Titre: Title:	Efficient Implementation of Particle Filters in Application-Specific Instruction-Set Processor
Auteur: Author:	Qifeng Gan
Date:	2013
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gan, Q. (2013). Efficient Implementation of Particle Filters in Application-Specific Instruction-Set Processor [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1169/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1169/
Directeurs de recherche: Advisors:	Yvon Savaria, & J. M. Pierre Langlois
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

EFFICIENT IMPLEMENTATION OF PARTICLE FILTERS IN APPLICATION-SPECIFIC INSTRUCTION-SET PROCESSOR

QIFENG GAN
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
JUIN 2013

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

EFFICIENT IMPLEMENTATION OF PARTICLE FILTERS IN APPLICATION-SPECIFIC INSTRUCTION-SET PROCESSOR

présentée par : GAN Qifeng

en vue de l'obtention du diplôme de : Philosophiæ Doctor

a été dûment acceptée par le jury d'examen constitué de :

M. BOYER François-Raymond, Ph.D., président

M. LANGLOIS J.M. Pierre, Ph.D., membre et directeur de recherche

M. SAVARIA Yvon, Ph.D., membre et codirecteur de recherche

M. BELTRAME Giovanni, Ph.D., membre

M. MANJIKIAN Naraig, Ph.D., membre

DEDICATION

Dedicated to my family

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to Dr. J.M. Pierre Langlois for his supervision during my Ph.D. study and research. Without his foresight and encouragement, this thesis could not be completed. What I have learned from Dr. Langlois is not only the abilities to research and investigate important problems, but also the techniques and attitude of being a researcher and engineer. I am also very grateful to my co-supervisor, Dr, Yvon Savaria. His broad knowledge and excellent insight and ideas always inspire me to pursue high standard research. His continuous help is one of the most important supports to me towards graduation with Ph.D. degree.

I would like to thank my colleagues at Polytechnique Montreal with whom I have had the pleasure of working over the years. These include Rana Farah, with whom I worked closely on the project, Shervin Vakili, Frédéric Blouin, Mohamed Taboubi, Diana Carolina Gil and all other members of the Digital Systems Laboratory.

I want to thank my parents (Jianmin Gan and Fuzhen Jin) for their unconditional love, support and encouragements every step of the way. They are the reason why I am always looking forward and moving forward.

Last, but not the least, to my family, my two daughters (Fiona Gan and Peggy Gan) are the greatest motivation and happiness during all the time of my Ph.D. life. I’m truly grateful to my wife, Jingyun Yang, for all your sacrifice, patience, support, understanding and most of all, for your love.

RÉSUMÉ

Cette thèse considère le problème de l'implémentation de filtres particulaires (*particle filters* PFs) dans des processeurs à jeu d'instructions spécialisé (*Application-Specific Instruction-set Processors* ASIPs). Considérant la diversité et la complexité des PFs, leur implémentation requiert une grande efficacité dans les calculs et de la flexibilité dans leur conception. La conception de ASIPs peut se faire avec un niveau intéressant de flexibilité. Notre recherche se concentre donc sur l'amélioration du débit des PFs dans un environnement de conception de ASIP.

Une approche générale est tout d'abord proposée pour caractériser la complexité computationnelle des PFs. Puisque les PFs peuvent être utilisés dans une vaste gamme d'applications, nous utilisons deux types de blocs afin de distinguer les propriétés des PFs. Le premier type est spécifique à l'application et le deuxième type est spécifique à l'algorithme. Selon les résultats de profilage, nous avons identifié que les blocs du calcul de la probabilité et du rééchantillonnage sont les goulots d'étranglement principaux des blocs spécifiques à l'algorithme. Nous explorons l'optimisation de ces deux blocs aux niveaux algorithmique et architectural.

Le niveau algorithmique offre un grand potentiel d'accélération et d'amélioration du débit. Notre travail débute donc à ce niveau par l'analyse de la complexité des blocs du calcul de la probabilité et du rééchantillonnage, puis continue avec leur simplification et modification. Nous avons simplifié le bloc du calcul de la probabilité en proposant un mécanisme de quantification uniforme, l'algorithme UQLE. Les résultats démontrent une amélioration significative d'une implémentation logicielle, sans perte de précision. Le pire cas de l'algorithme UQLE implémenté en logiciel à virgule fixe avec 32 niveaux de quantification atteint une accélération moyenne de 23.7× par rapport à l'implémentation logicielle de l'algorithme ELE. Nous proposons aussi deux nouveaux algorithmes de rééchantillonnage pour remplacer l'algorithme séquentiel de rééchantillonnage systématique (SR) dans les PFs. Ce sont l'algorithme SR reformulé et l'algorithme SR parallèle (PSR). L'algorithme SR reformulé combine un groupe de boucles en une boucle unique afin de faciliter sa parallélisation dans un ASIP. L'algorithme PSR rend les itérations indépen-

dantes, permettant ainsi à l'algorithme de rééchantillonnage de s'exécuter en parallèle. De plus, l'algorithme PSR a une complexité computationnelle plus faible que l'algorithme SR.

Du point de vue architectural, les ASIPs offrent un grand potentiel pour l'implémentation de PFs parce qu'ils présentent un bon équilibre entre l'efficacité computationnelle et la flexibilité de conception. Ils permettent des améliorations considérables en débit par l'inclusion d'instructions spécialisées, tout en conservant la facilité relative de programmation de processeurs à usage général. Après avoir identifié les goulots d'étranglement de PFs dans les blocs spécifiques à l'algorithme, nous avons généré des instructions spécialisées pour les algorithmes UQLE, SR reformulé et PSR. Le débit a été significativement amélioré par rapport à une implémentation purement logicielle tournant sur un processeur à usage général. L'implémentation de l'algorithme UQLE avec instruction spécialisée avec 32 intervalles atteint une accélération de 34× par rapport au pire cas de son implémentation logicielle, avec 3.75 K portes logiques additionnelles. Nous avons produit une implémentation de l'algorithme SR reformulé, avec quatre poids calculés en parallèle et huit catégories définies par des bornes uniformément distribuées qui sont comparées simultanément. Elle atteint une accélération de 23.9× par rapport à l'algorithme SR séquentiel dans un processeur à usage général. Le surcoût est limité à 54 K portes logiques additionnelles. Pour l'algorithme PSR, nous avons conçu quatre instructions spécialisées configurées pour supporter quatre poids entrés en parallèle. Elles mènent à une accélération de 53.4× par rapport à une implémentation de l'algorithme SR en virgule flottante sur un processeur à usage général, avec un surcoût de 47.3 K portes logiques additionnelles.

Finalement, nous avons considéré une application du suivi vidéo et implémenté dans un ASIP un algorithme de FP basé sur un histogramme. Nous avons identifié le calcul de l'histogramme comme étant le goulot principal des blocs spécifiques à l'application. Nous avons donc proposé une architecture de calcul d'histogramme à réseau parallèle (PAHA) pour ASIPs. Les résultats d'implémentation démontrent qu'un PAHA à 16 voies atteint une accélération de 43.75× par rapport à une implémentation logicielle sur un processeur à usage général.

ABSTRACT

This thesis considers the problem of the implementation of particle filters (PFs) in Application-Specific Instruction-set Processors (ASIPs). Due to the diversity and complexity of PFs, implementing them requires both computational efficiency and design flexibility. ASIP design can offer an interesting degree of design flexibility. Hence, our research focuses on improving the throughput of PFs in this flexible ASIP design environment.

A general approach is first proposed to characterize the computational complexity of PFs. Since PFs can be used for a wide variety of applications, we employ two types of blocks, which are application-specific and algorithm-specific, to distinguish the properties of PFs. In accordance with profiling results, we identify likelihood processing and resampling processing blocks as the main bottlenecks in the algorithm-specific blocks. We explore the optimization of these two blocks at the algorithmic and architectural levels.

The algorithmic level is at a high level and therefore has a high potential to offer speed and throughput improvements. Hence, in this work we begin at the algorithm level by analyzing the complexity of the likelihood processing and resampling processing blocks, then proceed with their simplification and modification. We simplify the likelihood processing block by proposing a uniform quantization scheme, the Uniform Quantization Likelihood Evaluation (UQLE). The results show a significant improvement in performance without losing accuracy. The worst case of UQLE software implementation in fixed-point arithmetic with 32 quantized intervals achieves 23.7× average speedup over the software implementation of ELE. We also propose two novel resampling algorithms instead of the sequential Systematic Resampling (SR) algorithm in PFs. They are the reformulated SR and Parallel Systematic Resampling (PSR) algorithms. The reformulated SR algorithm combines a group of loops into a parallel loop to facilitate parallel implementation in an ASIP. The PSR algorithm makes the iterations independent, thus allowing the resampling algorithms to perform loop iterations in parallel. In addition, our proposed PSR algorithm has lower computational complexity than the SR algorithm.

At the architecture level, ASIPs are appealing for the implementation of PFs because they strike a good balance between computational efficiency and design flexibility. They can provide considerable throughput improvement by the inclusion of custom instructions, while retaining the ease of programming of general-purpose processors. Hence, after identifying the bottlenecks of PFs in the algorithm-specific blocks, we describe customized instructions for the UQLE, reformulated SR, and PSR algorithms in an ASIP. These instructions provide significantly higher throughput when compared to a pure software implementation running on a general-purpose processor. The custom instruction implementation of UQLE with 32 intervals achieves 34× speedup over the worst case of its software implementation with 3.75 K additional gates. An implementation of the reformulated SR algorithm is evaluated with four weights calculated in parallel and eight categories defined by uniformly distributed numbers that are compared simultaneously. It achieves a 23.9× speedup over the sequential SR algorithm in a general-purpose processor. This comes at a cost of only 54 K additional gates. For the PSR algorithm, four custom instructions, when configured to support four weights input in parallel, lead to a 53.4× speedup over the floating-point SR implementation on a general-purpose processor at a cost of 47.3 K additional gates.

Finally, we consider the specific application of video tracking, and an implementation of a histogram-based PF in an ASIP. We identify that the histogram calculation is the main bottleneck in the application-specific blocks. We therefore propose a Parallel Array Histogram Architecture (PAHA) engine for accelerating the histogram calculation in ASIPs. Implementation results show that a 16-way PAHA can achieve a speedup of 43.75× when compared to its software implementation in a general-purpose processor.

TABLE OF CONTENTS

DEDICATION III

ACKNOWLEDGEMENTS IV

RÉSUMÉ V

ABSTRACT VII

TABLE OF CONTENTS IX

LIST OF TABLES XII

LIST OF FIGURES XIV

LIST OF ABBREVIATIONS XVI

Chapter 1 INTRODUCTION 1

- 1.1 Overview 1
- 1.2 Motivation and Challenges 1
- 1.3 Research Objectives 3
- 1.4 Contributions and Published Work 4
- 1.5 Organization of the Thesis 5

Chapter 2 LITERATURE REVIEW 7

- 2.1 Particle Filters 7
- 2.2 Applications of Particle Filters 15
  - 2.2.1 Target Tracking 16
  - 2.2.2 Video Tracking 17
- 2.3 PF Implementation in Embedded Systems 19
  - 2.3.1 Implementation Methods in Embedded Systems 19
  - 2.3.2 Examples of PF Implementations 21
- 2.4 Application-Specific Instruction-set Processors 23

2.4.1 Overview 23
2.4.2 Processor Generation 24
2.4.3 Automatic Generation of Custom Instructions 26
Chapter 3 EFFICIENT UNIFORM QUANTIZATION LIKELIHOOD EVALUATION FOR PARTICLE FILTERS IN EMBEDDED IMPLEMENTATIONS 28
3.1 Introduction 28
3.2 Computational Characteristics of PFs 31
3.2.1 Description of DSS Examples 31
3.2.2 Computational Complexity of SIR PFs 33
3.3 Proposed UQLE Algorithm 36
3.4 UQLE Accuracy Evaluation 39
3.4.1 Implementation Environment and Accuracy Metrics 39
3.4.2 Accuracy for the LG Model 40
3.4.3 Accuracy for the UNG Model 41
3.4.4 Accuracy for the BOT Model 43
3.5 UQLE Throughput Evaluations 45
3.5.1 UQLE Software Implementation and its Performance 45
3.5.2 UQLE ASIP Implementation and its Performance 47
3.6 Conclusion 50
Chapter 4 NOVEL SYSTEMATIC RESAMPLING ALGORITHMS AND THEIR IMPLEMENTATION IN ASIPS 51
4.1 Introduction 51
4.2 Reformulated SR Algorithm and its ASIP Implementation 53
4.2.1 The Reformulated SR Algorithm 53
4.2.2 ASIP Implementation 55

4.2.3 Performance ... 57
4.3 Parallel Systematic Resampling Algorithm and its ASIP Implementation ... 59
4.3.1 Parallel Systematic Resampling Algorithm ... 59
4.3.2 Fixed-point Implementation of the PSR Algorithm ... 60
4.3.3 ASIP Implementation of the Fixed-Point PSR Algorithm ... 61
4.3.4 Implementation Results ... 63
4.4 Conclusion ... 65
Chapter 5 PARALLEL ARRAY HISTOGRAM ARCHITECTURE FOR EMBEDDED IMPLEMENTATIONS ... 66
5.1 Introduction ... 66
5.2 Bottleneck in Histogram-based PFs for Video Tracking ... 67
5.3 Parallel Array Histogram Architecture ... 69
5.4 Flexible Parallel Array Histogram Architecture ... 70
5.5 Architecture Comparisons ... 71
5.6 Implementation Results ... 72
5.7 Conclusion ... 73
Chapter 6 CONCLUSIONS AND FUTURE WORK ... 74
6.1 Thesis Conclusions ... 74
6.2 Future Work ... 75
REFERENCES ... 77

LIST OF TABLES

Table 3.1 Average clock cycles of LG, UNG, and BOT models using SIR PFs with 512 particles in the Xtensa LX2 processor without FPU 35

Table 3.2 Average cycles of LG, UNG, and BOT models using SIR PFs with 512 particles in the Xtensa LX2 processor with FPU 35

Table 3.3 RMSE of the SIR PF using ELE for the LG model 40

Table 3.4 RMSE of SIR PFs with ELE for the UNG model. 42

Table 3.5 Combined MSE and Lost track of SIR PFs with ELE for the BOT model 43

Table 3.6 Execution time and speedup of UQLE for the worst and best case in the Xtensa LX2 processor 46

Table 3.7 Average execution time and speedup of UQLE for the LG, UNG and BOT models in the Xtensa LX2 processor 47

Table 3.8 Execution time and additional area of UQLE using its custom instruction in PFs in the Xtensa LX2 processor 49

Table 3.9 Speedup between running in GPP and in GPP with UQLE instruction for the LG, UNG, and BOT models 50

Table 4.1 Custom instructions for the reformulated SR algorithm 56

Table 4.2 Performance and resource requirements for the reformulated SR algorithm in ASIP implementation 58

Table 4.3 Custom instructions for the proposed PSR algorithm 62

Table 4.4 Number of times that incorrect RFs were generated by fixed-point PSR algorithm for 10000 simulations. 63

Table 4.5 Performance and resource requirements of various implementation methods 64

Table 5.1 Profiling results of histogram-based PF with 256 particles tracking an object (120 by 140 pixels) in the Xtensa LX2 processor 68

Table 5.2 Comparison of PAHA and FPAHA against previous work for the case of (M, N, P) = (8, 8, 20) 72

Table 5.3 Performance and resource requirements of various M-way PAHAs on the Xtensa LX2 processor..73

LIST OF FIGURES

Figure 2-1 Functional view of the resampling processing step..10

Figure 2-2 Data flow graph for the SR algorithm 11

Figure 2-3 Data flow graph for the RSR algorithm 13

Figure 2-4 Pseudo-code of SIR particle filter 15

Figure 2-5 Comparing GPPs, DSPs, ASIPs and CPs...20

Figure 3-1 Functional view of the SIR PF 33

Figure 3-2 Likelihood evaluation curve for the proposed UQLE with $M+1= 5$ intervals............38

Figure 3-3 Pseudo-code of the proposed UQLE algorithm..39

Figure 3-4 Maximum, average and minimum RMSE results of SIR PFs using UQLE for the LG model..41

Figure 3-5 Maximum, average, and minimum RMSE results of SIR PFs using UQLE for the UNG model 42

