Computer Vision and Image Understanding 114 (2010) 1203–1214

Contents lists available at ScienceDirect

Computer Vision and Image Understanding

journal homepage: www.elsevier.com/locate/cviu

Design and implementation of embedded computer vision systems based on particle filters

Sankalita Saha [a,*], Neal K. Bambha [b], Shuvra S. Bhattacharyya [c]

[a] Mission Critical Technologies Inc./NASA Ames Research Center, Moffett Field, CA 94035, United States
[b] U.S. Army Research Laboratory, Adelphi, MD 20783, United States
[c] Department of Electrical and Computer Engineering, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, United States

ARTICLE INFO

Article history:
Received 27 January 2009
Accepted 17 March 2010
Available online 29 April 2010

Keywords:
Design space exploration
Particle filters
Reconfigurable platforms

ABSTRACT

Particle filtering methods are gradually attaining significant importance in a variety of embedded computer vision applications. For example, in smart camera systems, object tracking is a very important application and particle filter based tracking algorithms have shown promising results with robust tracking performance. However, most particle filters involve vast amount of computational complexity, thereby intensifying the challenges faced in their real-time, embedded implementation. Many of these applications share common characteristics, and the same system design can be reused by identifying and varying key system parameters and varying them appropriately. In this paper, we present a System-on-Chip (SoC) architecture involving both hardware and software components for a class of particle filters. The framework uses parameterization to enable fast and efficient reuse of the architecture with minimal re-design effort for a wide range of particle filtering applications as well as implementation platforms.

Using this framework, we explore different design options for implementing three different particle filtering applications on field-programmable gate arrays (FPGAs). The first two applications involve particle filters with one-dimensional state transition models, and are used to demonstrate the key features of the framework. The main focus of this paper is on design methodology for hardware/software implementation of multi-dimensional particle filter application and we explore this in the third application which is a 3D facial pose tracking system for videos. In this multi-dimensional particle filtering application, we extend our proposed architecture with models for hardware/software co-design so that limited hardware resources can be utilized most effectively. Our experiments demonstrate that the framework is easy and intuitive to use, while providing for efficient design and implementation. We present different memory management schemes along with results on trade-offs between area (FPGA resource requirement) and execution speed.

1. Introduction

Particle filtering is an emerging and powerful methodology for computer vision applications especially in tracking based systems. Particle filters are based on the idea of approximating the probability density functions (PDFs) of the state of a dynamic model by random samples (particles) with associated weights and propagating them across iterations based on the probabilistic model of the state update and the measurements. But use of particle filters in real-time systems has been limited due to their computational complexity. A particle filter typically involves several complex mathematical operations that are invoked at every iteration of the filter, as well as a large number of particles, which in turn results in huge memory requirements. A possible solution for real-time implementation of such systems is parallelization and the use of multiprocessor systems; but this is also restricted because of the presence of an unavoidable computing step (resampling), which is serial in nature, and therefore difficult to parallelize. Nonetheless, parallel software implementation of particle filter based tracking applications have been explored such as the one demonstrated in [14] where a high-speed multiprocessor cluster comprising of 24 SUN UltraSparcIII machines running at 750 MHz was used as the target platform. Such tracking applications are of great significance to various embedded systems such as smart cameras that do not feature such powerful computing platforms. For example, the smart camera series 17xxx from National Instruments provides only a programmable digital signal processor (PDSP) chip along with a 533 MHz PowerPC. Such cameras have already started

* Corresponding author.
E-mail addresses: sankalita.saha@nasa.gov (S. Saha), nbambha@arl.army.mil (N.K. Bambha), ssb@umd.edu (S.S. Bhattacharyya).

1077-3142/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.cviu.2010.03.018

providing capabilities for relatively low-weight computer vision tasks such as edge detection, pattern matching and so on. However, incorporating more complex operations such as tracking remains a challenging task.

This suggests the need for exploration of customized solutions for new embedded platforms comprising of multiple components besides the main CPU such as embedded memory, memory interfaces, and specialized I/O interfaces, along with domain-specific IP cores. These new emerging class of architectures comprising of heterogeneous System-on-Chips (SoCs) are capable of providing advanced support for embedded computer vision applications. Examples of such heterogeneous processing platforms are platform field-programmable gate arrays (FPGAs). As more and more complex computer vision applications are being ported to such heterogeneous embedded platforms, the need for efficient implementation methodologies for such systems is increasing since the increased functionality of such architectures leads to increase in design and implementation complexity.

Design and implementation of a generic yet highly optimized architecture for all particle filter based computer vision systems is not possible because of the wide range of applications to which particle filtering techniques are applied currently and may be applied in the future. But, there are many tracking applications that share similarities with regards to the particle filtering framework. A generic architectural framework that can be suitably and easily reconfigured for such applications would be of significant utility. Such an architecture could be highly optimized as well because of the potential for streamlining based on a given set of particle filtering features.

In this paper, an SoC architecture involving parallel processing units for tracking applications using particle filters is proposed. The architecture utilizes specialized hardware elements as well as special soft cores. Additionally, a novel parameterized design framework to implement particle-filter-based applications on platform FPGAs is proposed. The main aim of this framework is to enable comprehensive design space exploration of complete particle filtering systems that can be used across different applications. Exploration of the hardware/software co-design and implementation, and analyzing trade-offs associated with partitioning and mapping with respect to experiments with three different applications are presented as well.

2. Related work

Real-time implementation of particle filter based computer vision applications presents a two-dimensional problem of efficient memory management, as well as high-speed processing. Various modifications to the particle filtering technique itself have been proposed to meet the above requirements [11]. Since most of the targeted embedded applications involve extensive computation, parallelization and hence multiprocessor implementation of particle filters is an important option to examine.

A significant body of work exists on optimizing generic particle filter systems with special focus on the non-parallelizable resampling step (e.g., see [5,16,17]). For example, a generic system architecture for particle filters has been proposed by Bashi et al. [3]. However, this work mainly concentrates at the algorithmic level. Architectural design and efficient memory management schemes for particle filter implementations are discussed in [2,4,16]. A low-power analog particle filter implementation has been described in [18]. Mixed mode implementations — that is, partially-analog and partially-digital realizations — have also been explored. In such mixed-mode approaches, the analog components are used for the non-linear computations that are involved in particle filtering [19]. In design efforts towards computer vision applications, in [14] a shared memory multiprocessor implementation of a particle-filter-based 3D facial pose tracking algorithm was developed which was shown to provide significant performance gains. However, the implementation domain was not embedded platforms.

Use of reconfiguration capabilities for enhanced design space explorations and robust implementations have been explored in limited scope. In [10], the authors provide a scheme for reconfigurable particle filtering, where two particle filtering algorithms are implemented on the same platform, and the system can be configured to use any one of them by switching mechanisms. Another relevant method of reconfiguration and dynamic design using parameterization was proposed in [9]. This method was developed for shape-adaptive template matching.

As can be observed from above, there is a lack of focus on specialized efforts for optimized embedded implementation solutions for particle filter applications in the computer vision domain with even less attention devoted to efficient design space exploration through exploitation of reconfigurability and interactions among the various processing sub-systems. The main objective of this paper is to help bridge this gap, and provide a systematic method to facilitate a comprehensive coupling between particle filter applications and their embedded implementations. An initial introduction to this framework was presented in our earlier work which focused on one-dimensional particle filter systems on pure hardware platforms [15].

