Radar Based Collision Avoidance System Implementation in a Reconfigurable MPSoC

Jehangir Khan, Smail Niar, Atika Rivenq and Yassin El-Hillali
Email: {jehangir.khan, smail.niar, atika.menhaj, yassin.elhillali}@univ-valenciennes.fr

Abstract—This paper discusses the design and optimization of an FPGA based MPSoC dedicated to Multiple Target Tracking (MTT) aimed at Driver Assistance applications. The use of MTT in DAS’s has not been sufficiently investigated before. After designing the application we propose a multi-processor system-on-chip (MPSoC) architecture for its physical implementation in FPGA. We formulate strategies to improve the system speed and reduce its demand for configurable resources. We identify performance bottlenecks and gradually optimize 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

Road accidents are primarily caused by misjudgement of a delicate situation by the driver. The main reason behind the drivers inability to judge a potentially dangerous situation correctly, is the mental and physical fatigue due the stressful driving conditions. An automatic early warning and collision avoidance system onboard a vehicle can greatly reduce the pressure on the driver. In the literature, such systems are called Driver Assistance Systems (DASs). Use of Multiple Target Tracking (MTT) enhances the effectiveness of DAS’s. The purpose of target tracking is to collect data from the sensor field of view (FOV) containing one or more potential obstacles of interest and to partition the sensor data into sets of observations, or tracks [1]. 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 [2]. Moreover, radar helps detect obstacles at longer distances and hence ensures longer reaction time for vehicle drivers.

Studies have been done on the isolated parts of MTT system [6], [7] but, to the best of our knowledge, design of the complete MTT system using a configurable platform and its application to automotive safety has not been addressed before. We designed our MTT system as an application specific MPSoC implemented in FPGA. Every module of the application is mapped onto a processor that best suits its performance requirements. Additionally we place performance critical code sections in the fast on-chip memories that are strategically sized according to the requirements of the application. We adjust the hardware features such that the system achieves the desired performance while using minimum configurable logic on the FPGA.

Applications with tight resource-needs and runtime constraints are increasingly resorting to MPSoC architectures. Implementation of the system in FPGA using MPSoC architecture makes it easily evolvable and cost effective. Having multiple processors that execute at lower frequency results in comparable overall performance in terms of instructions per second while allowing to slow down the clock speed, which is a major requirement for low power designs [9].

II. THE MTT APPLICATION

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: The Observation, which corresponds to the measurement of a target’s state by a sensor (radar) at discrete moments in time. It is one of the two representations of the true state of a target. The other representation is the prediction of the target’s true state before the observation arrives. Taking into account the observation and the prediction, an estimate about the true target state is made. Estimate is the corrected state of the target that depends upon the variances of both the observation and the prediction. 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. A text book view of a MTT system and can be found in [1], [5]. Practical implementation and internal details vary depending on the end use and implementation technology.

In Section III, we present a detailed description of our MTT application. Then we move on to architectural aspects of the system which are detailed in section IV.

III. MTT APPLICATION DESIGN: OUR APPROACH

For the purpose of modular implementation, we organized the application into sub-modules as shown in Figure 1. 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.
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. The Gate Checker tests whether an incoming observation fulfills the conditions set by the state prediction and error covariance prediction. In effect, this block sets or resets the binary elements of an NxN matrix M shown below.

\[
M = \begin{bmatrix}
m_{11} & m_{12} & \cdots & m_{1N} \\
m_{21} & m_{22} & \cdots & m_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
m_{N1} & m_{N2} & \cdots & m_{NN}
\end{bmatrix}
\]

Here N is the maximum number of targets and tracks being processed. Matrix M would typically have more than one 1’s in a column or a row. The ultimate goal is to have only one ‘1’ in a row or a column for a one-to-one coupling of observations and predictions. To achieve this goal, the first step is to attach a cost to every possible coupling. This is done by the Cost Matrix Generator block explained next.