Figure 3-6 Maximum, average and minimum combined MSE and lost track for SIR PFs using UQLE 44

Figure 3-7 Tracking of a moving target in two dimensions with SIR PFs using ELE and SIR PFs using UQLE with 2 intervals and high value. 45

Figure 3-8 Custom UQLE logic organization with 4 intervals 48

Figure 4-1 PF steps...52

Figure 4-2 Data flow graph of the reformulated SR algorithm..54

Figure 4-3 Pseudo-code of the parallel while loops...55

Figure 4-4 Pseudo-code of the reformulated SR algorithm 57

Figure 4-5 Speedup vs additional gates for various solutions..58

Figure 4-6 Data flow graph of the proposed Parallel Systematic Resampling algorithm.............60

Figure 4-7 Pseudo-code of the PSR algorithm in ASIP implementations 62

Figure 5-1 Parallel Array Histogram Architecture 69

Figure 5-2 M-bit Accumulator 70

Figure 5-3 Flexible Parallel Array Histogram Architecture 70

LIST OF ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
ASIR	Auxiliary Sample-Importance-Resampling
BOT	Bearing-Only Tracking
CP	Custom Processor
CW	Cumulative Weight
DC	Distance Calculation
DSP	Digital Signal Processor
DSS	Dynamic State Space
EKF	Extended Kalman Filter
ELE	Exact Likelihood Evaluation
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GPGPU	General-Purpose computing on Graphics Processing Unit
IG	Index Generation
KF	Kalman Filter
LE	Likelihood Evaluation
LG	Linear Gaussian
MAP	Maximum A Posteriori
MMSE	Minimum Mean-Square Error
PA	Particle Allocation

PAHA	Parallel Array Histogram Architecture
PDF	Probability Distribution Function
PF	Particle Filter
PG	Prediction Generation
PMP	Particle Measurement Processing
PSR	Parallel Systematic Resampling
RA	Resampling Algorithm
RF	Replication Factor
RFG	Replication Factor Generation
RMSE	Root-Mean-Square Error
RNA	Random Number Addition
ROI	Region Of Interest
RP	Resampling Processing
RR	Residual Resampling
RSR	Residual Systematic Resampling
SIS	Sequential Importance Sampling
SIR	Sample Importance Resampling
SR	Systematic Resampling
TP	Transition Processing
UDN	Uniformly Distributed Number
UNG	Uni-variate Non-stationary Growth
UQLE	Uniform Quantization Likelihood Evaluation
WC	Weight Calculation

Chapter 1 INTRODUCTION

1.1 Overview

Particle Filters (PFs) [1] are statistical signal processing methods that perform sequential Monte Carlo estimation based on a particle representation of probability densities. Over the past decade, they have gained in popularity to address various applications with nonlinear models and/or non-Gaussian noise due to their competitive accuracy in comparison with the Extended Kalman Filter (EKF). PFs use the concept of importance sampling to recursively compute the relevant probability distributions conditioned on the observations. In comparison with the EKF, PFs do not rely on linearization techniques and can robustly approximate the true system state with an appropriate number of particles. In contrast, the EKF sometimes has poor performance, lacks robustness, and may introduce large biases [2].

In practice, PFs have shown great promise as a powerful framework in addressing a wide range of complex applications that include target tracking, navigation, robotics and computer vision. Fox example, in target tracking, the Bearings-Only Tracking (BOT) application [1] [2] only employs noisy measurement of angular positions to estimate the object position and velocity. In computer vision, color-based PFs for video tracking [3] estimate the object position by calculating the distance between the reference color histogram and the current color histograms. In these applications, PFs achieve higher accuracy than other filters [2] [3] .

1.2 Motivation and Challenges

Because PFs are effective and popular filters for applications with nonlinear models and/or non-Gaussian noise, a vast amount of literature can be found on their theory, applications and implementations. To our knowledge, when implementing PFs in embedded systems, most works only focus on how to improve their throughput and thus implement them on dedicated hardware due to their high computational complexity. However, PFs still require design flexibility to favor a large range of various applications. There has been no effort to consider the important factor of design flexibility. Hence, we aim to fill that gap and demonstrate flexible approaches to PFs implementation. These alternatives to dedicated hardware design should achieve the same throughput requirements of target applications. In this thesis, we specifically consider tracking applications. These applications are characterized by complex target models. Meanwhile, in order to achieve a high level of accuracy, the number of particles should be as large as possible. In such a context, PFs must achieve high throughput and have low latency. In the context of an embedded system, the implementation should occupy a small area and consume as little power as possible. A final but important requirement, which we consider in this thesis, is that the system should be as flexible as possible to accommodate changes and support updates.

An appealing method is using Application-Specific Instruction-set Processors (ASIPs). ASIPs strike a good balance between efficient custom processor design and low-cost flexible general-purpose processor design. They maintain the flexibility of a programmable solution while improving the throughput with the instantiation of dedicated customized instructions and/or special purpose registers. Hence, the main objective of our research is to develop and implement PFs in ASIPs to achieve the desired requirements of tracking applications in the throughput and silicon area.

There are two main reasons for the heavy computational requirements in PFs. The first reason is that a large number of particles are often required to achieve the desired accuracy. As the number of particles increases, the processing speed of the particle filters tends to be seriously degraded. The second reason arises from the type of operations involved in the PFs. PFs may require nonlinear operations such as division and exponentiation. These expensive and complex operations are often important bottlenecks in the embedded implementation of PFs.

In order to meet the throughput requirements of tracking applications, we identify the bottlenecks at the algorithm and architecture levels. In accordance with the need for a large amount of particles in PFs, a parallel implementation is an appealing method to improve PF throughput. In ASIP design, parallelism can be exploited at the instruction level. However, standard resampling algorithms such as the Systematic Resampling (SR) algorithm [4] are sequential in nature. It may become the most time consuming portion when parallelism is employed in implementing the PFs. Making resampling algorithms feasible for further parallel PF implementation becomes a difficult challenge in order to improve PF throughput. Concerning the problem on nonlinear complex operations involved in PFs, it is necessary to simplify or avoid using such operations. However, simplification in PFs may affect the accuracy for tracking applications. Hence, this problem motivates us to find how to simplify the complex operations in PFs without losing accuracy.

At the architecture level, hardware unit design for the relevant bottlenecks in PFs is required in order to further improve the throughput of tracking applications in ASIPs. However, the design of ASIPs is constrained by the number of input and output operands for custom instructions and by the memory bandwidth. Under these restrictions, exploiting the data parallelism and/or fusing atomic operations at the heart of execution bottlenecks become the main challenges in designing specific hardware units in ASIPs.

1.3 Research Objectives

The main goal of this research is to design and implement PFs in ASIPs to achieve high throughput with a certain level of flexibility. In order to reach this goal, the following specific objectives are identified:

- Analyze and characterize PFs in the context of their implementation in ASIPs in order to identify opportunities for acceleration and implementation efficiency.

- In accordance with PF characteristics, propose new and efficient PF algorithms to reduce the impact of complex operations that exist in conventional PF algorithms.

- Propose new formulations of resampling algorithms for PFs in order to accelerate their execution.

- Identify bottlenecks for histogram-based PFs for video tracking and propose relevant hardware architectures for them in ASIPs.

- Simulate, implement, test and evaluate several ASIP designs of PFs to assess the performance of the proposed algorithms and architectures.

1.4 Contributions and Published Work

This thesis presents six main contributions that have been presented in papers published in or submitted to scientific journals and international conferences.

I. A novel characterization of PFs

Our proposed characterization of PFs, unlike previous work [5] [6], focuses on distinguishing the application-specific blocks and the algorithm-specific blocks. This contribution was presented in:

Qifeng Gan, J.M.P. Langlois, Y. Savaria, "Efficient Uniform Quantization Likelihood Evaluation for Particle Filters in Embedded Implementations," accepted by *Journal of Signal Processing Systems* on 29th May 2013, [7].

II. Simplified likelihood evaluation algorithm and its implementation in ASIPs

We proposed an efficient Uniform Quantization Likelihood Evaluation (UQLE) algorithm to replace the Exact Likelihood Evaluation (ELE) algorithm in PFs. Simulation results indicate that PFs using UQLE can achieve comparable or better accuracy than the PFs using ELE. We also implemented UQLE in an ASIP to achieve higher throughput. This contribution was presented in the same paper [7] as Contribution I.

III. Reformulated systematic resampling algorithm and its implementation in ASIPs

We proposed a reformulated systematic resampling algorithm instead of the Sequential SR algorithm. Our proposed reformulated systematic resampling is suitable for parallel implementation in ASIPs. The idea and results are presented in:

Qifeng Gan, J.M.P. Langlois, Y. Savaria, "A Reformulated Systematic Resampling and its Parallel Implementation on Application-Specific Instruction-set Processors," accepted by *MWSCAS 2013* on 10th May 2013, [8].

IV. Parallel systematic resampling algorithm and its implementation in ASIPs

We proposed a Parallel Systematic Resampling (PSR) algorithm for PFs. This algorithm makes iterations independent, thus allowing the resampling algorithm to perform its iterations in parallel. The idea and results are presented in:

Qifeng Gan, J.M.P. Langlois, Y. Savaria, "Parallel Systematic Resampling Algorithm for Particle Filters," submitted to *Journal of Circuits, Systems and Signal Processing* on 10th Apr. 2013, [9].

V. Parallel array histogram architecture engine design for histogram-based particle filtering video tracking systems.

We designed a Parallel Array Histogram Architecture (PAHA) engine, where multiple elements can be processed in parallel to update the histogram bins, to accelerate the histogram calculation for further use in histogram-based particle filters for video tracking. The idea and results were published in:

Qifeng Gan, J.M.P. Langlois, Y. Savaria, "Parallel Array Histogram Architecture for Embedded Implementations," *Electronics Letters*, vol.49, issue 2, January 2013, [10].

VI. Histogram-based PFs for video tracking implementation

In collaboration with Ms. R. Farah, we explored the simplification of PFs for video tracking and made them better suitable for embedded implementation.

R. Farah, Q. Gan, J.M.P Langlois, G.-A. Bilodeau, Y. Savaria, "A tracking algorithm suitable for embedded systems implementation," *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*, pp.256-259, 11-14 Dec. 2011, [11].

1.5 Organization of the Thesis

This thesis is organized in six chapters including this introduction. In Chapter 2, an overview of related works is presented. These works are separated into four main parts: theory of PFs, applications of PFs, PFs embedded implementation, and ASIPs.

In Chapter 3, we present a novel characterization of PFs which focuses on distinguishing the application-specific blocks and the algorithm-defined blocks. After profiling three applications, the likelihood processing block is identified as the most time consuming block in the algorithm-defined blocks. We also present a simplified likelihood evaluation algorithm based on a uniform quantization scheme to reduce its complexity. Moreover, this simplified likelihood evaluation algorithm was implemented in the Xtensa LX2 processor with a customized instruction.

In Chapter 4, we focus on modifying the sequential resampling algorithm in PFs and make it feasible for executing in parallel in ASIPs. Two novel resampling algorithms, reformulated SR and PSR algorithms, are proposed in accordance with the sequential SR algorithm. The reformulated SR algorithm is still a sequential algorithm but it can be executed in parallel in ASIPs with custom instructions. The PSR algorithm is a parallel algorithm because it makes the iterations in the sequential SR algorithm independent, thus allowing the iterations to be performed in parallel. We implement these two resampling algorithms in the Xtensa LX2 processor with several relevant customized instructions to show their significant speedup over the sequential SR algorithm.

In Chapter 5, we identify that histogram calculation is major bottleneck in histogram-based PFs for video tracking and present a PAHA engine to accelerate histogram calculation. It can process multiple elements to update the histogram bins. We also introduce a second version of PAHA with a flexible number of inputs, potentially avoiding the need for multiple PAHAs in a single application. The PAHA engine can significantly accelerate histogram calculation and thus improve the throughput of histogram-based particle filters for video tracking.

Chapter 6 concludes the thesis and discusses potential future work.

Chapter 2 LITERATURE REVIEW

In this chapter, we examine the relevant background in the areas of theory of Particle Filters (PFs), applications of PFs, implementation of PFs, and Application-Specific Instruction-set Processor (ASIP) design.

2.1 Particle Filters

PFs are sequential Monte Carlo methods [12] in which the a posteriori probability distribution function (PDF) is approximated by the importance weights of samples. Since the demonstration of their accuracy in comparison with other filters and their flexibility in addressing a vast amount of problems [2], PFs have gained in popularity to address applications with non-linear models and/or non-Gaussian noise. PFs were initially introduced as the bootstrap filter by Gordon et al. [2] [13]. They showed that PFs outperform the Extended Kalman Filter (EKF) in a Bearing-Only Tracking (BOT) application. Kitagawa [14] studied PFs as Monte Carlo Filter for prediction, filtering, and smoothing. Carpenter et al. [15] improved the quality of PFs by solving the sample impoverishment problem [13]. Many tutorials and books [1] [4] [16] [17] [18] [19] [20] have been published to review PFs and their applications. Aydogums et al. [21] used the EKF and PFs to perform sensorless speed control. They showed again that the estimation performance of PFs is more accurate than the EKF in applications with nonlinear models and/or non-Gaussian noise.

PFs are often applied to problems that can be written in the form of Dynamic State Space (DSS) models. Consider the general form of a DSS model expressed by Equations (2.1) and (2.2):

$$x_t = f_{t-1}(x_{t-1}, u_{t-1}), \tag{2.1}$$

$$z_t = h_t(x_t, w_t), \tag{2.2}$$

where $t \in \mathbb{N}$ is the time index, x_t is the target state vector, and z_t is the observation vector. f_{t-1} and h_t are possibly nonlinear prediction functions of the state $x_{t-1}$and observation functions of the state x_t, respectively. u_{t-1} is the process noise sequence and w_t is the observation noise sequence. Equation (2.1) describes how the state vector x_t evolves with time. Equation (2.2) develops the noisy observation vector z_t as a function of the state vector x_t. In PFs, the aim is to learn about the unobserved state based on a set of noisy observations as time evolves.

PFs base their operations on approximating the a posteriori PDF using a set of *N* weighted samples $\{x_{0:t}^i, \omega_t^i\}_{i=1}^N$, called particles. These particles are drawn independently from an importance density $q(x_t|x_{t-1},\ z_{1:t})$. Accordingly, if the observation $z_{1:t}\ = \{z_j, j = 1, \dots, t\}$ is available and the weights are normalized such that $\sum_i \omega_t^i = 1$, the a posteriori PDF at time t can be approximated as

$$p(x_{0:t}|z_{1:t}) \approx \sum_{i=1}^N \omega_t^i \delta(\, x_{0:t} - x_{0:t}^i) \tag{2.3}$$

where $x_{0:t} = \{x_j, j = 0, \dots, t\}$ is the set of all states up to time t.

After determining the a posteriori PDF $p(x_{0:t}|z_{1:t})$, the state estimate x_t can be computed using either the minimum mean-square error (MMSE), the maximum a posteriori probability (MAP), or other methods. The MMSE estimate is the conditional mean of x_t:

$$\hat{x}_{t|t}^{MMSE} = \int x_t\, p(x_{0:t}|z_{1:t}) dx_t, \tag{2.4}$$

while the MAP estimate is the maximum of $p(x_t|\boldsymbol{Z}_t)$:

$$\hat{x}_{t|t}^{MAP} = \arg\max_{x_t} p(x_{0:t}|z_{1:t}). \tag{2.5}$$

In the implementation of PFs, the a posteriori PDF $p(x_{0:t}|z_{1:t})$ may be obtained recursively through the three basic steps described below:

1) *Prediction Generation*: The particle x_t^i is drawn by the importance density $q(x_t|x_{t-1},\ z_{1:t})$. The choice of the importance density plays a fundamental role in the design of PFs because generating the particles and the relevant weights is related to this importance density [16] . If the drawn particles are in regions of the space where the density has negligible values, the estimation obtained from the particles with their associated weights would be poor and it would easily lead to a failure in the estimation. A standard scheme is to choose the state transition prior $p(x_t|x_{t-1})$ defined by equation (2.1) as the importance density. PFs using this scheme are known as the

bootstrap filter [2]. The advantage of this scheme is that it is efficient to implement. However, this transition prior does not take the current observation into account. When the likelihood distribution $p(z_t|x_t)$ is narrow with respect to the transition prior distribution $p(x_t|x_{t-1})$, many particles will receive negligible weights. This situation can easily lead to the degeneracy problem.