However, particle filters with multi-dimensional state space are in widespread use in tracking problems in computer vision. In particular, tracking problems that involve human pose require very high-dimensional state models. For example in [20], the authors use particle filters for multi-camera 3D person tracking which involved a 6-dimensional state space. In [8], the authors explored use of prior knowledge in a particle filtering framework for 3D tracking where the state space was 12-dimensional while the authors in [12] used a 14-dimensional state space for the particle filter based 2D articulate pose tracking. In this paper, we focus on the challenges in a multi-dimensional particle filter. In addition, we extend the framework for hardware/software heterogeneous platforms. We also explore memory optimization issues and their different solutions which are of critical importance to multi-dimensional particle-filter-based applications that have significant memory requirements.

3. System design framework

Particle filters provide a method for recursively estimating the unknown state, from a collection of noisy observations. The state parameters to be estimated are dependent on the exact problem being considered. The state transition and observation models are given by

$$\text{State Transition Model } X_t = F(X_{t-1}, W_t) \tag{1}$$

and

$$\text{Obsevation Model } Y_t = G(X_t, V_t) \tag{2}$$

where W_t is the system noise and V_t is the observation noise. X_t represents the dynamically evolving state of the system, and Y_t is the observation vector of the system, which is corrupted by the measurement noise at instant t. The particle filter estimates the state of the system and updates it based on the received, corrupted observations.

As shown in any Fig. 1 particle filter based system essentially consists of the following three computational steps:

- *Sampling*: in this step samples (particles) of the unknown state are generated based on the given sampling function. These samples provide an estimate of the current state of the system and also propagate the particles from previous time instant to current.
- *Weight calculation*: based on the observations an importance weight is assigned to each particle.

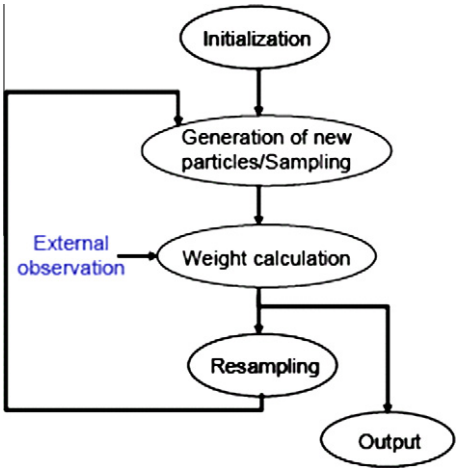

Fig. 1. Particle filtering algorithm.

- *Resampling*: this step involves the act of redrawing particles from the same probability density based on some function of the particle weights such that the weight of each new particle is approximately equal. Resampling is a very important step in a particle filter and without this step a particle filter is highly likely to degenerate, i.e., after a few iterations all the weights will go to zero except the weight of one particle.

While the sampling and the weight calculation steps are strongly dependent on the application, various standard methods of resampling exist and may be chosen based on system constraints such as accuracy, error tolerance etc. Also, sampling and weight calculation are generally the most computationally intensive and involve complex computations such as transcendental, trigonometric and exponential functions.

3.1. Overview

The architecture proposed in this paper is based on the computational framework described above. There exist wide ranges of computer vision applications that use the same particle filtering algorithm with different state models; e.g., tracking of human face can be done with same motion tracking platform with varying models for the face. Thus, it is possible to develop a generic architecture for a subset of applications and streamline it for specific applications within the subset. The goal of this framework is to provide the user a systematic approach for such streamlining — with the ability to explore the various design trade-offs between area and execution speed — and provide the capability to implement a wide range of applications with significantly reduced re-design effort.

To achieve this, first, a system architecture is devised that is based on the use of parallel processing elements to achieve as much performance improvement using parallelization as possible. A comprehensive design framework is required for efficiently mapping the applications to this architecture and then finally onto the implementation platform. For this, a parameterized design framework is proposed; the fundamental idea being dividing the overall system into small parameterized sub-systems. Each such subsystem can then be modified to the needs of a wide range of applications, as well as to final target constraints by setting appropriate parameters, such as the memory size, and the number of particles.

An overview of a two-processing-element configuration of such an architecture is given in Fig. 2. The framework essentially consists of an array of processing elements (PEs), and a resampling unit, along with a set of parameterized interfaces. A PE consists of three units, a PEcore, a weight calculation unit (WU), and a noise generator as shown in Fig. 3. Each of these units can operate independently of changes in functionality of the other units. However, the interaction between various units can change with the variation in the functionality of any one unit. These changes are handled by the interfaces so that the individual streamlined units need not be redesigned, which would require significant effort.

The PEcores perform the sampling operation, while a separate weight calculation unit (WU) is used for calculating the weights. The PEcore as well as the WU interact with memory banks whose sizes are dependent on system parameters. The interfaces provide parameterized interaction with memory banks and the resampling interface (where required), and perform synchronization operations. The individual units can be composed as specialized hardware modules or as software modules that are to be executed on embedded processors in the target platform.

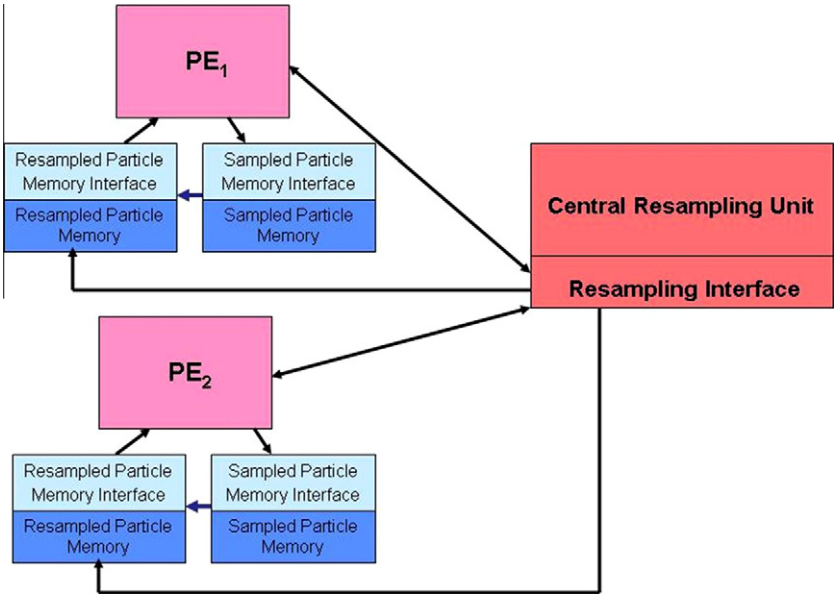

Fig. 2. Distributed particle filter architecture with *memory scheme A*.