B. Cost Matrix Generator

The Cost Matrix Generator 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 \( m_{ij} \) is 1. The cost \( c_{ij} \) is an arbitrary large number when \( m_{ij} \) is 0. The costs are then put together in the cost matrix.

The cost matrix demonstrates a conflict situation where several observations are potential candidates to be associated with a particular prediction and vice versa. To resolve the conflicts, the cost matrix is passed on to the Assignment Solver block which treats it as the assignment problem.

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

\[
C = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} \text{ is minimized}
\]

subject to:

\[
\sum_{i=1}^{n} x_{ij} = 1, \forall j \\
\sum_{j=1}^{m} x_{ij} = 1, \forall i
\]

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. This requires \( x_{ij} \) to be either 0 or 1 i.e. \( x_{ij} \in \{0, 1\} \).

D. Tracking Filters

The Tracking filters block in Figure 1, has to instantiated as many times as the maximum number of targets to be tracked. In our current work we have fixed this number at 20. We use Kalman filters for this block since it is considered to be the optimum recursive Least Square Estimator (LSE) for Guassian systems [3], [4].

The Kalman filter continuously cycles through a prediction-correction loop. In the prediction step, the filter predicts the next state and error covariance associated with the state prediction. In the correction step, it calculates the filter gain and estimates the current state and the error covariance of this estimation.

E. Gate Computation

The Gate Computation sub-block of the data association receives state prediction and error covariance prediction form the tracking filters for all the targets. Using these two quantities it defines the probability gates or windows which are used to verify whether an incoming observation can be associated with an existing track. The dimensions of the gates being dictated by the prediction error covariance, these gates demarcate the probability boundaries for the next state coordinate measurements.
F. Track Maintenance

The Track Maintenance block consists of three functions: the obs-less Gate Identifier, the New Target Identifier and the Track Init/Del. 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. The case of Observation-less Gate indicates the disappearance of a target from radar FOV. The Obs-less Gate Identifier looks for 3 consecutive misses in 5 scans to confirm the disappearance of a target. The Track Init/Del initiates new tracks or deletes existing ones when needed.

IV. PRELIMINARY SYSTEM ARCHITECTURE

The preliminary architecture includes different versions of the NiosII [10] processor and various peripherals as system building blocks.

We coded the application in ANSI C and distributed it over different processors as distinct functions communicating in a producer-consumer fashion as shown in Figure 2.

The Kalman filter involves matrix operations on floating point numbers. In its un-optimized form the every filter is mapped onto a separate processor. Consequently for tracking 20 targets at a time, we propose 20 identical processors executing the filters. We shall decrease this number as much as possible through the optimization techniques discussed latter.

The assignment solver takes an execution time almost five times that for the Kalman filter, hence this function cannot be combined with any of the other functions. Hence we map it onto a separate processor.

The Gate Computation block regularly passes information to Gate Checker which in turn, is in constant communication with Cost Matrix Generator. We group these three blocks together, 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. We use a small local memory for every processor and a large off-chip memory device connected to the common bus as shared memory. The utility of these considerations is discussed in section V-D.

Every processor communicates with the neighboring processors through buffers. These buffers are dual-port FIFOs with handshaking signals indicating when the buffers are full or empty and hence regulating the data transfer between the adjoining processors.

In the final system, the caches will either be I-cache and D-cache or I-cache only. In section V-B we demonstrate how to determine systematically the optimal sizes of these caches and the local memories.

V. CONSTRAINTS AND OPTIMIZATION STRATEGIES

We have to meet three main constraints in our system: first the overall response time of the system, second the limited amount of on chip block memory and third the amount of configurable logic used. The slowest application module must have a response time less than 25ms, the Radar Pulse Repetition Time.

The Stratix II FPGA we are using, contains a total of 318KB of configurable on chip block memory. This memory has to be shared among the memories. The internal registers, peripheral port buffers and locally connected dedicated RAM or ROM. Thus using these scarce resources carefully is the second constraint to be met.

