [2] F. Schirmeister, Multi-core Processors: Fundamentals, Trends, and Challenges, Embedded Systems Conference 2007 ESC351.
- [3] S. Blackman and R. Popoli, Design and analysis of modern tracking systems, Artech House Publishers 1999, ISBN 1-58053-006-0.
- [4] Brookner, E. Tracking and Kalman filtering made easy, John Wiley & Sons Inc. 1998.
- [5] J. Khan et al. An MPSoC Architecture for the Multiple Target Tracking Application in Driver Assistant System, 19th IEEE International Conference ASAP08, 2-4 July 2008, Leuven Belgium.
- [6] J. Khan et al. Multiple Target Tracking System Design for Driver Assistant Application, DASIP08, 24-26 November 2008, Brussels Belgium.
- [7] R. E. Kalman, A New Approach to Linear Filtering and Prediction Problems, Transaction of the ASME—Journal of Basic Engineering, pp.35-45(Mar 1960).
- [8] R. A. Pilgrim, Munkres' Assignment Algorithm. Modified for Rectangular Matrices, Course notes, Murray State University.
- [9] Altera Corporation. NIOS II processor reference handbook. www.altera.com/literature/hb/nios2/n2cpu\_nii5v1.pdf
- [10] D. P. Bertsekas, D. A. Castaon, A Forward/Reverse Auction Algorithm for Asymmetric Assignment Problems, Computational Optimization and Applications, Volume 1, Number 3 / December, 1992