2) *Weight Calculation*: When the likelihood distribution $p(z_t|x_t)$ is obtained upon the arrival of the observation z_t, the particle weights can be computed via the weight update equation as follows:

$$\omega_t^i = \omega_{t-1}^i \frac{p(z_t|x_t^i)p(x_t^i|x_{t-1}^i)}{q(x_t^i|x_{t-1}^i, z_{1:t})}. \tag{2.6}$$

Normalization is carried out as follows:

$$\widetilde{\omega}_t^i = \frac{\omega_t^i}{\sum_{i=1}^N \omega_t^i}. \tag{2.7}$$

3) *Resampling Processing*: The degeneracy problem, where after a few iterations most weights have a value of zero and only a few weights retain a substantial value, often occurs in PFs, especially when the importance density is the state transition prior $p(x_t|x_{t-1})$. This problem can strongly affect the overall application accuracy [1]. The resampling processing step is a critical process in PFs because it can overcome the degeneracy of particles. It replicates the particles in proportion to their weights, which allows the particles to be concentrated more in domains of higher posterior probability. More specifically, new particles $\tilde{x}_t^i$ are drawn from the set of particles x_t^i based on the particle weights ω_t^i through a resampling scheme. The resampled set of new particles and their weights is denoted by $\{\hat{x}_t^i, \widehat{\omega}_t^i\}$.

The resampling processing step can be decomposed into three blocks, shown Fig. 2-1. They are Replication Factors Generation (RFG), Index Generation (IG), and Particle Allocation (PA). The RFG block calculates the Replication Factors (RFs) for the particles according to their weights. The RFs show how many times each particle is replicated as a result of resampling. The IG block is optional. The IG algorithm [5] can be used to replace the need to temporarily store replicated particle states with the array indexes. Particle states can occupy a significant amount of memory, which is proportional to the number of particles and to the number of dimensions in the particle states, while the memory for the array indexes only is proportional to the number of particles. The IG block may prevent parallel execution of the resampling processing step due to non-predetermined memory accesses for allocating particles. In the PA block, the memory space occupied by the discarded particles can be re-allocated to the replicated particles if the IG block is used. If not, particles can be replicated into the new memory according to their RFs. The parallel implementation of the PA block depends on the implementation technology. Bolic et al. [22] introduced particle proportional allocation and particle non-proportional allocation algorithms for implementing the PA block in parallel on a distributed architecture. Hwang et al. [23] used a load balanced particle replication algorithm for parallelizing the PA block on Graphics Processing Units (GPUs).

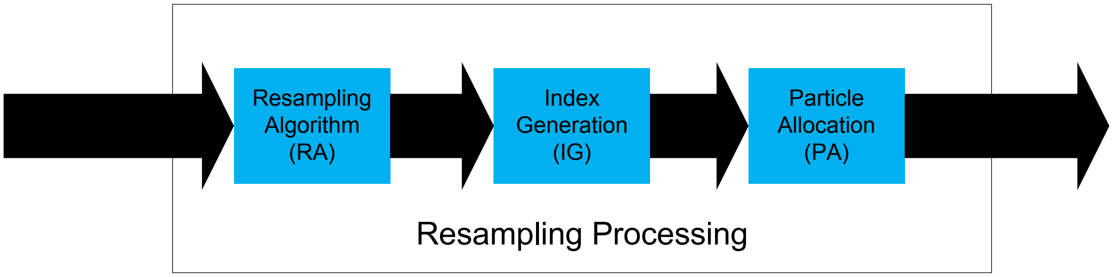


Figure 2-1 Functional view of the resampling processing step

Two main types of resampling algorithms (RAs) are used in the RFG block to calculate the RFs: threshold-based and standard RAs. Threshold-based RAs include the partial resampling [5] and compact resampling [24] [25] algorithms. Their accuracy is strongly dependent on proper threshold selection. Although they can have lower computational complexity than standard RAs and can potentially be implemented in a parallel architecture, their overall accuracy tends to be degraded [5] [25]. Standard RAs are derived from the stratified resampling [15] algorithm. They do not require the adjustment of thresholds or other parameters. The Systematic Resampling (SR) [4], Residual Resampling (RR) [26] and Residual Systematic Resampling (RSR) [5] [27] algorithms are the three most common standard RAs in use.

The SR algorithm calculates the RFs by comparing the Cumulative Weights (CWs) $\{CW_t^i\}_{i=1}^N$ with a set of corresponding Uniformly Distributed Numbers (UDNs) U^m, for $m = 1, \dots, M$. Here

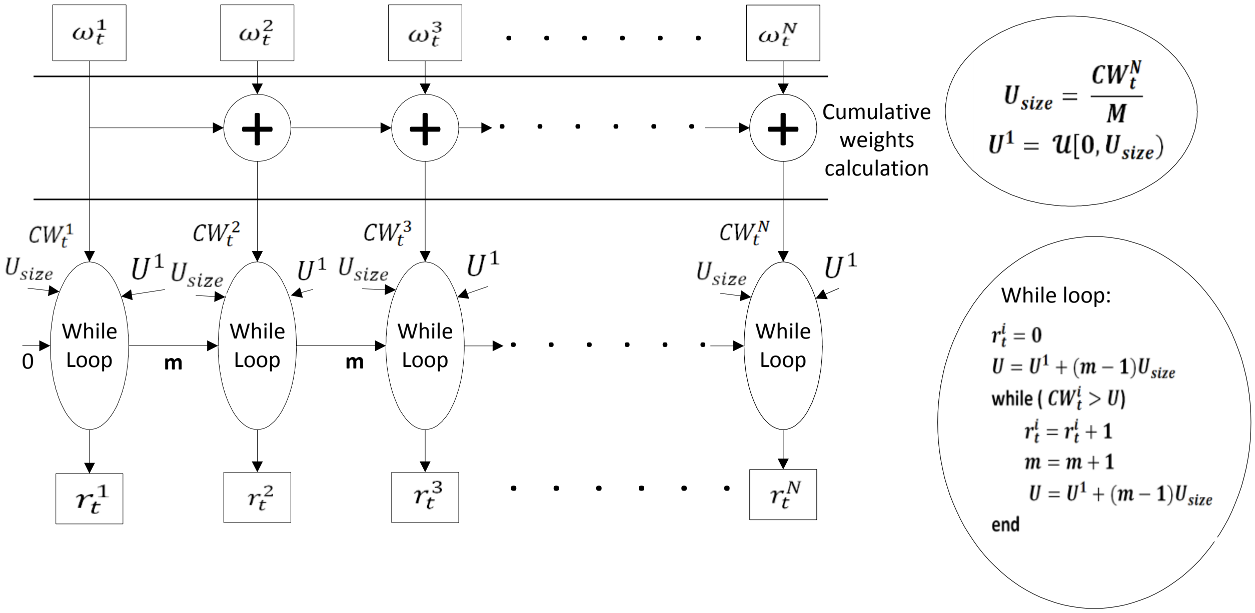
N is number of input particles and M is the number of resampled particles. U^m is given by equation (2.8).

$$U^m = \begin{cases} \mathcal{U}[0, U_{size}), & m = 1 \\ U^1 + (m-1)U_{size}, & m = 2, \dots, M \end{cases} \tag{2.8}$$

where $\mathcal{U}$ is a random number distributed uniformly in the specified interval, and U_{size} is the interval size defined by equation (2.9).

$$U_{size} = \frac{CW_t^N}{M} \tag{2.9}$$

Fig. 2-2 shows the data flow graph of the SR algorithm. The CWs $\{CW_t^i\}_{i=1}^N$ are calculated in a cascade of adders. U^1 and U_{size} are generated according to equations (2.8) and (2.9). The RFs $\{r_t^i\}_{i=1}^N$ are determined as the number of times the CWs are greater than the corresponding UDNs. For each RF, this is implemented using a while loop with a non-predetermined number of iterations. In software, these while loops would be placed inside a *for* loop of N iterations. In Fig. 2-2, this outer *for* loop is shown unrolled. Each while loop produces two results: a RF and an index m used by the following while loop.



```
While loop:
$r_t^i = 0$
$U = U^1 + (m-1)U_{size}$
while ( $CW_t^i > U$ )
    $r_t^i = r_t^i + 1$
    $m = m + 1$
    $U = U^1 + (m-1)U_{size}$
end
```

Figure 2-2 Data flow graph for the SR algorithm

Because the SR algorithm only involves operations such as addition and comparison, it is the fastest algorithm among these standard RAs when implemented in General Purpose Processors (GPPs) without Floating-Point Unit (FPU). However, the SR algorithm includes a *while* loop that has a non-predetermined number of iterations inside a *for* loop. Thus, it is not a priori possible to unfold the loops in the SR algorithm for parallel execution.

The RR algorithm is by far the most complex among these three standard RAs. It is composed of two steps. In the first step, the principal RFs are calculated by truncating the product of the normalized weights and the number of resampled particles. Due to the truncation, the sum of principal RFs may not be equal to the number of resampled particles. In the second step, it is necessary to calculate the residual RFs to compensate the principal RFs and then to guarantee the number of resampled particles. The SR algorithm is often applied in this step [5]. In terms of throughput, the best case for the RR algorithm occurs when all the products of the normalized weights and the number of resampled particles are integers. There is then no need to calculate the residual RFs. However, such case rarely happens. Hong et al. [28] [29] have proposed an efficient fixed-point RR algorithm. They simplified the complex residual RFs calculation using a particle-tagging method to compensate for the number of resampled particles. However, their results are not identical to the results of the original RR algorithm, since it uses a different algorithm in the residual RFs calculation. Also, these authors did not show the effects on the accuracy of the target applications when using this fixed-point RR algorithm. Furthermore, the weight normalization step cannot be merged with the RP step and it is necessary to calculate the normalized weights before using this fixed-point RR algorithm. Finally, their proposed method still requires a step for the selection, replication or removal of particles. This step involves dependencies between iterations. These features limit the speed at which this fixed-point RR algorithm can be computed.

Similarly to the RR algorithm, the RSR algorithm calculates the RFs by truncating the product of the weights and the number of particles. However, in order to eliminate the complex residual processing in the RR algorithm, the RSR algorithm uses a different approach to correct the RFs. From the data flow graph in Fig. 2-3, the RSR algorithm consists of only allowing for a loop with a known number of iterations. Each result of the iteration ΔU is used as the starting point for the next iteration. The RSR algorithm is the least complex among these three standard RAs [5]. Its loop can be unfolded in a hardware implementation. However, the RSR algorithm requires the multiplication and ceiling functions, and there remains a dependency between successive loop iterations. These make the critical path extremely long when executing the loop in parallel. This in turn implies a reduction of the processor clock frequency or the introduction of pipeline stages to maintain throughput.

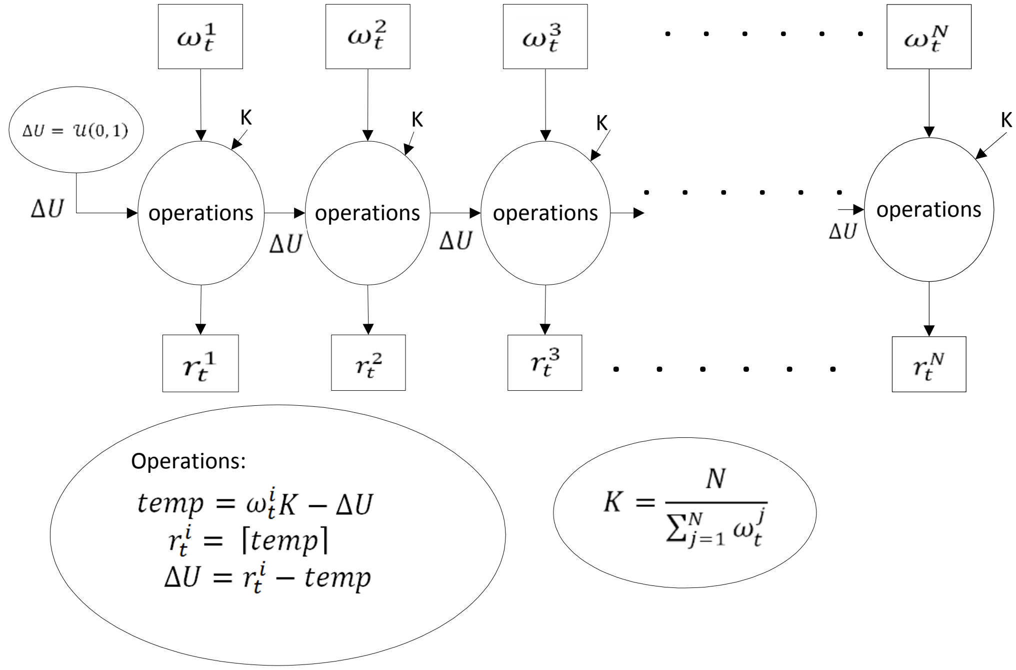


Figure 2-3 Data flow graph for the RSR algorithm

Another approach to implement the resampling processing step is to fuse the RFG, IG and PA blocks to avoid computing the RFs. This is applicable to the SR and compact resampling algorithms. The fusion can reduce the execution time in sequential software implementations, but it further increases the difficulty of parallelizing the RA step because it involves non-predetermined memory accesses in each iteration. Sankaranarayanan et al. [30] and Miao et al. [31] [32] used this approach and the Metropolis-Hastings algorithm to resample the particles. They generate resampled particles which only depend on the current and the previous resampled particles. Although the software implementation is simpler than for other RAs mentioned previously, it is a pipelined RA rather than a parallel RA. Moreover, due to the fact that resampled particles do not correlate with all of the particles, the overall accuracy of this algorithm tends to be lower than that of the SR algorithm [32].

In summary, no existing parallel RA provides a solution that matches the accuracy of the standard RAs. When the resampling step becomes a key bottleneck in implementing PFs in high throughput applications, a parallel RA may be necessary. Furthermore, it is highly desirable that such an algorithm achieve the same accuracy as the standard RAs.

There are many variations of PFs. The Sequential Importance Sampling (SIS) PF proposed in [16] forms the basis PF. It consists of the recursive of prediction generation and weight calculation stages when each measurement is received sequentially. The pseudo-code shown in Fig. 2-4 presents basic steps for the Sampling Importance Resampling (SIR) PF. After the initialization, for each iteration, the algorithm includes five steps: prediction generation, weight calculation, weight normalization, estimation calculation and resampling. Except for the SIS and SIR PFs, many different versions of the PFs have been developed so far. Pitt and Shephard introduced the Auxiliary SIR (ASIR) PF [33]. The basic idea is to perform the resampling step at time $t-1$ using the available measurements at time t before the particles are propagated to time t. This is done by using an extra index for each particle, so the origin of the particle can be traced, while the likelihood at the next time step is evaluated. In this way, the ASIR PF attempts to mimic the sequence of steps carried out when the optimal importance density is available. Kotecha and Djuric [34] proposed a Gaussian Particle Filtering algorithm that is suitable for hardware parallel implementation because it does not require the resampling step. It operates by approximating desired densities as Gaussian. Only the mean and variance of the densities are propagated recursively in time. However, the applicability of the approach is restricted to the models with non-Gaussian noise. Doucet et al. proposed [35] a Rao-Blackwellised particle filtering algorithm which partitions the high-dimension mixture state model into linear and nonlinear models. The Kalman Filters (KFs) and PFs are used for the linear and nonlinear partitions, respectively. This procedure enables a significant decrease in the particle state dimension and reduces the energy dissipation.

```
1) Initialization: Generate $x_0^i \sim p(x_1|\boldsymbol{Z}_0)$ $i = 1 \dots\dots N$
2) Prediction generation: $x_t^i \sim p(x_t|x_{t-1}^i)$
3) Weight calculation: $\omega_t^i = p\left(z_t \middle| x_t^i\right)$
4) Weight normalization: $\widetilde{\omega}_t^i = \frac{\omega_t^i}{\sum_{i=1}^N \omega_t^i}$
5) Estimation calculation: $x_t = \sum_{i=1}^N \widetilde{\omega}_t^i x_t^i$
6) Resampling: $\left\{\{\hat{x}_t^i, \widehat{\omega}_t^i\}_{i=1}^N\right\} = Resampling\left\{\{x_t^i, \omega_t^i\}_{i=1}^N\right\}$.
7) Let $\mathrm{t} = \mathrm{t} + 1$ and repeat from 2.
```

Figure 2-4 Pseudo-code of SIR particle filter

2.2 Applications of Particle Filters

In many important applications of today's high technology, e.g., computer vision, navigation, telecommunication, and the biomedical domain, many challenging problems consist of estimating unknown states from given noisy measurements. These applications are normally based on non-linear and non-Gaussian models. Under these circumstances, PFs can be used in order to obtain robust results of high quality. Hence, PFs are used in many applications. For example, Xu and Li [36] proposed a tracking algorithm based on the Rao-Blackwellised PF and discussed how to use this algorithm in typical surveillance applications. Boucher and Noyer [37] introduced a hybrid PF, which fuses all available pseudo-range measures, applied in global navigation satellite systems applications when the Global Positioning System fails. Kim et al. [38] improved the particle filtering algorithm for the estimation of the number of competing stations in wireless networks. Furthermore, some challenging problems in medical applications such as tracking of protein molecules in the body [39], tracking of instruments for minimally invasive surgery [40] can be solved by using PFs.