We must choose our hardware components carefully to avoid unnecessary use of the programmable logic on the FPGA. Hence we trade speed for logic conservation where we can, or we modify our software to achieve both the goals of speed and logic conservation.

To meet the constraints discussed above, 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. To reduce the time for this design space exploration, we tackle the hardware configurations individually one by one.

A. Choice of NiosII type

The NiosII processor comes in three customizable types. 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 [10]. The choice of the right processor type is driven by the speed requirements of the concerned application module and the availability of sufficient FPGA resources.

B. I-cache & D-cache

A second criterion for selecting a particular type of the NiosII processor is the need for instruction and data cache. The Nios II architecture supports both instruction cache) and data cache. The need for higher memory performance and hence the need for cache memory, is application dependent. A Nios II processor core might include one, both, or neither of the cache memories. Furthermore, for cores that provide data and/or instruction cache, only the sizes of the cache memories are
TABLE I
MEMORY REQUIREMENTS OF THE APPLICATION MODULES

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Memory Foot Print</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kalman Filter</td>
<td></td>
</tr>
<tr>
<td>Whole Code + Initialized Data</td>
<td>81KB</td>
</tr>
<tr>
<td>.text Section alone</td>
<td>69.6KB</td>
</tr>
<tr>
<td>.data Section alone</td>
<td>10.44KB</td>
</tr>
<tr>
<td>stack Section alone approximately</td>
<td>2KB</td>
</tr>
<tr>
<td>heap section alone approximately</td>
<td>1KB</td>
</tr>
<tr>
<td>Gating Module</td>
<td></td>
</tr>
<tr>
<td>Whole Code + Initialized Data</td>
<td>63KB</td>
</tr>
<tr>
<td>.text Section alone</td>
<td>51.81KB</td>
</tr>
<tr>
<td>.data Section alone</td>
<td>8.61KB</td>
</tr>
<tr>
<td>stack Section alone</td>
<td>approximately 2KB</td>
</tr>
<tr>
<td>heap section alone approximately</td>
<td>1KB</td>
</tr>
<tr>
<td>Munkres Algorithm</td>
<td></td>
</tr>
<tr>
<td>Whole Code + Initialized Data</td>
<td>62KB</td>
</tr>
<tr>
<td>.text Section Alone</td>
<td>52.34KB</td>
</tr>
<tr>
<td>.data Section Alone</td>
<td>10.44KB</td>
</tr>
<tr>
<td>.stack Section Alone</td>
<td>approximately 2KB</td>
</tr>
<tr>
<td>.heap section alone</td>
<td>approximately 1KB</td>
</tr>
</tbody>
</table>

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

VI. 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.

![Kalman Filter performances for different Cache Sizes](image1)

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 profoundly when the I-cache size is increased to 16KB. Beyond 16KB I-cache, the execution time remains almost unchanged with increasing I-cache size. 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, as mentioned earlier, for tracking a maximum of 20 obstacles we need 20 of these processors. A NiosII/f having 16KB I-cache and 2KB D-cache, uses total on chip which accounts for 7% of the block memory available on our FPGA. Obviously replication of this system composition 20 times is not feasible.

![Kalman Filter performances with 4KB I-cache](image2)

We experimented with various combinations of I-cache and D-cache sizes to determine the optimum cache sizes for each module. In sections VI, VII and VIII we discuss the results of these experiments.

C. Floating point custom instructions

The floating-point custom instructions, optionally available on the NiosII processor, implement single precision floating-point arithmetic operations in hardware. They accelerate floating-point operations in Nios II C/C++ applications.

When the floating-point custom instructions are present in the target hardware, the Nios II compiler compiles the code to use the custom instructions for floating-point operations, including addition, subtraction, multiplication, division and the newlib math library.

However the floating-point custom instructions add substantially to the size of the hardware design. Hence floating point custom instructions must be used with caution to avoid unnecessary hardware generation. In sections VI, VII and VIII we discuss these issues.