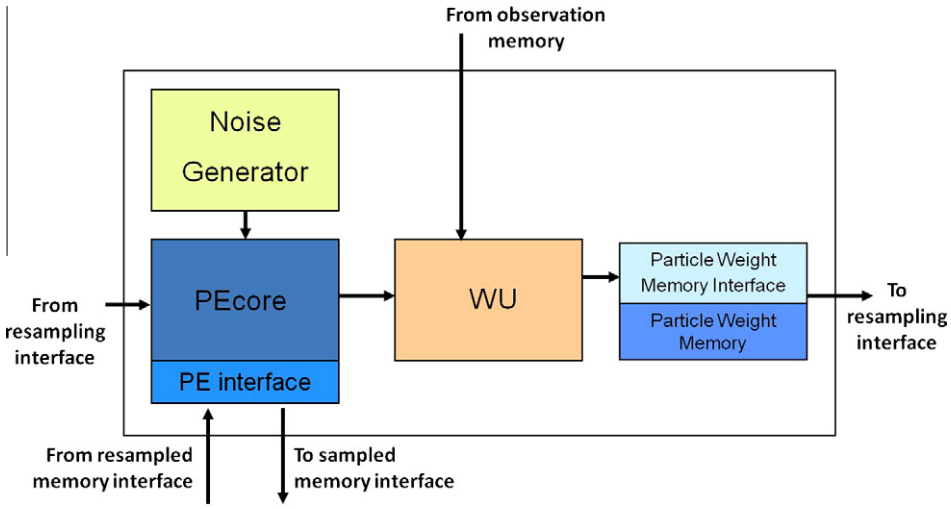

Fig. 3. Single PE architecture.

3.2. Design framework

In this section we present the details of the design framework and the parameterizations that can be employed based on restrictions imposed by the available implementation resources. Fig. 4 shows the overall design framework for a heterogeneous implementation platform comprising of both hardware and software components. We use Xilinx's System Generator and the Xilinx EDK for design and functional verification of the hardware components and processor (for software modules), and the Xilinx ISE tool-set for synthesis of the hardware modules. Xilinx System Generator provides a hardware library that consists of various architectural units, such as RAMs and adders, for modular design. It allows the use of custom Verilog or VHDL modules for system design. In order to incorporate software implementation of a part of the system, the soft core modules such as the MicroBlaze or PowerPC modules need to be configured appropriately. The use of such soft core modules increases the complexity of the interfaces between heterogeneous components as compared to hardware only implementation as explored in [15].

As mentioned in section A, multiple processing elements (PEs) for the sampling and weight calculation step are used. Within a given PE, further pipelining can generally be used, but the degree to which pipelining can be employed is strongly dependent on the characteristics of the targeted application. The sampling and weight calculation operations involve complex mathematical operations, and thus impose restrictions on the number of PEs that can be implemented. The number of particles handled by each PE is

$$p = \lceil P/N \rceil, \tag{3}$$

where $\lceil x \rceil$ denotes the smallest integer that is greater than or equal to the real number x; P is the number of particles; and N is the number of PEs. Note that for multi-dimensional particle filters, each particle represents a vector with multiple values; thus for a state vector of dimension m, a single particle is a vector with m values. In general, the sampling step can be implemented in hardware. However, since the weight calculation or "weight update" (WU) step for some applications may involve many complex mathematical functions, it may not always be possible to accommodate multiple WU units. In such cases, the most complex part is moved to the resampling unit. The WU unit computes an intermediate result that is then sent to the central resampling unit, which computes the final value before resampling. Since the resampling unit is serialized, it accommodates only one unit for computing the complex operations.

The following straightforward memory management scheme for particle storage and updating may be used for most applications. Three memory banks or buffers are used for each PE for storing (1) sampled particles, (2) particle weights, and (3) resampled particles. Since the number of memory banks that are available on a given platform is limited, we have

$$N \leqslant (M/3), \tag{4}$$

where M is the number of memory banks available on the targeted FPGA board. However, for a multi-dimensional particle filter with m dimensions, the memory requirement becomes

$$N \leqslant (M/(3 \times m)). \tag{5}$$

For memory-intensive applications commonly encountered in computer vision systems, more optimized memory management schemes are required and such schemes can often depend on the specific application. A more efficient memory management strategy is discussed later in this paper in the context of the 3D facial pose tracking application.

The area consumed by the associated memory banks directly depends on P, N and m. The observation data is stored in a shared memory between clusters of PEs. The memory interface for this buffer handles the read requests from the PEs. The reading from this memory for the ith operation can be overlapped with either the resampling step of the $(i-1)$th operation, the sampling step of the ith operation, or both. However, if the system throughput is greater than or equal to the observation input rate, this interface becomes trivial as only a single buffer is required. Note that in the case the WU unit is partially integrated with the resampling unit, the particle weight memory stores the intermediate weight.

There are seven main interfaces corresponding to the operations of: (1) observation data reading, (2) sampled particle memory interfacing, (3) resampled particle memory interfacing, (4) particle weight memory interfacing and (5) resampling unit interfacing. Among these, the reading of observation data is not dependent on m, N or P, while the rest are dependent on N, P and m. The resampling unit varies based on the resampling scheme being used and is functionally independent from the rest of the units. It is triggered when all the P particles have been processed for a given iteration.

The resampling interface consists of a global address generator and a local address generator. The global address generator generates addresses for P particles and depends on P. These addresses are routed to individual PEs by the local address generator, which,

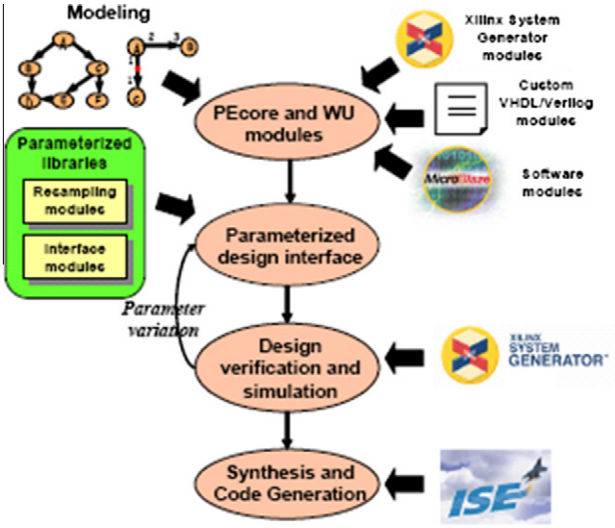

Fig. 4. Parameterized design framework.

thus, depends on both P and N. In this framework, systematic resampling has been used. However, this can be easily replaced with other sequential resampling mechanisms. Systematic resampling is often a preferred method due to its computational simplicity and good empirical performance [7]. When the PEs carry out partial weight calculation, the remaining weight calculation is carried out in the resampling unit and hence the corresponding module is integrated. A library of these parameterized interfaces and resampling schemes are created using a combination of Xilinx System Generator hardware components and custom modules.

The execution time for resampling directly depends on P and is constant over all iterations. Thus, the total execution time (in terms of clock cycles) for one iteration is:

$$T = T_{\text{PEcore}} + L_{\text{resampling}} + L_{\text{WU}}, \tag{6}$$

where $L_{\text{resampling}}$ is the latency due to the resampling unit, L_{WU} is the latency induced by the WU unit, which increases as m increases, since the number of weight update increases for a multi-dimensional particle filter. The exact nature of how this increase in latency varies based on the complexity of the step and hence is application dependent. T_{PEcore} is the execution time of PEcore, which for a fully pipelined PEcore is given as

$$T_{\text{PEcore}} = L_{\text{PEcore}} + \lceil P/N \rceil. \tag{7}$$

Note that for a multi-dimensional particle filter P depends on the number of dimensions i.e., m, and hence T_{PEcore} increases as m increases. The same holds for the latency of the resampling step which for systematic resampling is given by Athalye et al. [2]:

$$L_{\text{resampling}} = 2 \times P - 1. \tag{8}$$

This signifies that the latency of the resampling unit increases directly with an increase in number of particles, and thus the latency will generally become a bottleneck for applications requiring very high P and/or for multi-dimensional particle filters with high m. When partial weight calculation is performed in the PE, Eq. (6) is modified to

$$T = T_{\text{PEcore}} + T_{\text{resampling}} + L_{\text{WU1}}, \tag{9}$$

where L_{WU1} refers to the latency of the WU unit which partially computes the weight of the particles. The total resampling time is now given by

$$T_{\text{resampling}} = L_{\text{resampling}} + L_{\text{WU2}} + L_{\text{interface}}, \tag{10}$$

where L_{WU2} refers to the latency due to weight calculation performed in the resampling unit and is non-zero only when partial weight calculation is done in the PEs. L_{WU2} depends on the complexity of the computation. For software implementation of the resampling module — as explored in one of our implementations — $L_{\text{interface}}$ provides the latency due to interfacing between hardware modules and EDK processor and depends on m, N and P. For the first processing iteration, any initial latency that exists should be added to the latency model of Eq. (7). Such initial latency may exist, for example, because of startup time associated with the noise generator. Note that this execution time analysis is intended to aid the system architect in making design choices in order to create an efficient implementation. In our experiments, this analysis was used to aid the design process as well as reduce design time.