In summary, PFs see applications in many different areas. Because these real applications are often related to nonlinear models with non-Gaussian noise, PFs can be used effectively and produce robust results. Due to the different characteristics of each application, in our research, we mainly focus on two types of tracking applications: target tracking and video tracking. We describe these two types of tracking applications in the next two subsections.

2.2.1 Target Tracking

The target tracking problem consists of processing measurements obtained from a sensor in order to maintain an estimate of the target's state [41]. It can be described by state space models, where the state vector of a system contains the position and derivatives of the position. For example, the system state vector could be the kinematic characteristics of the target (i.e., position, velocity, etc.). Several references, e.g. [1] [41] [42] [43] [44], describe the concepts of sensor models, target models, and estimation theory.

In target tracking, common sensors are radar, sonar, and infrared sensors [45]. According to the different characteristics of these sensors, target tracking can be divided into two categories [1]: active target tracking and passive target tracking. Active target tracking exploits both range and bearing measurements. This is the most common type of target tracking in real applications. However, to avoid the risk of being detected by a hostile target, passive target tracking is often used. In a passive tracking mode, only the bearing measurement from a target is available and no explicit range information is measured. Hence, it may be troublesome to compute the position of the target. This is often referred to as the BOT application [1]. The BOT application arises in a variety of important practical applications in surveillance, guidance, or positioning systems [1] [46].

Traditionally, the KF and its variations have been used for target tracking. However, when the system tracks a maneuvering target [47], where the target may have abrupt changes of its state (acceleration) by sudden operation of brake or steering, multiple dynamic models will be adopted to describe this target. These models usually involve strong nonlinearities. In these cases, the KF-based methods will not provide accurate estimations. Instead, the PF-based methods are used in order to yield better performance than the KF-based methods [48].

Gordon et al. [2] simulated a basic nonmaneuvering BOT application by using the SIR PF. The experiments showed that the performance of the SIR PF is greatly superior to the EKF. Kalsson et al. [46] discussed several BOT applications such as air-to-air passive ranging, as well as an air-to-sea application. They used various PFs and the EKF to implement these applications. Their results showed that PFs outperform the EKF. The authors also observed that the EKF is much faster than the PFs.

Gustafsson et al. [6] [49] proposed a framework for target tracking and navigation problems using the PFs based on the motion model and different types of measurements in distance, angle, or velocity. They introduced several applications in practice such as vehicle position, terrain elevation matching, integrated navigation systems and BOT applications. These applications can be easily deployed in the proposed framework. Evaluations of these applications also showed a clear improvement in performance using PFs compared to the existing KF-based methods.

2.2.2 Video Tracking

Video tracking is a crucial component of several applications such as intelligent video surveillance systems [50], animal tracking [51], human gesture recognition [52], human face recognition [53], and tracking sport players on court [54]. The objective of video tracking is to locate one or more objects in time and space in video sequences. Yilmaz et al. [55] presented a survey on video tracking. They discuss the important issues related to video tracking including the selection of proper object models, selection of motion models, and tracking algorithms.

Tracking objects in video can be plagued by several problems, such as object occlusion, irregular and fast object motion, illumination variation, and object deformation. A suitable object model which is employed and integrated into video tracking, can solve the illumination variation and object deformation problems. The object model can include points [56], color histograms [57], edges [58], or histograms of edge-oriented gradients [59]. A robust tracking algorithm is still required to overcome the challenges of object occlusion and irregular and fast object motion.

There are various tracking algorithms to perform video tracking, such as mean-shift tracking [60], the KF [61], and PFs [62]. Mean-shift tracking [60] is a non-parametric density gradient estimator that is iteratively executed within the local search kernels. It is computationally simple. However, if the object relocation between successive frames is larger than the kernel size, the algorithm fails to track the object. The KF is still not a robust tracking algorithm because it is limited to Gaussian and linear models and object models of video tracking are highly nonlinear.

PFs have been popular in video tracking due to their competitive accuracy. Isard and Blake [62] first used PFs for contour tracking, and named their algorithm the CONDENSATION filter. In the CONDENSATION filter, the authors use the SIR PF coupled with an active contour model as the measurement. The CONDENSATION filter performs well in challenging situations such as cluttered backgrounds and in the presence of occlusion.

Several other works followed the CONDENSATION filter to perform video tracking. In order to have a large coverage of the search space with a small number of particles, Isard and Black proposed the ICONDENSATION filter [63]. This filter uses importance sampling [52] instead of the transition prior function, which is used by the CONDENSATION filter. The intent is to avoid, as much as possible, generating particles with low weights. Deutscher et al. [64] also proposed another version of the PF that uses a simulated annealing step in order to propagate samples with the intention of producing particles that have a higher probability of representing the target.

A proper selection of object model is important to achieve good tracking. The color histogram is relatively insensitive to object deformation and it can robustly solve partial occlusion of the object. Thus, the color histogram is widely used as an object model for video tracking in PFs. This idea was initially proposed by Nummiaro et al. [3] and Perez et al. [65]. They both used the Bhattacharyya distance between the current and reference color histograms to calculate the measurement likelihood. Nummiaro et al. [57] extended their work by proposing an adaptive object model where the color histogram of the reference can be updated as the object moves. Perez et al. [66] also proposed another work to combine sound and motion cues into the color-based PFs to increase the robustness of video tracking systems.

In order to simplify PFs for video tracking, P. Dunne and B. Matuszewski [67] examined the dissimilarity distance measures and likelihood functions of PFs in the context of video tracking. Their experimental results suggest that the forms of the likelihood function and distance measure are not critical in video tracking. Their simpler formulation offers more potential computational economy than the CONDENSATION filter.

2.3 PF Implementation in Embedded Systems

2.3.1 Implementation Methods in Embedded Systems

Selecting a suitable implementation method in real-time embedded systems for an algorithm or an application is necessary to meet its requirements. Some of the main requirements are information throughput, power dissipation, accuracy of the results, silicon area, cost, and time to market involved in the embedded implementation. Choices of embedded implementation can be classified into four categories: General-purpose Processors (GPPs), Digital Signal Processors (DSPs), ASIPs and Custom Processors (CPs). In this section, we briefly review these embedded implementation methods.

As their name indicates, GPPs are general in purpose. They are designed for accommodating a wide variety of applications. It has been the mainstream of processor architecture for implementing applications due to their high design flexibility. However, with increasing the complexity of the applications, implementing it in GPPs tends to be power-hungry and time consuming. This situation limits some real-time applications from being efficiently implemented in this programmable solution [68].

DSPs are a special version of GPPs. The major improvement of DSPs over GPPs is that they can accelerate various arithmetic operations such as multiply-and-accumulate. Their performance can be significantly higher over GPPs when implementing applications where the relevant arithmetic operations are the main bottlenecks. However, DSPs are ad-hoc processors optimized for digital signal processing. When the bottlenecks of target applications exist in other domains rather than arithmetic operations such as data-intensive computing, DSPs may not improve their performance when compared to GPPs.

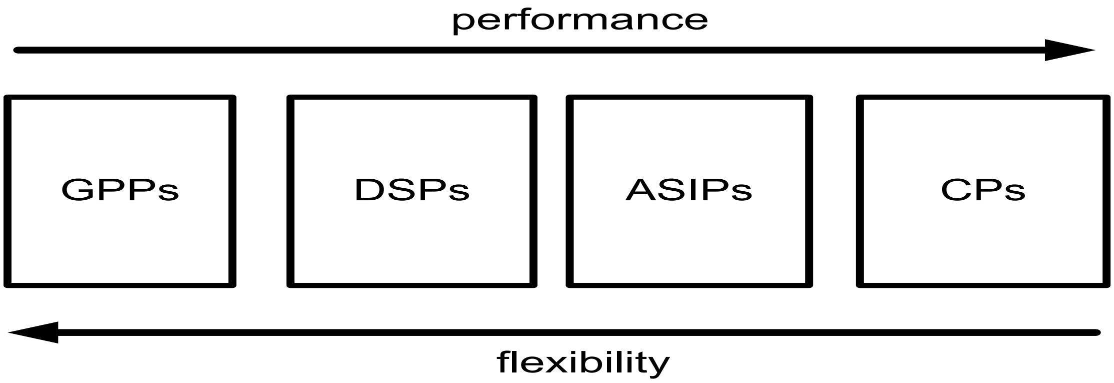


Figure 2-5 Comparing GPPs, DSPs, ASIPs and CPs

The design effort for CPs is normally much larger than for GPPs or DSPs. CPs are therefore usually reserved for applications that require a specific type of calculation not efficiently supported in a GPP or DSP. CPs are often implemented in Field Programmable Gate Array (FPGA) technology where data and communication parallelism can be exploited. Implementing an application in CPs can provides better performance and lower power dissipation when compared to GPPs or DSPs. However, CPs are designed for a single application. They cannot be easily adapted to different applications. For every change in the application specification, a redesign is required [69].

ASIPs fill the gap between GPPs and CPs. They maintain the design flexibility of a programmable solution and overcome the performance limitations of GPPs by including a set of custom instructions optimized for the target applications. Indeed, ASIPs are extension of GPPs with dedicated hardware units generated in accordance with custom instructions and special purpose registers [70]. They can provide a good balance between computation efficiency and design flexibility to meet application requirements such as throughput, time-to-market, design flexibility and power consumption [71].

Fig. 2-5 compares GPPs, DSPs, ASIPs and CPs in terms of performance and flexibility. GPPs or DSPs can provide design flexibility to accommodate any applications. But in order to achieve high throughput for computationally intensive applications, GPPs or DSPs should improve their clock rate. It tends to increase power consumption. On the other hand, the implementation of a specific application in CPs benefits from high performance and energy efficiency but at the expense of design flexibility. When implementing computationally intensive applications for real-time execution, both computational and energy efficiency and design flexibility are important requirements. An alternative, the design of ASIPs for these applications can be considered as a better choice than others.

2.3.2 Examples of PF Implementations

Because PFs are computationally intensive, implementing them in a GPP or DSP may not meet the throughput requirements of tracking applications. For example, Bolic used a TI TMS320C54x DSP to implement a BOT application with the SIR PF [5]. It provides a maximum sampling frequency of 1.8 kHz for 1000 particles. However, most target tracking applications are more complex than the BOT model. In addition, they require a large amount of particles rather than 1000 particles to increase their accuracy. Under these circumstances, the sampling frequency in 1.8 kHz is not high enough to accommodate for most target tracking applications.

In order to obtain high data rate and power efficiency, the dedicated hardware implementation of PFs has been a focus of recent research activity. Several works have reported PF implementation based on this approach [5] [22] [27] [72] [73] [74] [75]. These studies focused on parallelizing the Resampling Processing stage, which is a sequential portion in PFs.

Several works by Bolic et al. [5] [22] [72] [73] [74] implemented the SIR PF [2] for a BOT application in an FPGA prototype. The prediction generation and weight calculation stages in PFs can be easily parallelized. Hence, the authors used several Processing Elements (PEs) and a central unit to build a distributed architecture for the BOT application and then proposed several resampling algorithms such as the RSR algorithm and the partial resampling algorithm to facilitate the resampling algorithm executing in this distributed architecture. Using a Xilinx Virtex II Pro FPGA for implementation, the authors achieved a maximum sampling frequency of 50 kHz. Sadasivam and Hong [76] designed an application specific reconfigurable architecture in an FPGA for the prediction generation and weight calculation stages. This architecture is used in the PE proposed by Bolic [5] to improve the throughput.

El-Halym et al. [27] [75] proposed three different architectures to implement the SIR PF in FPGA: a two-step sequential machine, a pure parallel architecture, and a distributed architecture. They used a piecewise linear function to simplify the exponential function. Their work, like the work by Bolic [5], focused on executing the resampling algorithm in parallel.

Velmurugan et al. [77] [78] presented a mixed-mode implementation of the SIR PF for a BOT application to reduce power dissipation in an Application-Specific Integrated Circuits (ASIC) design. They used an analog Multiple-Input Translinear Element (MITE) network [79] to implement the weight calculation stage. This mixed-mode implementation dissipates approximately 20× less power when compared to a digital implementation.

More recently, one popular way to reduce the computation time is to use a parallel implementation with multiple PEs. This implementation allows design flexibility while improving the throughput. Maskell et al. [80] proposed a Single-Instruction Multiple-Data (SIMD) processor that uses one PE per particle. They proved that the time complexity of the prediction generation and weight calculation stages can be reduced from $O(N)$ to $O(1)$ but the time complexity of the resampling stage is $O((logN)^2)$. Hendeby et al. [81] implemented a range-only application with the SIR PF using a General-Purpose computing on Graphics Processing Unit (GPGPU). The speed increase was significant due to the nature of parallelization in GPGPU and the interpolation method for the exponential operation. Medeiros et al. [82] [83] [84] implemented a color-based PF for video tracking on a SIMD linear processor with 320 PEs. They focused on parallel computation of the particle weights and parallel construction of the intensity histogram because these are the major bottlenecks in standard implementations of color-based PFs for video tracking. Li et al. [85] implemented and optimized PFs for video tracking on a multi-DSP system. They combined the mean shift tracking with PFs to reduce the number of particles.

Most works above can achieve high performance. However, in the context of an embedded system, the implementation should occupy a small area and consume as little power as possible. An architecture with a high volume of PEs can achieve high performance for PFs but at the expense of the silicon area and energy. Hence, it is not suitable in general for embedded applications.

2.4 Application-Specific Instruction-set Processors

Designing and deploying ASICs is becoming increasingly, even prohibitively, expensive with each succeeding generation [69]. While there are several different possible silicon implementation alternatives such as FPGAs or DSPs that may replace ASICs, ASIPs have the potential to provide a programmable solution with high performance. ASIPs aim to generate customizable microprocessors specialized for a given set of applications to meet the desired requirements in performance, cost, and power dissipation.

ASIP design has been studied for several years. In this section, we review the most important works that have been published on the subject. The ASIP design methodologies are presented in Section 2.4.1. We then introduce two basic approaches to automatically generate ASIPs in Section 2.4.2. In Section 2.4.3, the techniques of automatic generation of ASIP specifications are presented.

2.4.1 Overview

Increasing design time and manufacturing costs are constantly pushing the design from ASICs to ASIPs in spite of power and execution time overheads. Keutzer et al. [69] predicted this trend and described the benefits and challenges of ASIPs. The authors followed the MESCAL methodology [86] to identify five key steps in ASIP development, which are disciplined benchmarking, defining the architectural space, efficiently describing the space to be explored, exploring the design space, and exporting the programming environment.

Hoffman and Nohl [71] analyzed wireless communication market trends and demonstrated that ASICs are not the right choice in this market because re-designs in ASICs cause high development costs. The authors pointed out that the ASIP design flow should be driven by a set of target applications. The design successively starts with a successive architecture exploration that involves stepwise refinement of the architecture. The simulation and profiling results are used to design the abstraction levels. The abstraction levels range from the instruction level down to the register transfer level (RTL), Jain et al. [87] surveyed the state of the art in ASIP design methodologies, and also identified five key steps in ASIP design—application analysis, architectural design space exploration, instruction-set generation, code synthesis and hardware synthesis.