D. 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). We can control the placement of .text, .data, heap and stack memory partitions by altering the NiosII system library or BSP settings. We can place any of these memory sections in the on-chip RAM if needed to achieve desired performance.

Table I summarizes the memory requirements of all the application modules. 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. However if a module uses malloc() or new() abundantly, placing the heap section in the on-chip memory can improve its speed by a large margin. Similarly if a module makes frequent calls to other functions, putting the stack section in the on-chip memory can help reach a higher execution speed for that module.
However with the optimal configuration (16KB I-cache and 2KB D-Cache) it is possible to reduce the number of processors for the Kalman filters and thus conserve precious FPGA logic. 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 application to deal with this housekeeping function. This would obviously add some latency into the system and we would exceed the 25ms time limit. Therefore we tested the floating point custom instructions’ impact on Kalman filter’s performance. We further investigated the prospects of improving the runtime for the filter through on chip memory placement. The outcome of this investigation is summarized in Figure 4. Moving just the stack section to the on-chip memory reduces that runtime 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 can 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 arrangement.

Since the housekeeping is not a computation intensive task, we propose a NiosII/e processor for it alongside the NiosII/f 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.

VII. GATING MODULE OPTIMIZATION

The gating module’s behavior with respect to the I-cache and D-cache sizes is shown in Figure 5. The overall processor run time is the smallest (70ms) when I-cache size is 16KB and D-cache size is 2KB. Therefore these are the right I-cache and D-cache sizes for the processor executing the Gating module. 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 the 25ms mark we are aiming for. We charted the performance of the processor while varying the number of obstacles from 2 to 20. Figure 6 shows the performance of the Gating module after the floating point custom instructions were added to the processor. The overall performance improves by approximately 50%. To improve the performance further, we experimented with placing various memory sections in the on-chip memory. Figure 6 highlights Gating module’s performance for different memory placement experiments.

VIII. MUNKRES ALGORITHM OPTIMIZATION

Munkres algorithm displayed the behavior as presented in Figure 7. 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 combination 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. We gain only 6ms if all the memory sections are placed on-chip. This is neither sufficient nor feasible given the memory footprint of the algorithm.

The solution finding process of the Munkres algorithm remains the same whether the elements of the cost matrix are floating point numbers or integer numbers. We found out that by truncating the fractional part of the floating point elements of the cost matrix, the final solution does not change. Hence we 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. Additionally, using an integer cost matrix obviates the need for the floating point custom instructions consequently 8% of the FPGA logic is conserved. A glimpse of the advantage of these optimizations can be seen in Figure 8 which shows the performance of the algorithm using this system composition for number of obstacles ranging from 2 to 20. The Call to Munkres signifies the total run time of the algorithm including the six sub functions. The run time for the overall algorithm drops down to 24ms as opposed to the 71ms with floating point cost matrix.

IX. TRACK MAINTENANCE

So far we have not mentioned the track maintenance block of the MTT application in the context of optimization. The reason for this deliberate omission is that a simple NiosII/e processor executes this block in 8ms. In future we may even get rid of this processor and run the track maintenance block as a second task on one of the other processors

X. CONCLUSION

After developing the application, we profiled it to identify performance bottlenecks and dependencies among the application modules. Using three different hardware implementations of the NiosII soft core embedded processor and other components, we devised a preliminary heterogeneous MPSoC architecture for the system. To avoid overusing the on-chip memory we optimized the I-cache and D-cache sizes for each application module. We incorporated the floating point custom instructions hardware in the relevant processors to accelerate them further. 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 25ms cut-off and hence we settled for it. The final output of the Munkres algorithm remains unchanged if we drop the fractional part of the floating point elements of the input Cost Matrix. This manipulation of the input matrix reduced the runtime for the algorithm to 24ms without compromising the accuracy of the final solution. We optimized the amount of processing resources to reduce the hardware size e.g. in the case of the Kalman filters. We finalized the optimum architecture as shown in Figure 9. The whole system fits in a single Stratix II 2S60 FPGA and it meets the time constraints of the application.

REFERENCES