4. Experiments and results

In this section, implementations for three different particle filter problems using our proposed architectural framework are demonstrated along with corresponding experimental results. First, the basic framework is illustrated by means of two generic particle filter systems using underlying one-dimensional models. The results of the implementations show that the block RAMs (BRAMs) were not fully utilized for these designs, which indicates that systems with multi-dimensional models can be supported as well. The next application explored is based on such a multi-dimensional model. This application is a 3D facial pose tracking system in video. Based on our proposed design methodology, details of this application are provided, along with partitioning and mapping results.

The three systems were designed and synthesized using Xilinx System Generator 8.2, Xilinx EDK 8.2, and Xilinx ISE 9.1. For the first two applications, the target device family was the Xilinx Virtex-4SX series. Although the FPGA board used in the experiments could support a clock frequency of 500 MHz, this frequency could not be attained in most cases. For the third application, the Video Starter Kit (ML 402) from Xilinx was utilized that provides advanced support for video and imaging applications. By varying key parameters appropriately, different implementations were obtained and various design options were explored. We elaborate on this exploration in the remainder of this section.

4.1. Uni-variate non-stationary growth model

The first application explored is an example of a one-dimensional non-linear system (typically studied in the context of stochastic systems) [5]. The state transition and observation models are as follows

$$X_t = 0.5 \times X_{t-1} + \frac{25 \times X_{t-1}}{1 + X_{t-1}^2} + 8 \times \cos((1.2 \times (t-1)) + W_t), \tag{11}$$

and

$$Y_t = X_t^2/20 + V_t, \tag{12}$$

where W_t and V_t, are zero-mean Gaussian white noise with variances 10 and 1, respectively. The execution of the PEs and the resampling units is fully pipelined. The above equations were mapped to appropriate Xilinx System Generator computation blocks to build the PEcore and the WU. The noise generation was performed using Xilinx's Gaussian white noise generator. This noise generator needs only periodic resetting to provide continuous output, thus the PE interface does not have to send requests for data. However, the initial latency of the generator is 10 cycles, which is present only for the first iteration. Additionally, Xilinx's lookup-table-based cosine generators were used. These are both fast and inexpensive (area-efficient) compared to standard CORDIC cosine

generators. We employed fully-pipelined multipliers and dividers. In the design, the WU uses an exponential calculation unit that uses a combination of a lookup table and a polynomial approximation method. Uniform random number generation for resampling is done using multiple-bit, leap-forward linear feedback shift registers (LFSRs) [6]. Parameterized interfaces were used to build the interconnections between the various sub-systems.

4.2. Uni-dimensional failure prognosis model

This practical particle filtering application is adapted from [13], where particle filtering is used to track crack faults in the blades of a turbine engine. The state transition and observation models for the fault growth system are given by

$$X_t = X_{t-1} + \frac{1}{X_{t-1}^5 + X_{t-1}^4 + X_{t-1}^3 + X_{t-1}^2 + 1} + W_t, \tag{13}$$

and

$$Y_t = X_t + V_t, \tag{14}$$

where W_t and V_t, are zero-mean Gaussian white noise with variances 10 and 1, respectively. The PEcore is comprised of multipliers and a divider, all of which are fully pipelined cores from Xilinx. The Gaussian white noise generator, exponential calculation unit, and uniform random number generator used in Section A. are reused again. The resampling unit and interfaces were selected from the library and design space exploration is done by varying P and N.

4.3. 3D facial pose tracking in video

For this system, the computational complexity of the application made implementation of the whole system on hardware unfeasible. Hence, an embedded processor to implement software modules was used as well. Thus, partitioning and mapping decisions were required to appropriately identify the units of the system to be moved into hardware and software modules on the platform. This was done based on profiling of a MATLAB-based software prototype.

4.4. Application overview

The aim in facial pose tracking is to recover the 3D configuration of a face in each frame of a video. The 3D tracking algorithm considered in this work uses the particle filtering technique along with geometric modeling [1]. There are three main aspects that capture the 3D tracking system. The first is the model to represent the facial structure. The second is the feature vector used. The third is the tracking framework used.

A model attempts to approximate the shape of the object to be tracked in the video. In our application, a 3-dimensional cylinder

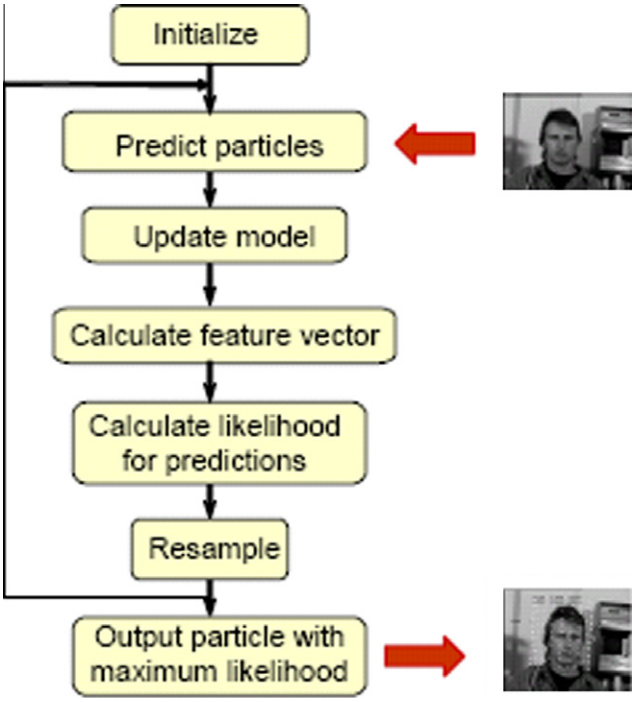

Fig. 6. Flow of 3D facial pose tracking algorithm.

with an elliptical cross-section is chosen as a model to represent the structure of face. Thus, in order to track the facial pose over the video, the position and orientation of this cylinder is tracked. Therefore, the state vector is comprised of 3 translation parameters and 3 orientation parameters which is essentially the state vector for the particle filter tracking framework.

The feature vector represents characteristics from the image that can be used to update the particle filter. Thus, the features should be easy to detect yet robust to occlusions, changes in pose, expression and illumination. A hybrid approach is used for the feature set which combines the advantages of a purely geometric approach and the power of statistical inference. A rectangular grid superimposed around the curved surface of the elliptical cylinder and the mean intensity for each of the visible grids/cells forms the feature vector. Given the current configuration, the grids can be projected onto the image frame and the mean can be computed for each of them; a perspective projection of the cylinder and the grids is used. For further details please refer to [14]. Fig. 5 taken from an illustration in [14] shows the model along with the rectangular grid.