Pozzi and Paulin [88] discussed the challenges in ASIP design such as how to reuse custom instructions and how to use multiple ASIPs on chip to achieve high performance, and lower power.

Ienne and Leupers [89] described three steps in the evolution of an ASIP methodology. The first step is migrating from the concept of a general-purpose processor or ASIC to the concept of an ASIP. Keutzer et al. [69], then Gries and Keutzer [86], summarized this concept. The second step is to develop a highly automated, low-risk, and reliable ASIP generation process or methodology driven by a compact and efficient specification. This step will be explained in the next section. The last step is automating the creation of the ASIP specification, based on an automated analysis and abstraction of the underlying application source code. The work on this step will be described in Section 2.4.3.

2.4.2 Processor Generation

The design of ASIPs is a demanding process involving the design of the instruction set, micro-architecture, RTL description, and software tools such as compiler, simulator, assembler, and linker. In the traditional approach, each step of this process requires its own design tools and is often conducted by a separate team of designers. For instance, RTL description requires hardware development tools such as Xlinx EDK used by hardware developers. Application software implementation requires relevant software languages such as C or Matlab. These high-level languages are often used by software developers. Moreover, processor architecture exploration cannot be accomplished without the assistance of all the hardware and software developers. Processor designers must interact with the developers in other departments to obtain an optimal solution. Under these circumstances, design engineers rarely have the time to explore architecture alternatives for the applications. Hence, it is necessary to develop a well-integrated and unified approach to efficiently build ASIPs.

The original approach to build ASIPs is based on Architecture Description Language (ADL) such as LISA [90], EXPRESSION [91], and nML [92], which can model complex processors at a high level of abstraction and automatically generate consistent software development tools and synthesizable HDL code. Ienne and Leupers [89] show the ADL-driven design automation methodology for ASIPs. Various execution tools including a simulator, a retargeted compiler and hardware implementation are automatically generated from the ADL specification. This automation of design reduces the design effort and lets the designers focus on architecture exploration. In this approach, the target processor can be freely defined through adding custom instructions or subtracting unused instructions.

Even with the support of various design automation tools under the ADL model, synthesizing and verifying an ASIP from scratch is difficult. Most researchers use templates or a library of instruction sets to reduce the design space, such as PEAS-III [93] [94]. In PEAS-III, designers only need to specify the architectural parameters, type of resource, instruction formats, and micro-operation descriptions of the target processor, and the datapath and control path are automatically generated from the description as well as a set of application program development tools.

Another approach has emerged to build ASIPs. It is based on the concept of a configurable and extensible processor. Building an ASIP is based on the configurable and extensible processor. The configurable and extensible processor approach covers a limited architectural design space but enables high design efficiency. Designers only need to tune the existing carefully designed architecture and add custom instructions for the specific applications to obtain the optimal ASIP rather than design it from scratch. Consequently, the relative design time can be kept down. Several commercial examples exist such as Tensilica Xtensa, ARCtangent, and Altera Nios II.

Leibson [95] shows an example of an architecture for configurable and extensible processor, the Xtensa LX hardware architecture. The author clearly points out that a small portion of the processor is fixed. Designers are free to select or change the parameters for many configurable options. Designers also can write a large number of custom instructions for the specific application, which are integrated in the pipeline stages of the processor.

2.4.3 Automatic Generation of Custom Instructions

Ienne and Leupers [89] state that the third step in ASIP development is to move from manually generated ASIP specifications to the automatic generation of ASIP specifications. This work mainly involves automatic identification of custom instructions. The custom-instruction identification problem can be divided into the following two phases: generation of a set of custom-instruction templates and selection of the most profitable templates.

Ienne et al. [96] studied the limits of custom instruction extensions on embedded systems. They concluded that custom instructions may achieve reasonable speedups at low cost and write port size from the function units to the register file appears very important. In addition, hardcoding of constants achieves a minor performance gain at the cost of inflexibility. Memory interfaces for custom instructions can be beneficial but not essential. Predication and loop unrolling are fundamental for parallelism to achieve desired results. Bit-width analysis and arithmetic optimization can carry significant intrinsic advantages.

Yu and Mitra [97] also studied the impact of different architectural constraints on the effects of custom instructions. They concluded that relaxing control flow constraints may achieve more performance improvement and a reasonable limit on resources and the number of custom instructions may not affect speedup. Moreover, the pattern of multiple input and single output (MISO) may limit the performance but the speedup achieved by four inputs and three outputs pattern is acceptable. Those studies may provide some guidance for direction in custom instruction generation. However, these conclusions are based on the assumptions above, which may or may not be suitable for a specific situation.

A popular approach to generate custom instructions involves analyzing the data flow graphs (DFGs) of target applications. Atasu et al. published a series of articles [98] [99] [100] [101] [102] describing how to automatically generate custom instructions. The authors introduced constraints on the number of input and output operands for subgraphs and showed this constraint could significantly reduce the search time. Additionally, the authors proposed a greedy algorithm, which iteratively selects non-overlapping DFG subgraphs having maximal speedup potential based on a high-level metric. However, DFGs are often limited to a few hundred nodes and the input/output constraints must be tight enough to reduce the search time. Pozzi et al. [103] optimized the greedy algorithm in Atasu's work [98] and showed that enumerating connected subgraphs only can substantially reduce the speedup potential. Bonzini and Pozzi [104] derived a polynomial bound on the number of feasible subgraphs if the number of inputs and outputs for the subgraphs are fixed. However, the complexity grows exponentially as the input/output constraints are relaxed. CHIPS [101] combined the advantages of the work above and employed integer-linear-programming proposed by Atasu et al. [99] to identify custom instructions. The authors demonstrated that their algorithms are able to handle benchmarks with large basic blocks consisting of more than 1000 instructions, with or without the input/output constraints. Furthermore, Atasu et al. [102] proposed a novel method to enumerate the instruction template that enables fast design space exploration. This simplified method is used to find the optimal instructions via enumeration of maximal convex subgraphs of DFGs.

Chapter 3 EFFICIENT UNIFORM QUANTIZATION LIKELIHOOD EVALUATION FOR PARTICLE FILTERS IN EMBEDDED IMPLEMENTATIONS

In this chapter, we propose a Uniform Quantization Likelihood Evaluation (UQLE) algorithm for Particle Filters (PFs). This algorithm simplifies the Exact Likelihood Evaluation (ELE) algorithm, the most computationally demanding function in PFs, by using a uniform quantization scheme to generate approximated weights. Simulation results indicate that PFs using UQLE can achieve comparable or better accuracy than PFs using ELE. The worst case of UQLE software implementation in fixed-point arithmetic with 32 quantized intervals achieves 23.7× average speedup over the software implementation of ELE. An Application-specific Instruction-set Processor instruction was designed to accelerate the UQLE algorithm in a hardware implementation. The custom instruction implementation of UQLE with 32 intervals achieves 34× average speedup over the worst case of its software implementation on a 79 K general-purpose processor with 5% additional gates.

The material of this chapter was presented in my contribution [7].

3.1 Introduction

PFs [1] [17] [4] [19] are statistical signal processing methods that perform sequential Monte Carlo estimation based on a particle representation of probability densities. Since their introduction in 1993 [2] [13], PFs have gained in popularity to solve non-linear and/or non-Gaussian applications. They have shown great promise as a powerful methodology in addressing a wide range of complex applications including video tracking [62] [11] and navigation [6] [49]. PFs use the concept of importance sampling to recursively compute the relevant probability distributions conditioned on the observations. In comparison with the Extended Kalman Filter (EKF) [61], PFs do not rely on linearization techniques and can robustly approximate the true system state with an appropriate number of particles. In contrast, the EKF sometimes has poor performance, lacks robustness, and may introduce large biases [2].

One of the drawbacks of PFs comes from their significant computational requirements. This feature tends to limit their use in some embedded applications requiring real-time, high-throughput processing. There are two main reasons for the significant computational requirements in PFs. The first reason is that a large number of particles are often required in order to achieve acceptable accuracy. As the number of particles increases, the computational complexity of the PF increases, although this problem can be mitigated through the use of a distributed architecture [105]. The second reason is related to the type of operations involved. In addition to non-linear operations in the DSS model defined by the target applications, traditional PFs, such as Sample-Importance-Resampling (SIR) PFs, may require non-linear operations such as division and exponentiation to calculate the particle weights in the likelihood evaluation step. These expensive and complex operations are often important bottlenecks in embedded implementations of PFs. Simplifying such complex operations is therefore a promising first step in order to improve the PF processing speed and energy efficiency.

PFs are commonly implemented in General-Purpose Processors (GPPs) or Digital Signal Processors (DSPs). Bolic used a TI TMS320C54x DSP to implement SIR PFs as a reference [5]. In that DSP, the exponential operation is approximated by a Taylor series. Implementation in GPPs and DSPs is advantageous from a programmability point of view, but may not satisfy the performance requirements of applications that demand high throughput. A hardware implementation is an appealing solution to this problem. This can include custom processors in Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). Several works have reported PF designs based on this approach [5] [22] [73] [25] [72]. These studies focused on parallelization of the resampling step, which is a sequential portion of the algorithm. Hendeby et al. implemented SIR PFs in a Graphics Processing Unit (GPU) [81]. The speed increase was significant due to the nature of parallelization in the GPU and the interpolation method for the exponential operation, but this solution is not suitable in general for embedded applications.

A hardware implementation can provide higher performance, but at the expense of making the implementation less flexible. A modification to an application's specifications may require significant redesign effort. Consequently, the other class of hardware implementation considered is the Application-Specific Instruction-set Processor (ASIP) [106]. ASIPs aim to strike a balance between GPPs and custom processors by combining a programmable solution with customized hardware units. With this approach, once the bottlenecks of the PFs are found, local acceleration can be applied with customized hardware units to improve the application's overall performance.

In this chapter, we target SIR PFs, where resampling is a necessary step. We analyze PF characteristics and profile them to identify possible bottlenecks for three applications. We specifically consider the likelihood evaluation. Previous work only focused on the exponential operation of the likelihood evaluation step. We broaden the scope of simplification. We also present a customized hardware unit for the proposed simplified algorithm, which can be locally accelerated with an ASIP. The main contributions of this chapter are:

1) A characterization of PFs, which, unlike previous work [5] [6], focuses on distinguishing the application-specific blocks and the algorithm-defined blocks.

2) A demonstration that the likelihood evaluation is a significant and sometimes dominant step affecting PF throughput, and that it is worth optimizing.

3) A novel efficient UQLE algorithm to replace the ELE algorithm.

4) An evaluation of the impact of the approximated weights generated by UQLE under various options on the accuracy of the target applications.

5) An efficient customized instruction for UQLE that achieves significant local acceleration in an ASIP.

The rest of this chapter is organized as follows. In Section 3.2, we analyze the computational properties of the SIR PF and justify the need for UQLE. The proposed UQLE is presented in detail in Section 3.3. In Section 3.4, we evaluate the accuracy of the proposed UQLE in comparison with that of ELE and profile both approaches in a general-purpose processor. In Section 3.5, we evaluate the throughput of UQLE in its software and customized ASIP versions. Section 3.6 concludes this chapter.

3.2 Computational Characteristics of PFs

In this section, we analyze the computational complexity of the SIR PF to justify the need for the simplification of the likelihood evaluation.

3.2.1 Description of DSS Examples

In this chapter, we apply Dynamic State Space (DSS) models to three concrete examples: Linear Gaussian (LG), Uni-variate Non-stationary Growth (UNG), and Bearing-Only Tracking (BOT) models.

Example 1: LG model

Consider the following LG model [19]:

$$x_t = x_{t-1} + u_t, \tag{3.1}$$

$$z_t = x_t + w_t, \tag{3.2}$$

where $u_t \sim \mathcal{N}(0, \sigma_u^2)$ and $w_t \sim \mathcal{N}(0, \sigma_w^2)$, and where σ_u^2 and σ_w^2 are considered fixed and known with variance $\sigma_u^2 = \sigma_w^2 = 1$. The initial state distribution is $x_0 \sim \mathcal{N}(5,1)$. This is the basic model for estimating the problem.

Example 2: UNG model

We consider here a classical UNG model which has been used extensively in the literature for benchmarking numerical filtering techniques [2] [4] [19]. The state space equations are as follows:

$$x_t = \frac{x_{t-1}}{2} + 25\frac{x_{t-1}}{1+x_{t-1}^2} + 8\cos(1.2t) + u_t, \tag{3.3}$$

$$z_t = \frac{x_t^2}{20} + w_t, \tag{3.4}$$

where $u_t \sim \mathcal{N}(0, \sigma_u^2)$ and $w_t \sim \mathcal{N}(0, \sigma_w^2)$, with $\sigma_u^2 = 10$ and $\sigma_w^2 = 1$. The initial state distribution is $x_0 \sim \mathcal{N}(0.1,5)$.

We choose this model because it is highly nonlinear and its observation equation (3.4) introduces a bimodal problem to the estimation. This makes the estimation problem more difficult to solve by traditional methods such as the Kalman filter.

Example 3: BOT model

This example is placed in the context of radar-based target tracking. The BOT model concerns an object moving in the x-y plane (2-D space). The observations taken by the sensor to track the object are in terms of the bearing or angle with respect to the sensor [2] [5].

Let the sensor be stationary and located at the origin in the x-y plane. The object moves according to the following state space model:

$$X_t = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} X_{t-1} + \begin{bmatrix} 0.5 & 0 \\ 1 & 0 \\ 0 & 0.5 \\ 0 & 1 \end{bmatrix} [u_{x_t} \quad u_{y_t}]^T \tag{3.5}$$

$$z_t = \tan^{-1}(\frac{y_t}{x_t}) + w_t \tag{3.6}$$

where $X_t = [x_t \; v_{x_t} \; y_t \; v_{y_t}]^T$, x_t and y_t denote the coordinate position of the target and v_{x_t} and v_{y_t} denote the target velocities in the x and y directions, respectively. The vector $[u_{x_t} \; u_{y_t}]^T$ is composed of white Gaussian noise with standard derivation $\sigma_u = 0.001$. Parameter $w_t \sim \mathcal{N}(0, \sigma_w^2)$ is the observation noise with $\sigma_w = 0.005$. The set of initial states is set to $X_0 = [-0.05, 0.001, \; 0.7, -0.055]^T$ and $\sigma_1 = 0.5, \; \sigma_2 = 0.005, \sigma_3 = 0.3$ and $\sigma_4 = 0.01$, which are the standard derivations of the noise for the initial state.

From Equation (3.6), we see that no range information is available to the sensor. Thus, only the angle of the object movement but not its distance from the point of sensor can be detected with a series of observations. The observation consists of a modal ridge along the line $y_t = \tan(z_t)x_t$. Hence, the BOT model is the multimodal case. The estimate of the object trajectory is difficult to solve because it only depends on this multimodal measurement and the prior information, which is the position and the velocity of the object at the initial stage.

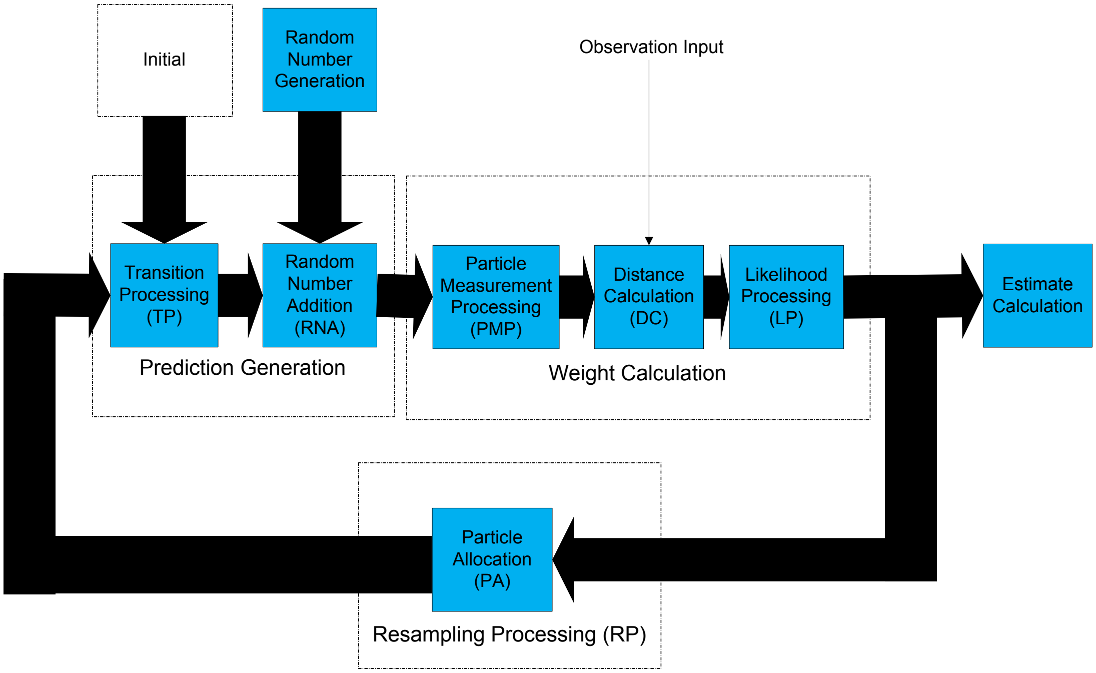

Initial
Random Number Generation
Observation Input
Transition Processing (TP)
Random Number Addition (RNA)
Prediction Generation
Particle Measurement Processing (PMP)
Distance Calculation (DC)
Likelihood Processing (LP)
Weight Calculation
Estimate Calculation
Particle Allocation (PA)
Resampling Processing (RP)

Figure 3-1 Functional view of the SIR PF

3.2.2 Computational Complexity of SIR PFs

3.2.2.1 Functional View of SIR PFs

The functional blocks of SIR PFs with merged steps (weight normalization, resampling, and estimation) are shown in Fig. 3-1. For each observed input, the SIR PFs sequentially perform three steps: Prediction Generation (PG), Weight Calculation (WC) and Resampling Processing (RP). The PG step is decomposed into two blocks, and the WC step is decomposed into three blocks. The RP step is made up of a single block. The three main steps, decomposed into a total of six blocks, are in the critical path. Thus, all these blocks are potential optimization targets when a high performance implementation is required. The complexity of SIR PFs can be partitioned into the six considered blocks: Transition Processing (TP), Random Number Addition (RNA), Particle Measurement Processing (PMP), Distance Calculation (DC), Likelihood Evaluation (LE) and Resampling Algorithm (RA). The TP and PMP blocks are application-specific blocks and their respective complexity depends on the application. The other blocks are algorithm-defined. Their complexity is closely related to the selected algorithms. For example, the normal or exponential likelihood evaluation can be used in the LE block. The choice of the algorithms depends on the type of operations that must be used to obtain a suitable accuracy, a suitable complexity, or an appropriate trade-off between them.

There are two additional significant blocks in Fig. 3-1. One calculates the estimates for the filter output. The other generates random numbers used for updating the states in the prediction generation step. Random number generation can consume a substantial part of the overall processing time. But due to the fact that it is not in the critical path, a random number generator implemented as a co-processor can be employed to operate in parallel and not affect the speed of PFs. Similarly, the output block is not in the critical loop.

3.2.2.2 Computational Complexity

The computational complexity of SIR PFs depends on the complexity and dimensionality of the underlying DSS model. Indeed, that model is updated in the TP and PMP blocks. Most of the computational effort of the PFs may be spent in these two blocks if the DSS model is complex or has a large number of dimensions. From Table 3.1, we can see that the TP block in the UNG model and the PMP block in the BOT model require 51.5% and 45.4% of the overall execution time in a GPP without FPU, respectively. It is in general not possible to analyze and determine the computational property of a DSS model without knowing the specific application. In contrast, with the LG model, the TP and PMP blocks require only about 0.1% of the whole execution time. In this chapter, we focus on the analysis of the generic blocks that compose the particle filtering algorithm. As the TP and PMP blocks are application-specific blocks, we therefore concentrate on analyzing the other blocks that are in the critical path: RNA, DC, LE, and RA.

Table 3.1 and Table 3.2 present profiling results for an Xtensa LX2 processor [107] with and without FPU, respectively. The tables include average cycle counts and percentage of total execution time consumed by each block for the LG, UNG, and BOT models, after 50 runs. The Xtensa LX2 processor is a general-purpose processor with an optional FPU and it also supports optional customizable instructions that can be added to improve performance.

From Table 3.1, we observe that the LE block always take a significant portion of the total number of clock cycles (90%, 41%, and 47% for the LG, UNG, and BOT models, respectively).

Table 3.2 shows that when a FPU is employed, the LE block consumes a larger fraction of the total execution time, and the proportions increase to 98%, 43%, and 49% for the LG, UNG, and BOT models, respectively. This is remarkable and may appear to be somewhat counterintuitive. Indeed, it was found that the FPU provided by the Xtensa LX2 processor does not support the complex operations (exponentiation, division, conditional branch, and so on) that the LE block often uses. The additional hardware complexity introduced by the FPU option is not very useful for the LE block, which is either dominant or very significant, as it does not improve much its performance. According to the results in Table 3.1 and 3.2, when a GPP is used, whether a FPU is present or not, any attempt at optimizing SIR PFs must consider the LE block.

Table 3.1 Average clock cycles of LG, UNG, and BOT models using SIR PFs with 512 particles in the Xtensa LX2 processor without FPU

	LG Model		UNG model		BOT model	
	Cycles (K)	%	Cycles (K)	%	Cycles (K)	%
Transition Processing (TP)	2.01	0.11	2045.86	51.53	34.98	0.90
Random Number Addition (RNA)	18.38	1.04	18.18	0.46	63.96	1.64
Particle Measurement Processing (PMP)	2.01	0.11	111.25	2.80	1769.81	45.41
Distance Calculation (DC)	56.40	3.19	66.38	1.67	112.04	2.88
Likelihood Evaluation (LE)	1593.61	90.01	1639.41	41.29	1827.19	46.88
Resampling Algorithm (RA)	98.17	5.54	89.45	2.25	89.27	2.29
Total	1770.56	100	3970.53	100	3897.25	100

Table 3.2 Average cycles of LG, UNG, and BOT models using SIR PFs with 512 particles in the Xtensa LX2 processor with FPU

	LG Model		UNG model		BOT model	
	Cycles (K)	%	Cycles (K)	%	Cycles (K)	%
Transition Processing (TP)	2.01	0.13	1996.68	53.68	7.02	0.20
Random Number Addition (RNA)	5.52	0.35	5.52	0.15	10.04	0.28
Particle Measurement Processing (PMP)	2.01	0.13	83.24	2.24	1767.56	49.77
Distance Calculation (DC)	5.52	0.35	5.52	0.15	5.52	0.16
Likelihood Evaluation (LE)	1552.50	97.55	1604.82	43.14	1734.22	48.85
Resampling Algorithm (RA)	23.72	1.49	23.77	0.64	26.09	0.74
Total	1591.27	100	3719.55	100	3550.45	100

One popular way to reduce the computation time is to use a parallel implementation with multiple processing elements (PEs) or multi-cores. The execution time of all PF blocks, except the RA block, can be reduced significantly, because the computations are independent for each particle. In principle, the execution time in the LE block can be reduced by a factor equal to the number of PEs or cores. Due to its sequential nature, the RA block that performs systematic resampling (SR) [2], may become the most time consuming block when many PEs are employed. However, various efforts [5] [22] [73] were made to derive distributed resampling algorithms/architectures that allow the RA block to exploit the parallelism. The execution time of the RA block can also be reduced. When such distributed resampling algorithms/architectures are used, the LE block becomes the most computationally expensive part again. This is a further motivation to focus on optimizing the LE block, irrespective of whether PFs are executed in a single GPP or using a parallel implementation with multiple PEs.

3.3 Proposed UQLE Algorithm

Most likelihood evaluation algorithms used in PFs are based on the class of exponential family of distributions, which includes the normal, exponential, and gamma distributions. These distributions require the calculation of the exponentiation, division, multiplication and other operations. These operations make the likelihood evaluation become the most computationally intensive part in PFs. A simplified likelihood evaluation algorithm is one way to improve the particle filtering speed.

The proposed UQLE algorithm is based on the assumption that approximated weight values do not significantly affect the accuracy of the target application. Under this assumption, UQLE can employ a uniform quantization method to enable the use of a simple approximate representation for some quantity of the output value, the particle weights.

In SIR PFs, the particle weights can be given by the likelihood evaluation:

$$\omega_t^i = \exp(\frac{-d_i^2}{2\sigma^2}), \tag{3.7}$$

where d_i, the input to the likelihood evaluation, is the i$^{\text{th}}$ particle distance between the observation and the i^{th} particle result in the PMP block, and σ^2 is the variance of the observation. The proposed UQLE takes advantage of the fact that the particle weights are in the range [0, 1]. This range can be divided into M intervals of length Q, where Q is the quantization step-size:

$$Q = \frac{1}{M}. \tag{3.8}$$

Based on this partition, the range of the weights consists of M intervals: $\{[0,Q),$ $[Q,2Q),[2Q,3Q),\ldots,[(M-2)Q,\ (M-1)Q),[(M-1)Q,1]\}$. Inside each interval, we can choose the high value, the middle value, or the low value to represent the quantized value as the approximated weight W_m. They are defined as:

$$W_m = \begin{cases} mQ, & low\ value \\ mQ + \frac{Q}{2}, & middle\ value \\ (m+1)Q, & high\ value \end{cases}, \tag{3.9}$$

where $m = 0,1,\ldots,M-1$, is the index for each of the intervals. With this approach, once the index for each of the intervals is determined according to the particle distance d_i, the approximated weights W_m are given by equation (3.9).

In accordance with the M intervals of the particle weights, the data range of the particle distance can also be divided into M intervals: $\{[T_1,+\infty),[T_2,T_1),[T_3,T_2),\ldots,[T_{M-1},T_{M-2}),$ $[0,T_{M-1}]\}$, where $T_m, m = 0,1,\ldots,M-1$ is a boundary of each interval in the particle distance direction. T_m can be obtained through the inverse likelihood evaluation. The inverse normal distribution likelihood evaluation is given by:

$$T_m = \sqrt{-2\ln(mQ)\sigma^2}. \tag{3.10}$$

When the particle distance d_i is available, the index of the interval m can be determined from the interval of the distance range in which this d_i falls.

When there are significant deviations from the most recent estimate, the value of all the weights can be small, and the distances can all fall in the $[T_1,+\infty)$ range. This result is due to the fact that all weights are equal to zero when we use the low value of the intervals as the approximated weight. In order to avoid this situation, we insert an additional interval $[0,\delta), 0 < \delta \ll Q$ into $[0,Q)$. The approximated weight at the index $m = 0$ is then modified as follows:

$$W_0 = \begin{cases} \delta, & d_i \in [T_\delta, +\infty) \\ 2\delta\,, & low\ value \\ \frac{Q}{2}, & middle\ value \\ Q, & high\ value \end{cases}, \tag{3.11}$$

where T_δ is calculated by equation (3.10) with the input δ . There are therefore $M + 1$ intevals.

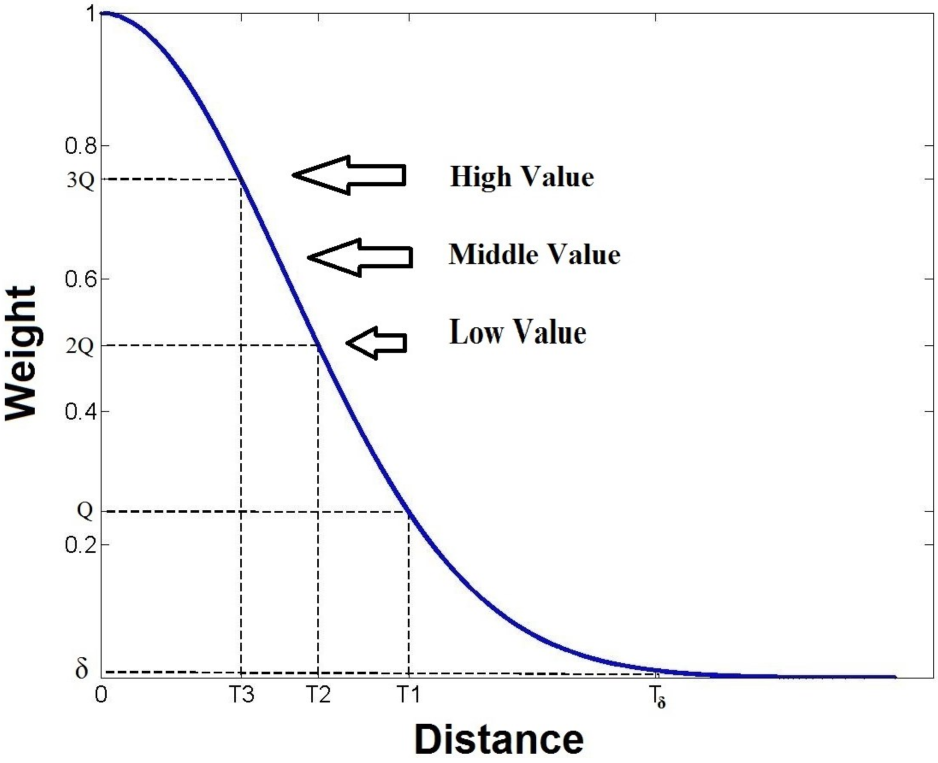


Figure 3-2 Likelihood evaluation curve for the proposed UQLE with *M+1= 5* intervals

Fig. 3-2 shows an example of the likelihood evaluation curve for the proposed UQLE. The range of particle weights is divided into $M + 1 = 5$ intervals: $\{[0, \delta), [\delta, Q), [Q, 2Q),\ [2Q, 3Q), [3Q, 1]\}$. Because $M = 4$, we have $Q = \frac{1}{M} = 0.25$. Then, T_δ, T_1, T_2, T_3 can be calculated by the inverse likelihood evaluation defined by equation (3.10). The index m can be determined via the comparison between the particle distance and T_δ, T_1, T_2, T_3. For instance, for a particle distance $d_i \in [T_3, T_2)$, we can obtain its index $m = 2$, as shown in Fig. 3-2. The weight for this particle distance can be approximated by equations (3.9) and (3.11), which in this case is equal to W_2.

The pseudo-code for UQLE is given in Fig. 3-3. The whole algorithm is divided into two parts: a pre-calculation part and an on-line execution part. After defining the number of intervals and the weight value, which is a low, middle, or high value as the representation, the pre-calculation part provides the boundary in the distance direction $\mathrm{T_m}$ and the approximated weights for each interval according to equations (3.9) and (3.11). These results are used for the on-line execution part later. In the on-line execution part, the approximated weights can be obtained by comparing the input, i.e. the particle distance, with the boundary in the distance direction $\mathrm{T_m}$. Using the proposed procedure, the execution time is reduced significantly because we obtain the weights from comparisons instead of complex arithmetic for exact likelihood evaluation.

```
Pre-calculation part
    Define $\delta$, $0 < \delta \ll Q$
    $T_\delta = sqrt(-2ln(\delta)\sigma^2)$
    $T_m = sqrt(-2ln(mQ)\sigma^2), m = 0,1,2,\ldots,M-1$
On-line execution part
$\omega_t^i = likelihood(d_t^i)$
    If $(d_t^i \geq T_\delta)$
        $\omega_t^i = W_0$
    else
        Find $m$ using linear search, the index of the intervals, such that $T_m > d_t^i \geq T_{m+1}$,
        $\omega_t^i = W_m$
    end
end
```

Figure 3-3 Pseudo-code of the proposed UQLE algorithm

3.4 UQLE Accuracy Evaluation

3.4.1 Implementation Environment and Accuracy Metrics

In this section, we simulate SIR PFs using UQLE for the LG, UNG, and BOT models in the MATLAB environment. According to the UQLE algorithm, the particle distance must be compared with pre-calculation boundaries T_m. We can then find the index of the intervals m and obtain the particle weight directly from the pre-calculation weight W_m. The input and output do not have any numerical calculation relationships. It is not necessary to set all the data in UQLE in floating-point arithmetic. In order to be consistent with the implementation in the Xtensa processor, we implemented UQLE with floating-point arithmetic for the input, the particle distance, and fixed-point arithmetic with 16-bit representation for the output, the particle weight. Thus, we set the range [0, 65535] for the particle weight and δ equal to 1. Under this situation, we compare the resultant accuracy to SIR PFs using ELE. For the LG and UNG models, the accuracy is measured by the Root-Mean-Square Error (RMSE):

$$RMSE = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(x_t - \hat{x}_t)^2} \tag{3.12}$$

where x_t is the true simulated state and $\hat{x}_t$ is the estimated state. For the BOT model, in order to obtain a better illustration for the performance, the accuracy is measured by the combined Mean-Squared-Error (MSE) (where MSE = RMSE^2) of the x and y positions:

$$MSE_{xy} = \sqrt{(MSE_X)^2 + (MSE_Y)^2} \tag{3.13}$$

where MSE_X and MSE_Y are the MSE for the x and y positions, respectively. The smaller the value of the combined MSE is, the better the performance obtained. Furthermore, simulation for a lost track situation is conducted for the BOT model, as suggested in [5]. The track is considered lost if all particles have zero weight in the ELE or if all the particle weights are equal to δ in the UQLE.