For the tracking framework — i.e., for estimating the configuration or pose of the moving face in each frame of a given video — a

Fig. 5. An example of 3D facial pose tracking with a cylindrical mesh.

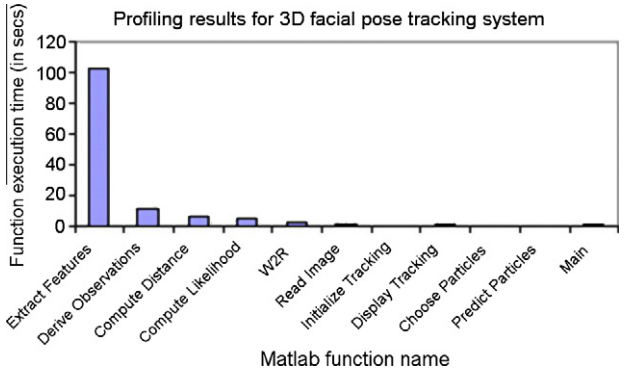

Fig. 7. MATLAB profiler result for the 3D facial pose tracking system.

particle-filter-based technique is used. Here, each particle has 6 dimensions comprised of 3 translation and 3 rotation parameters. For each new image frame read in from the camera, multiple predictions for these configuration parameters are the particles. The model is updated based on the particles, i.e., the 3 position and 3 rotation parameters are updated. This is followed by extraction of the feature vector for each new position of the cylinder as represented by the particle value and the weight of each particle is calculated. The particle that yields the best likelihood value gives the position of the face in the frame. The number of particles to be used in the system is decided by the user and is constant for one application. The complete algorithmic flow is given in Fig. 6.

4.5. Partitioning and mapping

The initial algorithm was developed in MATLAB and hence the MATLAB profiler was used to derive a distribution of execution times across the various functional sub-systems. The MATLAB profiler provides information about individual function execution times along with the total execution times and the number of calls made to each function. The profiling results for individual execution times are shown in Fig. 7. In this figure, the execution time for a function is the total time spent in the function for a single execution of the overall program.

As we can observe from the figure, the "extract features" function, which extracts the feature vector, contributes the most to the overall execution time. This function is a part of the weight calculation step of the overall particle-filter. Thus, it is necessary to speed up this unit as much as possible. Computationally, this function consists of computing the mean pixel value of the grids/cells. This can be done in parallel since the computation for one grid/cell does not depend on the rest. However, if multiple units for this function are created to exploit this parallelism, the number of units is restricted by the number of RAMs present in the target platform, since each of these parallel units requires a copy of the image. Shared memory can be used, but the number of read ports for such memories in FPGAs is limited, and hence, again multiple copies of the image would be required for maximal parallel data access. In our implementations, dual port RAMs have been used to enable limited sharing. Thus a single image is simultaneously shared by two units for the "extract features" function. The size of the image used in the application were considerable and the available RAMs were not sufficient to hold more than one copy of the image; thus only a single copy of the image was used which was shared by two units via dual ports.

For this application, the WU unit is split up into hardware and software sub-units to enhance performance within given resource constraints. The WU unit takes data i.e., external observations which is the current image frame and extracts features from the images as described earlier to compute the likelihood values. Thus, the weight calculation comprises of calculating the feature vector followed by computation of the likelihood values for the particles. The "likelihood calculation" is moved to software since it involves complex math functions. Moving this unit to software, saves resources which can then be exploited for the parallel hardware implementation of the "extract features" function. The resampling function was implemented in software. This was because no signif-

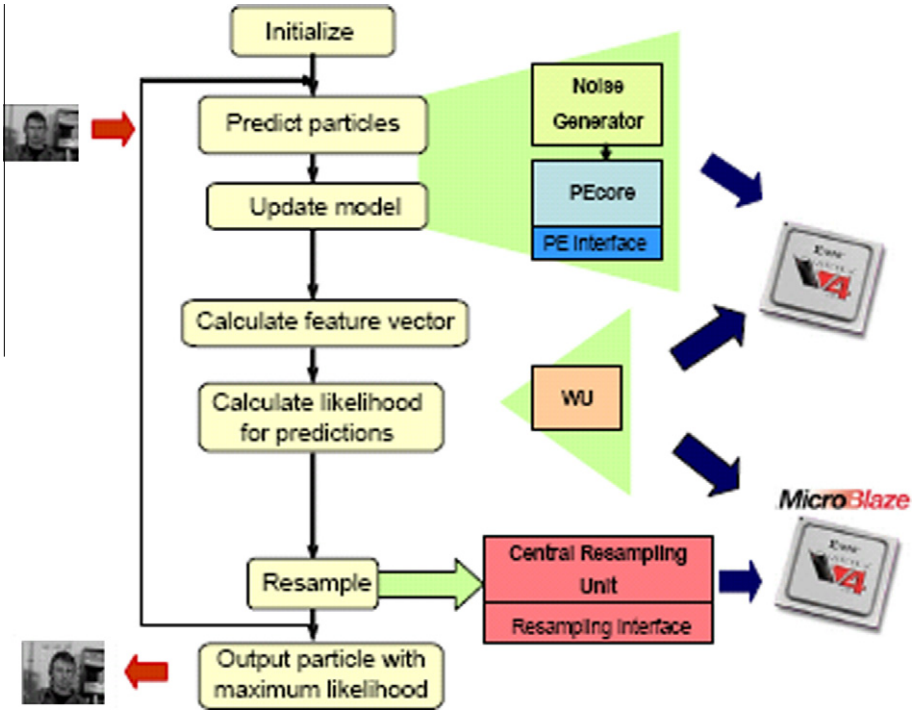

Fig. 8. Mapping of sub-systems of 3D facial pose tracking system onto hardware and software modules.

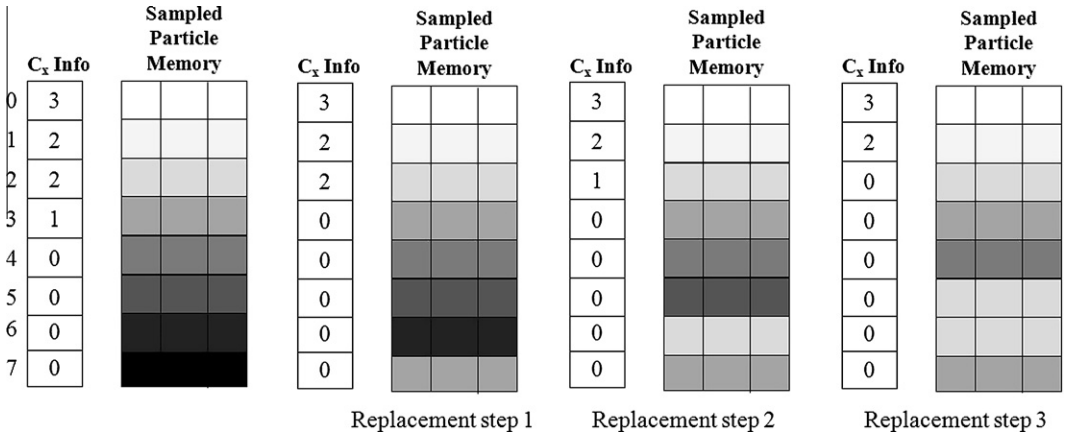

Fig. 9. First 3 iterations of systematic update of sampled memory with resampled particle value for $N = 8$, and $N_D = 3$.