In order to obtain stable performance results, 10000 simulations were performed with 50 observation inputs for the LG and UNG models and 25 observations for BOT model.

3.4.2 Accuracy for the LG Model

Table 3.3 shows maximum, average and minimum RMSE results for SIR PFs using ELE, for 512, 1024, 2048, and 4096 particles. We use these results as the reference to compare the results generated by UQLE.

Table 3.3 RMSE of the SIR PF using ELE for the LG model

# of particles	512	1024	2048	4096
Maximum RMSE	1.030	1.053	1.037	1.052
Average RMSE	0.785	0.784	0.783	0.783
Minimum RMSE	0.559	0.545	0.567	0.472

In Fig. 3-4, maximum, average and minimum RMSE results of SIR PFs using UQLE are plotted. The white bar gives RMSE results of SIR PFs using ELE as summarized in Table 3.3. From Fig. 3-4, we can see that SIR PFs using UQLE can produce average RMSE performance close or equivalent to SIR PFs using ELE when *M*, the number of intervals, exceeds 16. Hence, UQLE can replace ELE in SIR PFs in the LG model. It can also be observed that with the number of particles *N* increasing, UQLE with $M = 8$ can achieve equivalent accuracy to ELE. Decreasing the number of intervals implies reducing the computation time. UQLE outperforms ELE in speed without sacrificing the accuracy. Concerning the choice between the low, middle or high value, UQLE with the low value is slightly better than other choices.

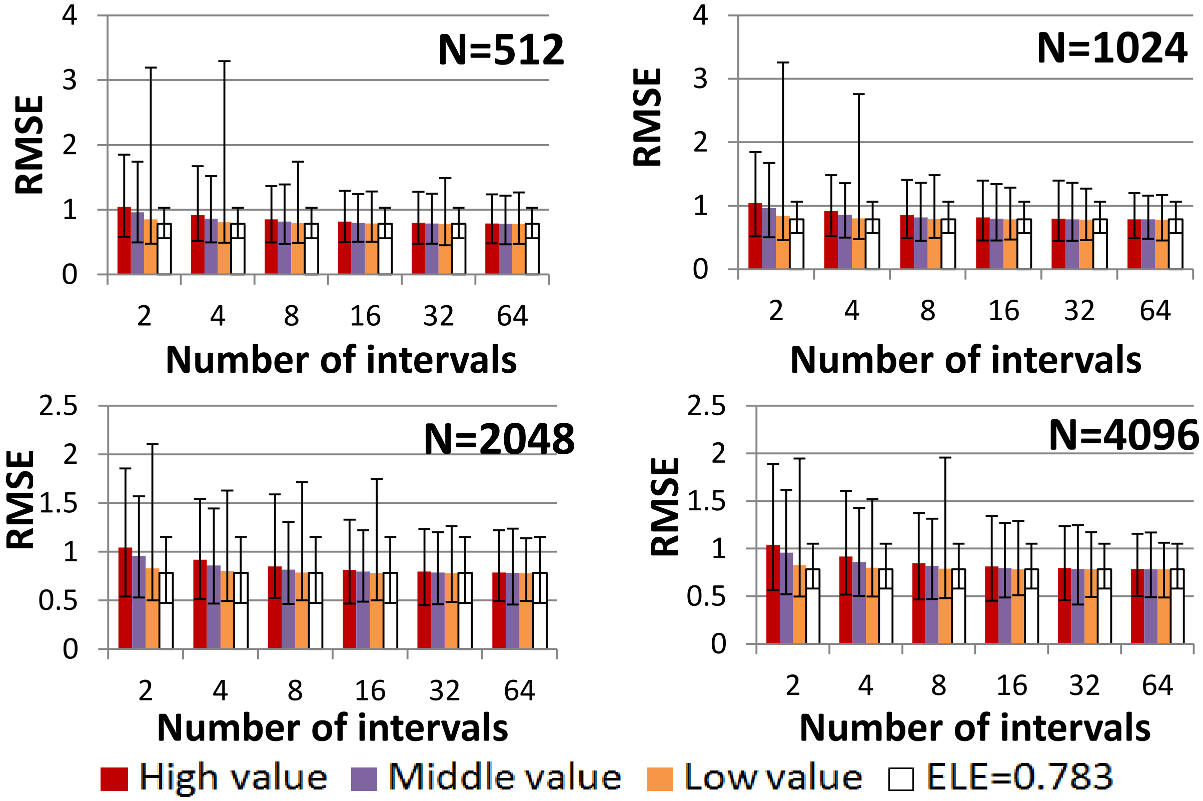


Figure 3-4 Maximum, average and minimum RMSE results of SIR PFs using UQLE for the LG model

3.4.3 Accuracy for the UNG Model

Fig. 3-5 compares the maximum, average and minimum RMSE results for 512, 1024, 2048 and 4096 particles for the UNG model. The dashed line displays the reference accuracy of SIR

PFs using ELE, with the values summarized in Table 3.4. Fig. 3-5 shows that UQLE achieves an accuracy similar to ELE for a number of intervals ($M = 16$) when the low or middle values are used. In addition, the accuracy of UQLE in the UNG model is similar to that in the LG model, with UQLE using fewer intervals and achieving equivalent accuracy to ELE when the number of particle increases.

Table 3.4 RMSE of SIR PFs with ELE for the UNG model.

# of particles	512	1024	2048	4096
Maximum RMSE	5.098	5.471	5.139	5.100
Average RMSE	3.78	3.76	3.76	3.76
Minimum RMSE	2.528	2.516	2.554	2.549

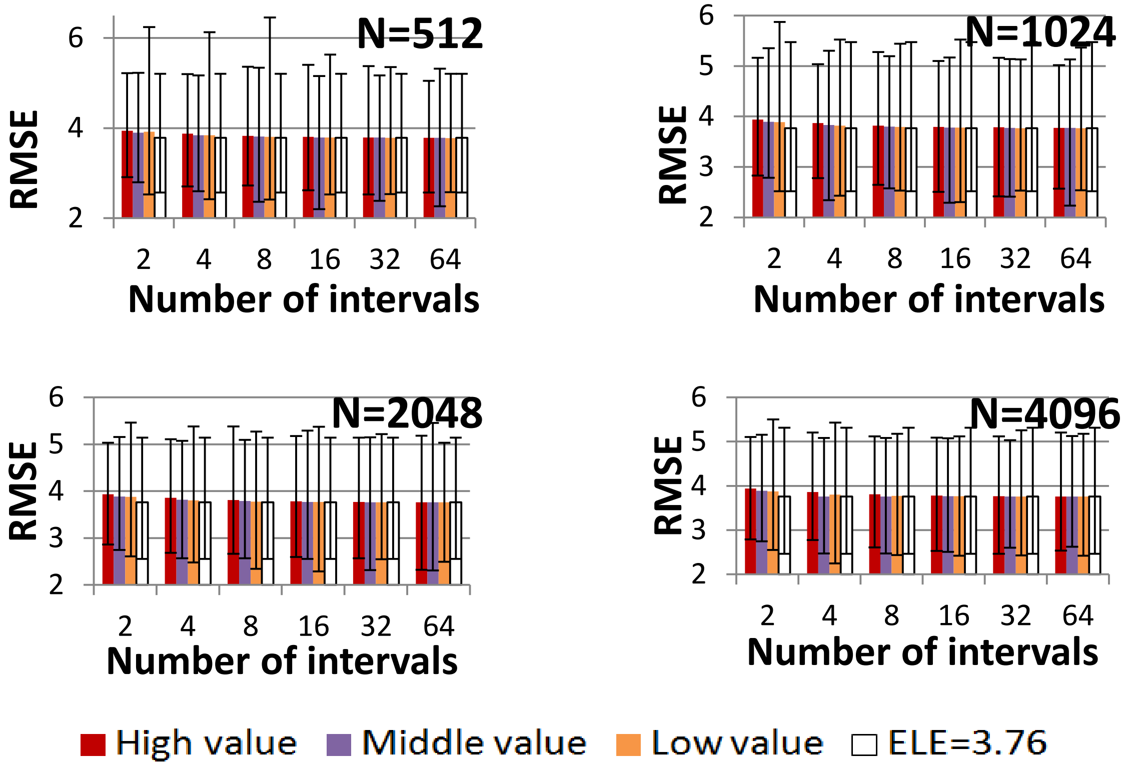


Figure 3-5 Maximum, average, and minimum RMSE results of SIR PFs using UQLE for the UNG model

3.4.4 Accuracy for the BOT Model

In Table 3.5, we show the maximum, average, and minimum combined MSE and the number of times when the track is lost versus the number of particles for SIR PFs using ELE. From Table 3.5, with the increase of the number of particles, the average combined MSE decreases from 0.21 for 512 particles to about 0.14 for 4096 particles. Meanwhile, the number of lost tracks drops from about 8052 for 512 particles to below 2700 for 4096 particles. It should be noted that in ELE we assume that the weights are clamped to $\delta = 2^{-16}$ when their value is smaller than 2^{-16}, which is the same as for UQLE, because the lost track situation significantly depends on the smallest weights considered.

Table 3.5 Combined MSE and Lost track of SIR PFs with ELE for the BOT model

# of particles	512	1024	4096
Maximum Combined MSE	1.22	0.98	0.78
Average Combined MSE	0.21	0.17	0.14
Minimum Combined MSE	0.01	0.01	0.02
Lost track	8052	6824	2637

Fig. 3-6 shows the maximum, average, and minimum combined MSE and the lost track situation for SIR PFs using UQLE. The dashed lines are the corresponding results obtained by SIR PFs using ELE. From Fig. 3-6, although UQLE employing the low value of the intervals has poor performance in our experiments, UQLE with the high value or the middle value of intervals significantly outperforms SIR PFs using ELE, even for the smallest number of intervals $M = 2$ in our experiments. This means that the speed using the proposed UQLE can increase significantly in the BOT model because each particle only needs to be compared with two boundaries in the UQLE algorithm. Fig. 3-7 displays a representative trajectory and the tracking obtained by SIR PFs using ELE and SIR PFs using UQLE with 2 intervals and high value. It shows again that UQLE with few intervals can replace ELE in the BOT model.

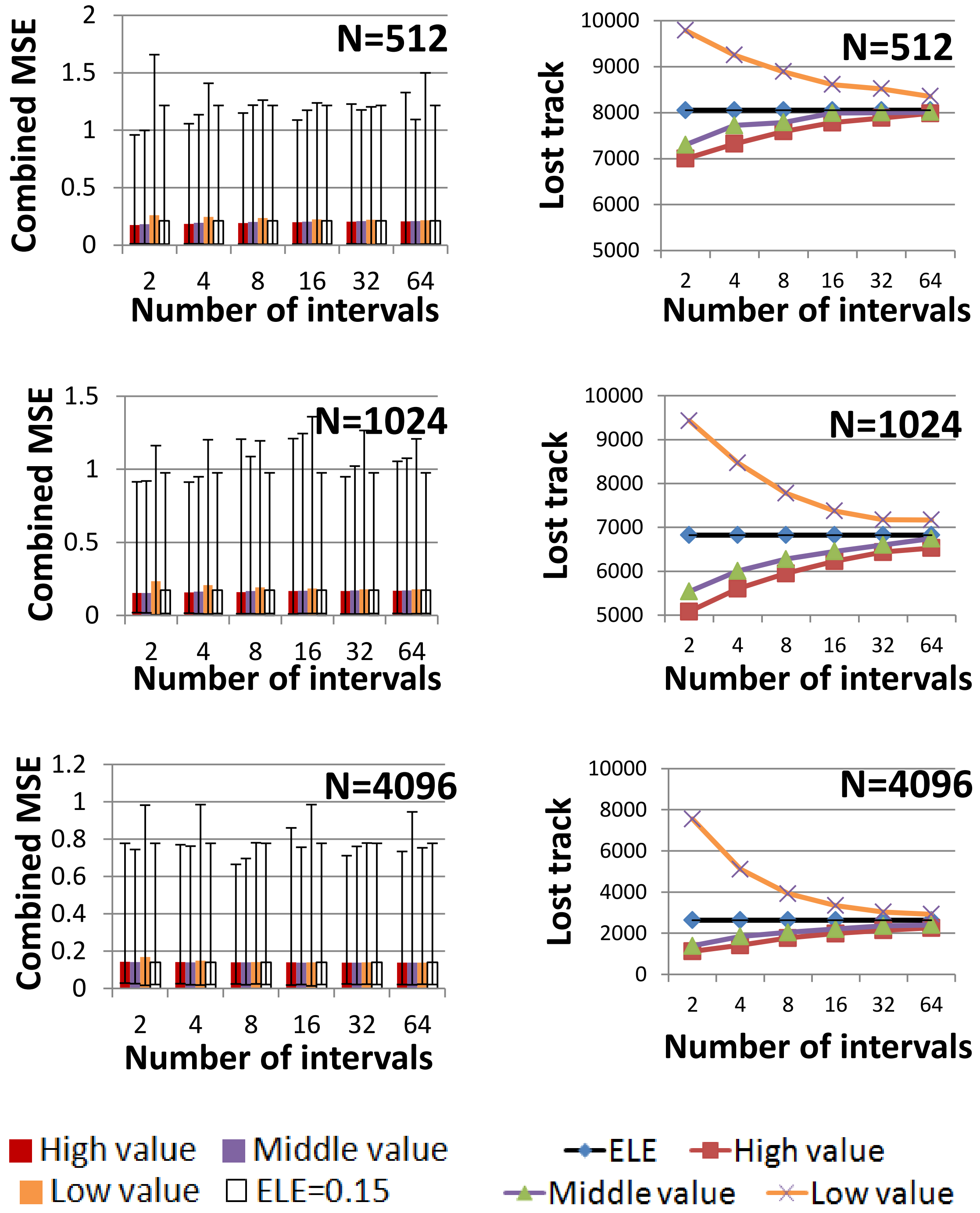


Figure 3-6 Maximum, average and minimum combined MSE and lost track for SIR PFs using UQLE

In summary, simulation results show that UQLE can achieve equivalent or better accuracy than ELE for SIR PFs when the number of intervals and the representation of the output value are suitably chosen. In our experiments, in order to obtain comparable accuracy to ELE, the minimum number of intervals is *M = 16, 32* and *2* for the LG, UNG, and BOT models, respectively.