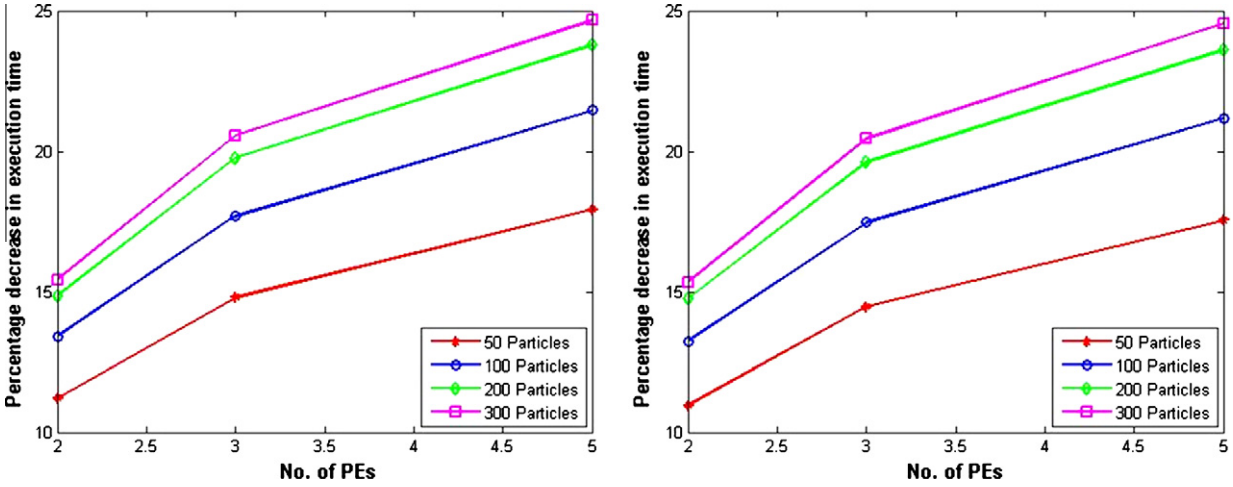

Fig. 10. Percentage decrease in execution time (1 iteration) for uni-variate non-stationary growth model and uni-dimensional failure prognosis model implementations.

Table 1
Execution time in μsecs (per frame) variation for the two 1-dimensional particle filter applications.

No. of PEs	$P = 50$	$P = 100$	$P = 150$	$P = 200$
Uni-variate non-stationary growth model implementation				
1	223	371	674	975
2	198	323	573	823
3	190	307	540	773
5	183	293	513	733
Uni-dimensional failure prognosis model implementation				
1	228	381	680	981
2	203	328	578	828
3	195	312	545	778
5	188	298	518	738

Table 2
FPGA resource utilization for uni-variate non-stationary growth model and uni-dimensional failure prognosis model implementations.

No. of PEs	Slices (%)	Slice flip-flops (%)	4 Input LUTs (%)	DSP48s (%)	BRAMs (%)
Uni-dimensional failure prognosis model implementation					
2	29.17	8.54	19.91	43.23	15.63
3	42.25	12.56	28.6	60.93	23.44
5	68.97	20.61	45.73	96.35	39.06
Uni-dimensional failure prognosis model implementation					
2	19.81	6.13	14.27	43.23	15.63
3	28.39	8.94	20.18	60.93	23.44
5	45.57	14.57	31.98	96.35	39.06

icant performance gain would be obtained with its hardware implementation. Further, hardware implementation of the resampling function would result in overheads in terms of interfacing with other units besides using up resources that can be utilized for more critical units.

The high-level task partitioning is illustrated in Fig. 8. In this implementation we ignore rotation effects due to resource constraints; thus our particle filter is 3-dimensional comprised of 3 translation parameters. Taking such effects into account provides more tracking accuracy, but requires complex trigonometric and matrix manipulation functions. Given the targeted system architecture, which involves multiple PEs parallelized over the set of particles, including rotation effects would translate to providing multiple instantiations of the required mathematical manipulation units (for the trigonometric and matrix manipulations). These multiple instantiations can be supported logically as an extension of our system architecture, but they would exceed the resources available on the targeted FPGA device. For future FPGA device families that provide more resources, incorporating rotation effects into our design framework is a useful direction for further work.

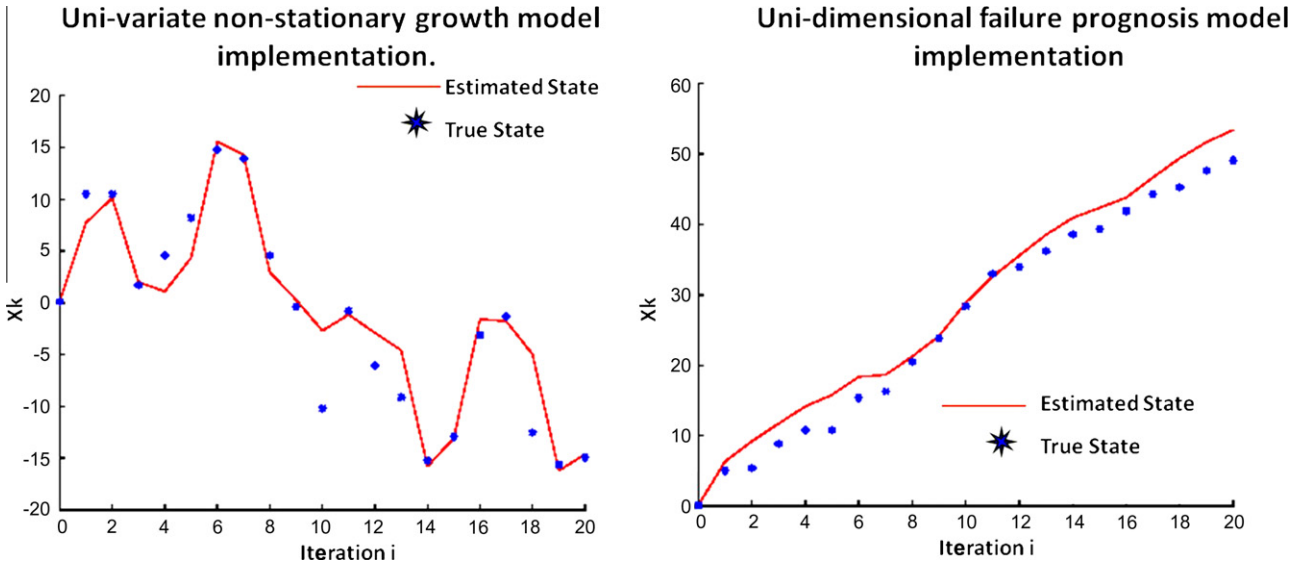

Fig. 11. Tracking results for the two 1-dimensional particle filter application implementations.

Table 3
FPGA resource utilization for 3D facial pose tracking system implementation (memory scheme A).

No. Of PES	Slices	Slice flip-flops	4 Input LUTs	DSP48s	BRAMs
2	87.57%	42.26%	41.03%	66.67%	96.35%

Table 4
Execution time (per frame) variation with total particles for a 2PE system (memory scheme A).

No. of particles (N)	Execution time per frame (in ms)
50	36.855
100	73.72
200	143.94
300	215.33

4.6. Memory management schemes

Though a straightforward memory management scheme (*memory scheme A*) is proposed in the main system design, for computer vision applications which almost always involve huge memory requirements, exploring different memory management schemes is a necessary requirement for optimized implementation. In this section, we focus on alternatives to the memory management advocated in Section III B.

With the straightforward scheme of using one memory bank for sampled particles and another bank for resampled particles, the memory requirement for holding particle information for a system with dimension N_D and utilizing N particles is $2 \times N_D \times N$. However, the use of two memory banks may be avoided at the expense of minor computational overhead. In order to reduce this memory requirement, let us first analyze how the sampled particle and resampled particle memory banks are accessed during execution. During resampling for iteration t_i, a few particles from the sampled particle set are replicated and the rest are discarded to form the new particle set for the next iteration t_{i-1}. While, in the subsequent iteration i.e., iteration t_{i+1}, the sampling operation involves accessing the resampled particle set and applying the sampling function to it. Once this operation is completed the resampled particle set is no longer required for the current iteration. Since the sampled particles and resampled particles are not used simultaneously except during sampling, a single memory bank of size $N_D \times N$ is needed to hold both the sampled as well as resampled particles with an additional memory bank to hold indices of particles while resampling. In such a scheme, during resampling, instead of copying the entire information of the particle being replicated from the sampled memory to the resampled memory, only the indices of the particles $(x_i, i = 1...N)$ to be replicated are stored in a memory bank of size N.

Fig. 12. Sample tracking results. The superimposed cylinder moves and tracks the face in each frame.

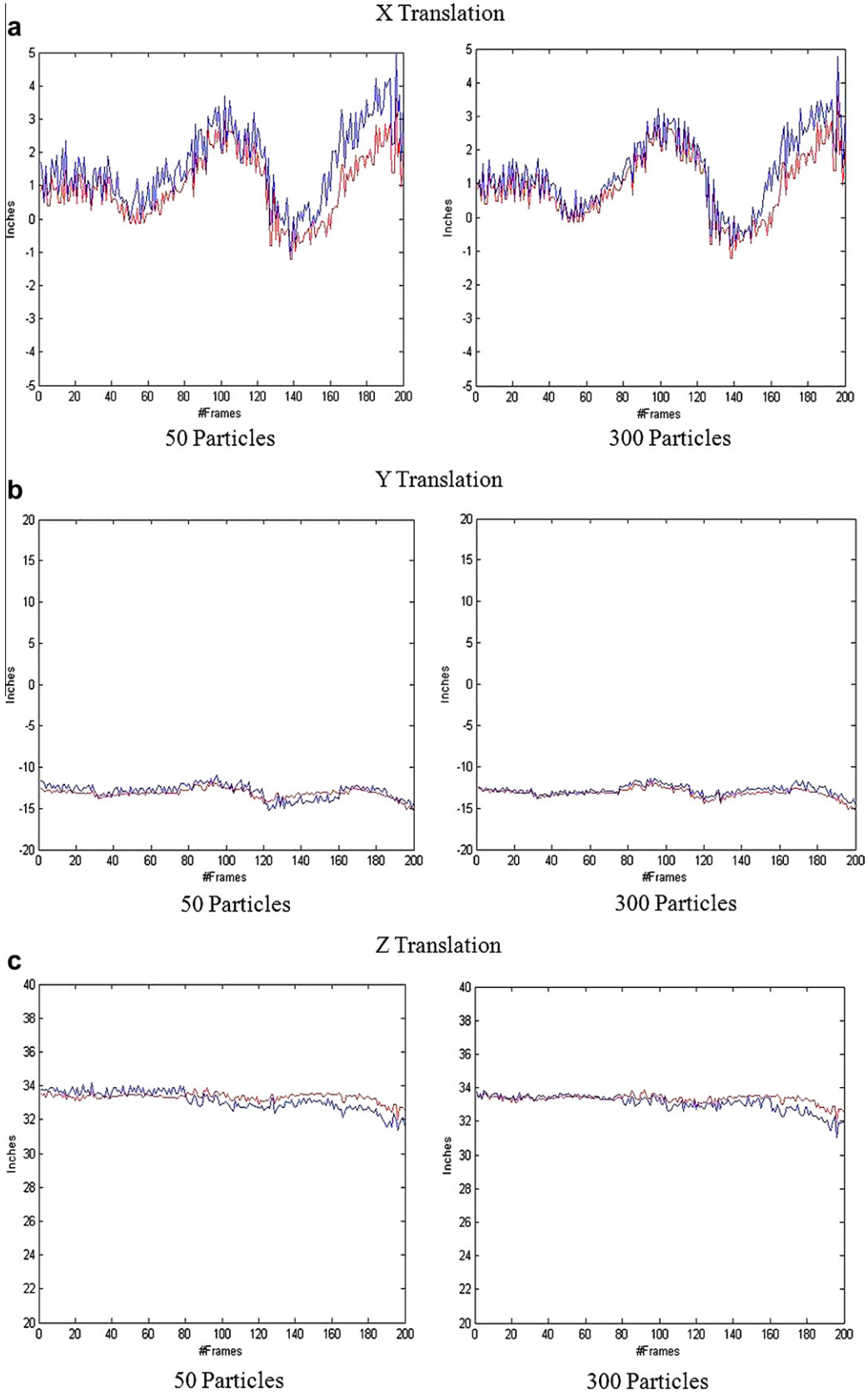

Fig. 13. Tracking results for 50 and 100 particles for the three translation parameters. The red line shows the actual values while the blue line shows the tracking values. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

In addition, the number of copies of the particle to be replicated (C_{x_i}) can also be stored. Once this information is stored, the sampled memory space can be systematically updated using x_i and C_{x_i} information to form the resampled particle memory bank. If

the information of C_{x_i} is not stored before, it will have to be computed during this update. Avoiding the storage of C_{x_i} information saves additional memory usage at the expense of extra computation. We call this management scheme *memory scheme B*.

Note that the above memory management scheme may not be efficient for all applications and is completely dependent on the resampling scheme. For some resampling scheme the overhead of the systematic updating of the sampled particle memory with resampled particle information may be prohibitively high. For our application, the resampling function is based on the weight of the particles also known as sample importance resampling. In this scheme, the number of copies of a particle to be replicated (p) is proportional to the weight of the particle. Thus, a particle with higher weight will be replicated more number of times compared to a particle with lesser weight which may not get replicated at all.

Such a resampling scheme lends well to the second memory management scheme discussed above i.e., *memory scheme B* which was used in our implementation with further optimization. During resampling, the sampled particle memory is sorted based on decreasing order of weight. Thus, in this case there is no need to collect x_i information and only C_{x_i} information is stored. Since the sampled memory is sorted in decreasing order of weight, x_i is for particle i its index in the sampled memory. To begin the replacement, the particle with the minimum C_{x_i} and highest index in the memory set, say $x_{C\min}$, is detected in this sorted particle set. The replacement of sampled particles with resampled particle values can then proceed by replacing the last sampled particle with $x_{C\min}$ and proceeding upward. Fig. 9 illustrates the first three steps of this procedure for $N = 8$ and $N_D = 3$. This memory management scheme (*memory scheme B*) reduces the particle memory requirement from $2 \times N_D \times N$ to $(1 + N_D) \times N$.

4.7. Implementation

The system architecture that we employed for this application is the same as that shown in Fig. 2. As mentioned earlier, the video starter kit (ML 402) was used to implement this application. The FPGA family supported by this board is the Xilinx Virtex-4 SX. The board also includes a video input/output daughter card and a CMOS image sensor camera. Xilinx's EDK version 8.2 along with Xilinx System Generator version 8.2 was used to create the MicroBlaze processor (clock frequency 100 MHz) for the software module implementation. The final mapping of the various sub-systems is shown in Fig. 8.

In this implementation, an important parameter is the number of RAMs (M) present in the board as that decides the amount of parallelization that can be achieved for the "extract features" function as well as the total number of PEs N.

4.8. Results

The percentage decreases in execution times compared to serial execution are shown in Fig. 10 for the various design cases for the first two applications while the values are shown in Table 1. The results shown are for one iteration at steady state — i.e., not the first iteration, where there is additional latency due to the Gaussian white noise generator. From the tables and the figure, it may be observed that the execution performance improves the most when moving from 1 PE to 2PE. This is due to the presence of the resampling step in the particle filter algorithm that restricts the improvement in performance with increase in parallelization. Note that the execution times for both of the applications are similar because the latencies of the PEs are relatively small compared to the latency induced by P. The corresponding resource utilizations of the two implementations are shown in Table 2.

The block RAM (BRAM) memory banks available for the Virtex-4 device family are each of size 18 Kb, which is much higher than what is required for any of the implementations. Increasing P affects only the required memory bank sizes, thus the resource utilization remains the same for different numbers of particles. However, for applications with larger memory requirements, this would not be the case. The tracking performances of the two system implementations are shown in Fig. 11.

For the 3D facial pose tracking implementation the FPGA resources allowed the implementation of only a 2PE system, the resource requirements and execution results (per frame) for different values of particles P are shown in Tables 3 and 4 respectively. A Texas Instruments PDSP (TMS320c64xx processor series) implementation for this application with 50 particles yielded an execution time of 4.23s per frame. Thus, though a full hardware implementation was not used, the FPGA based hardware/software design resulted in a more efficient system. Fig. 12 shows the sample tracking results for the benchmark video while plots for the tracking are shown in Fig. 13. From Fig. 13 it may be observed that the tracking deteriorates towards the end of the video and also when there is a sudden change in movement.

5. Conclusions

In this paper, an architecture for embedded implementation of particle filter based computer vision applications is provided with a new methodology for design, modeling and design exploration for such systems on reconfigurable system-on-chips (SoCs). Our methodology uses the notion of parameterization to provide a useful tool for evaluating multiple design alternatives, and exploring the associated trade-offs in an efficient and intuitive manner. It also provides scope for implementing a wide range of applications with minimal re-design effort between different applications. From the experiments it was observed that the execution speed was determined mainly by the number of particles, and thus, the latency of the resampling unit played a significant role in determining the overall execution time. This stresses the need to look further into methods for optimizing this unit. Although multiple expensive (area-consuming) computational units were used, the area constraint imposed by the target platform was met.

References

[1] G. Aggarwal, A. Veeraraghavan, R. Chellappa, 3D facial pose tracking in uncalibrated videos, in: Proc. International Conference on Pattern Recognition and Machine Intelligence (PReMI), 2005.
[2] A. Athalye, M. Bolic, S. Hong, P.M. Djuric, Generic hardware architectures for sampling and resampling in particle filters, EURASIP Journal on Applied Signal Processing 17 (2005) 2888–2902.
[3] A.S. Bashi, V. P. Jilkov, X. R. Li, H. Chen, Distributed implementations of particle filters, in: Proc. Sixth International Conference of Information Fusion, New Orleans, pp. 1164–1171.
[4] M. Bolic, Architectures for efficient implementation of particle filters, Ph.D. Dissertation, Stony Brook University, New York, August 2004.
[5] B.P. Carlin, N.G. Polson, D.S. Stoffer, A Monte-Carlo approach to nonnormal and nonlinear state space modelling, Journal of American Statistical Association (1992) 493–500.
[6] P.P. Chu, R.E. Jones, Design techniques of FPGA based random number generator, in: Proc. Military and Aerospace Applications of Programmable Devices and Technologies Conference, The Johns Hopkins University – Applied Physics Laboratory, September 1999.
[7] R. Douc, O. Cappe, Comparison of resampling schemes for particle filtering, in: Proc. 4th International Symposium on Image and Signal Processing and Analysis, September 2005, pp. 64–69.
[8] J. Gall, B. Rosenhahn, T. Brox, H.-P. Seidel, Learning for Multi-View 3D tracking in the context of particle filters, in: Proceedings of International Symposium on Visual Computing (ISVC'06), Springer, LNCS 4292, 59–69, 2006.
[9] J. Gause, P.Y.K. Cheung, W. Luk, Reconfigurable shape-adaptive template matching architectures, in: Proc. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002, pp. 98–107.
[10] S. Hong, X. Liang, P.M. Djuric, Reconfigurable particle filter design using dataflow structure translation, in: Proc. of IEEE Workshop on Signal Processing Systems, 2004, pp. 325–330.

[11] C. Kwok, D. Fox, M. Meila, Real-time particle filters, in: Proc. of the IEEE, vol. 92, March 2004, pp. 469–484.
[12] C. Liu, P. Liu, J. Liu, J. Huang, X. Tang, 2D articulated pose tracking using particle filter with partitioned sampling and model constraints, Journal of Intelligent and Robotic Systems, 2009. doi:10.1007/s10846-009-9346-6.
[13] M. Orchard, B. Wu, G. Vachtsevanos, A particle filter framework for failure prognosis, in: Proc. WTC2005 World Tribology Congress III. Washington, DC, September 2005.
[14] S. Saha, C. Shen, C. Hsu, A. Veeraraghavan, A. Sussman, S.S. Bhattacharyya, Model-based Open MP implementation of a 3D facial pose tracking system. in: Proc. of the Workshop on Parallel and Distributed Multimedia, Columbus, August 2006, pp. 66–73.
[15] S. Saha, N.K. Bambha, S.S. Bhattacharyya, A parameterized design framework for hardware implementation of particle filters, in: Proceedings of the International Conference on Acoustics, Speech and Signal Processing, Las Vegas (NV), March 2008.
[16] A.C. Sankaranarayanan, R. Chellappa, A. Srivastava, Algorithmic and architectural design methodology for particle filters in hardware, in: Proc. of IEEE International Conference on VLSI in Computers and Processors, October 2005, pp. 275–280.
[17] A.C. Sankaranarayanan, A. Srivastava, R. Chellappa, Algorithmic and architectural optimization for computationally efficient particle filtering, IEEE Transactions on Image Processing 17 (5) (2008) 737–748.
[18] R. Velmurugan, S. Subramanian, V. Cevher, D. Abramson, K. M. Odame, J. D. Gray, H.-J. Lo, J. H. McClellan, D.V. Anderson, On low-power analog implementations of particle filters for target tracking, in: Proc. of EUSIPCO 2006, Italy, September 2006.
[19] R. Velmurugan, S. Subramanian, V. Cevher, J.H. McClellan, D.V. Anderson, Mixed-mode implementation of particle filters, in: Proc. of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, August 2007, pp. 617–620.
[20] J. Yao, J.-M. Odobez, Multi-camera 3D person tracking with particle filter in a surveillance environment, in: Proceedings of 16th European Signal Processing Conference (EUSIPCO 2008), Lausanne, Switzerland, August 25–29, 2008.