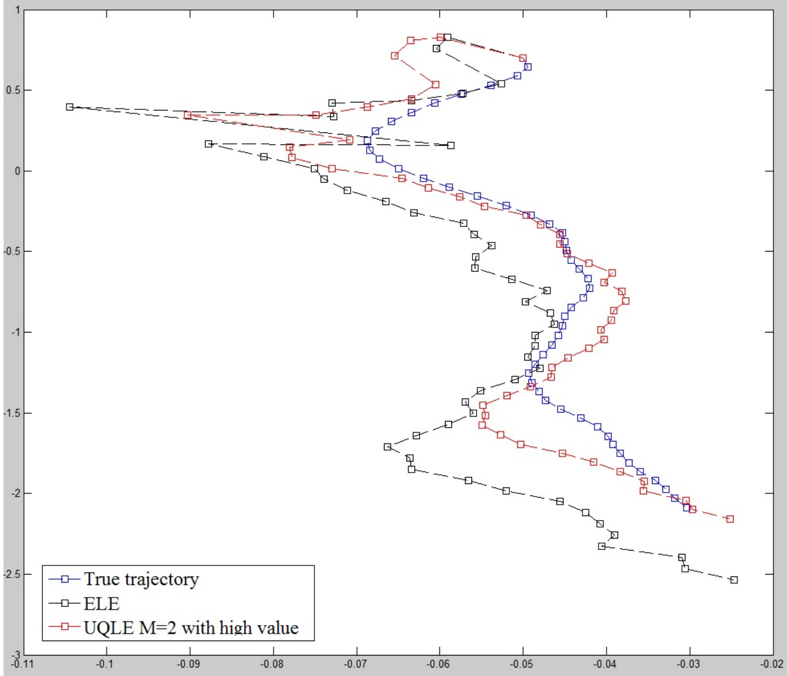


Figure 3-7 Tracking of a moving target in two dimensions with SIR PFs using ELE and SIR PFs using UQLE with 2 intervals and high value.

3.5 UQLE Throughput Evaluations

3.5.1 UQLE Software Implementation and its Performance

We implemented the proposed UQLE in the Xtensa LX2 processor in order to compare its performance to that of ELE. We used the same configuration simulated in the MATLAB environment, with the particle distances expressed in floating point representation and the particle weights expressed in fixed point with 16 bits. In order to further reduce the execution time, we also evaluated the particle distance in fixed-point arithmetic.

Table 3.6 Execution time and speedup of UQLE for the worst and best case in the Xtensa LX2 processor

Likelihood Evaluation (LE)	Particle Distance Representation	M	Case	Execution time (cycle counts)	Average speedup
ELE	Floating-point			3221	1×
UQLE	Floating-point	64	Worst	1112	2.9×
		32		568	5.7×
		16		296	10.9×
		4		92	35.0×
		2		58	55.5×
		Any	Best	40	80.5×
	Fixed-point	64	Worst	274	11.8×
		32		136	23.7×
		16		38	84.8×
		4		14	230.1×
		2		11	292.8×
		Any	Best	9	357.9×

Table 3.6 shows the speedup for UQLE with the particle distance expressed with floating-point and fixed-point accuracy when compared to ELE. The average execution time for the ELE in Table 3.6 is chosen as a baseline based on 10000 simulations since the execution time of calculating ELE in the floating-point arithmetic on the Xtensa LX2 processor is not fixed. The best case for UQLE using any number of intervals achieves 80.5× and 357.9× average speedup over ELE when executed in floating-point and fixed-point arithmetic for the particle distance, respectively. The worst case for floating-point UQLE using 64 intervals still achieves 2.9× average speedup over ELE. The speedup is greater when the number of intervals for UQLE is smaller than 64 or the weights are calculated by UQLE in the fixed-point arithmetic. UQLE with 2 intervals in fixed-point arithmetic can achieve 292.8× average speedup over the software implementation of ELE on an Xtensa LX2 processor.

As shown in Fig. 3-4, 3-5 and 3-6, in order to achieve the same accuracy as ELE, the UQLE requires at least 16, 32 and 2 intervals for the LG, UNG and BOT models, respectively. Table 3.7 shows the average execution time for ELE and UQLE in the LG, UNG and BOT models. UQLE for the UNG model is therefore the most computationally demanding of the three. Still, its fixed point implementation achieves 53.4× average speedup over the software implementation of ELE.

Table 3.7 Average execution time and speedup of UQLE for the LG, UNG and BOT models in the Xtensa LX2 processor

Model	Likelihood Evaluation (LE)	Particle Distance Representation	M	Average execution time (cycle counts)	Average speedup
LG Model	ELE	Floating point	-	3113	1.0×
	UQLE	Floating point	32	327	9.5×
			16	197	15.8×
		Fixed point	32	91	34.2×
			16	28	111.2×
UNG Model	ELE	Floating point	-	3202	1.0×
	UQLE	Floating point	64	314	10.2×
			32	183	17.5×
		Fixed point	64	111	28.8×
			32	60	53.4×
BOT Model	ELE	Floating point	-	3568	1.0×
	UQLE	Floating point	4	60	59.5×
			2	54	66.1×
		Fixed point	4	11	324.4×
			2	10	356.8×

3.5.2 UQLE ASIP Implementation and its Performance

We designed a custom instruction for UQLE to improve the speed of SIR PFs in ASIPs. As shown in Fig. 3-8, the UQLE algorithm has one input d_t^i and one output ω_t^i. This situation is favorable for the generation of a custom instruction because there are no additional input/output data that could create a bottleneck preventing effective acceleration.

The proposed logic organization to support the custom instruction is shown in Fig. 3-8. In a typical configuration of that instruction, 4 quantization intervals are supported and the output is the high value of the intervals expressed with 8 bits. The critical path consists of a comparator, an

XOR gate, an encoder, and a multiplexer. When a particle distance is available, this value is compared with distance boundaries $\{T_0, T_1, T_2, T_3\}$, which are stored in the state registers. For example, if $d \in [T_1, T_2]$, which means $d < T_0, d < T_1$, $d > T_2$ and $d > T_3$, the outputs of the comparators are {0,0,1,1}. Through the XOR gates, the encoder input is changed to {0,1,0,0}. At the output of the encoder, the interval index is equal to 01, which means d is in the second interval. The weight can be set to {01111111}, which corresponds to the high value of the second interval. The number of intervals supported is a parameter of the architecture that can easily be changed as a function of the specific requirements. We implemented the instruction for five numbers of intervals (4, 8, 16, 32 and 64) and three choices of W_m (low, middle and high value of the quantization intervals).

The custom instruction was described in the Tensilica Instruction Extension (TIE) language [108] to generate a specific functional unit in the Xtensa LX2 processor. The special instruction for UQLE is:

$$\omega_t^i = likelihood(d_t^i). \tag{3.14}$$

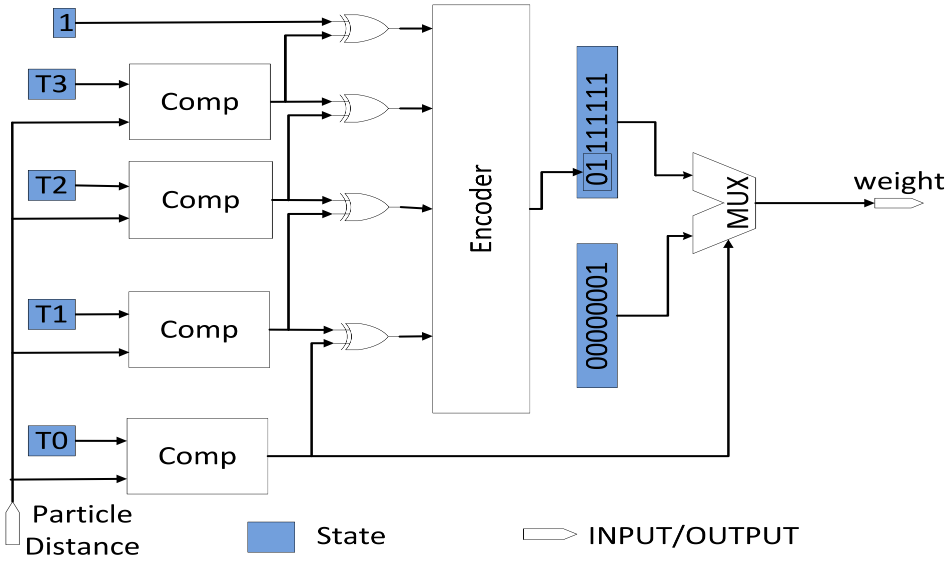


Figure 3-8 Custom UQLE logic organization with 4 intervals

Profiling results are shown in Table 3.8. With the custom instruction, the speedup for UQLE using 64, 32, and 4 intervals can reach up to 278×, 142×, and 23× over their floating-point software implementation in the worst case, respectively. In addition, the cycle count for UQLE is independent of the number of intervals. Increasing the number of intervals only increases the processor size. The additional size for UQLE with 4, 32, 64 intervals, respectively only occupies 0.7%, 4.7%, 8.4% of the reference processor that consumes 79K gates.

Table 3.8 Execution time and additional area of UQLE using its custom instruction in PFs in the Xtensa LX2 processor

M	Implementation method	Case	Particle distance representation	Execution time (clock cycles)	Speedup	Additional area (gates)
64	Software	Worst	Floating-point	1112	1×	-
			Fixed-point	274	4.06×	
	ASIP	-	Fixed-point	4	278×	6.60 K
32	Software	Worst	Floating-point	568	1×	-
			Fixed-point	136	4.18×	
	ASIP	-	Fixed-point	4	142×	3.75 K
4	Software	Worst	Floating-point	92	1×	-
			Fixed-point	14	6.57×	
	ASIP	-	Fixed-point	2	23×	0.57 K

From Table 3.8, the UQLE instruction can significantly reduce the execution time to only 4 cycles. For the LG model, the likelihood evaluation dominates the execution time of the whole application. It is significant that using only 3.75K additional gates for UQLE instruction with 32 quantization intervals can achieve almost 10× average speedup as shown in Table 3.9. These additional gates only occupy 4.7% of the reference processor. If a processor with multiple PEs is used to implement the LG model, at least 10 PEs are required to achieve 10× speedup. But the number of additional gates to build 10 PEs is much larger than the number of gates that the UQLE instruction uses. Hence, for those applications using PFs where the DSS model is not complex, using UQLE is a powerful first step to reduce the execution time and energy consumption.

For applications using PFs like the UNG and BOT models, where the DSS model is rather complex, simplifying the algorithm or using hardware implementation must be shifted to the DSS model. Finding the bottleneck in Fig. 3-1 requires a careful consideration of the PF application. For instance, we know that the bottleneck of the UNG model is in the TP block. The BOT model has a complex operation, the triangle function, in the PMP block. With this approach, we can focus on optimizing the TP block in the UNG and the PMP block in BOT model. Regardless, the likelihood evaluation is still a time consuming algorithm. For applications that require very high throughput, the UQLE instruction is a promising step to improve the throughput and reduce the energy consumption. From Table 3.9, using the UQLE instruction results in 1.7× and 1.9× average speedup over the software implementations of the UNG and BOT models, respectively.

Table 3.9 Speedup between running in GPP and in GPP with UQLE instruction for the LG, UNG, and BOT models

Applications	Average execution time (GPP)	Average execution time (GPP with UQLE Instruction)	Average Speedup
LG	1770.56 K	179.20 K	9.88×
UNG	3970.53 K	2333.37 K	1.70×
BOT	3897.25 K	2072.31 K	1.88×

3.6 Conclusion

In this chapter, a novel PF functional view is constructed. It focuses on distinguishing the blocks defined by the applications and algorithms. Under this characterization, we demonstrate that the likelihood evaluation is a time-consuming block and is not affected by the target applications. In order to speed up the execution of the likelihood evaluation, we presented an efficient uniform quantization likelihood evaluation algorithm. We then built an ASIP UQLE instruction to improve its throughput in a custom processor. Simulation results demonstrate that PFs using the proposed UQLE can achieve equal or better performance than particle filters using the exact likelihood evaluation. We also demonstrate that the proposed ASIP UQLE instruction can achieve an average speedup of 805× in comparison with ELE implemented in a GPP, while the processor size only increases 4.7 % for UQLE with 32 intervals.

Chapter 4 NOVEL SYSTEMATIC RESAMPLING ALGORITHMS AND THEIR IMPLEMENTATION IN ASIPS

In this chapter, we propose two novel resampling algorithms to replace the sequential Systematic Resampling (SR) algorithm in Particle Filters (PFs). There are the reformulated SR and Parallel Systematic Resampling (PSR) algorithms.

The reformulated SR algorithm is a new form of the SR algorithm suitable for parallel implementation in an Application-Specific Instruction-set Processor (ASIP). Experimental results show that the ASIP implementation of the reformulated SR algorithm achieves a 23.9× speedup over the sequential SR algorithm in a General-Purpose Processor (GPP). This is for the case of four weights calculated in parallel, and eight categories defined by uniformly distributed numbers that are compared simultaneously. This comes at a cost of only 54K additional gates, or 68% overhead to be added to a base processor with 79K gates.

The reformulated SR algorithm was presented in my contribution [8].

The PSR algorithm makes iterations independent, thus allowing the resampling algorithm to perform loop iterations in parallel. A fixed-point version of the PSR algorithm is also proposed, with a modification to ensure that a correct number of particles is generated. Experiments show that the fixed-point implementation of the PSR algorithm can use as few as 22 bits for representing the weights, when processing 512 particles, while achieving results equivalent to a single-precision floating-point SR implementation. Four customized instructions were designed to accelerate the proposed PSR algorithm in ASIPs. These four custom instructions, when configured to support four weight inputs in parallel, lead to a 53.4× speedup over a floating-point SR implementation on a general-purpose processor at a cost of 47.3 K additional gates.

The PSR algorithm was presented in my contribution [9].

4.1 Introduction

A PF is a sequential Monte Carlo estimation method, often used in signal processing applications [1] [17] [4] [19]. The operation of PFs is based on representing the a posteriori probability density function of system state variables by a group of particles and corresponding weights. Since their introduction in 1993, PFs have gained in popularity to solve non-linear and non-Gaussian problems [2] [13]. It has also been shown that they outperform other filters, including the Extended Kalman Filter [109], in many practical applications such as video tracking [11] [62] and navigation [6] [49].

PFs have four major steps, shown in Fig. 4-1 [1]: Prediction Generation (PG), Weight Calculation (WC), Resampling Processing (RP), and Estimate Calculation (EC).

The PG step generates random samples, which can approximate the a priori probability density function of some quantity or objects of interest. The weight of each particle is then calculated in accordance with information derived from observations in the WC step. The RP step solves the undesirable degeneracy problem, where after a few iterations, most weights tend to have a value of zero and only a few weights retain a substantial value. Resampling consists of removing the particles whose weights are negligible and generating a new group of particles from those with substantial weights. In the EC step, the state estimate can be computed by the minimum mean-square error, the maximum a posteriori probability, or other methods. A weight normalization step, shown separately in previous work [5] [74], may be merged with the RP and EC steps. Such merging can reduce the execution time of PFs. The critical path of PFs is thus composed of the PG, WC, and RP steps. These steps are primary optimization targets in applications requiring high throughput.

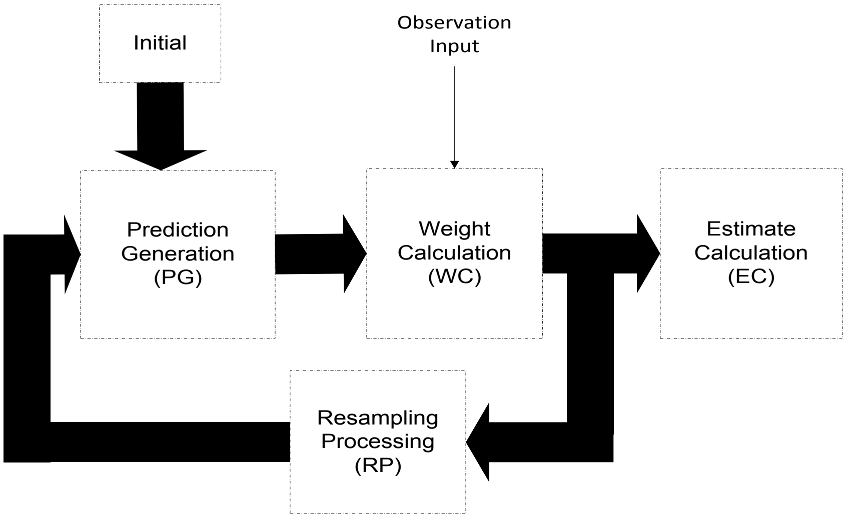

Initial
Observation Input
Prediction Generation (PG)
Weight Calculation (WC)
Estimate Calculation (EC)
Resampling Processing (RP)

Figure 4-1 PF steps

A parallel architecture is an appealing option to improve the throughput of a PF. The execution time of the PG and WC steps can be reduced significantly by exploiting their inherent parallelism. However, the RP step is sequential in nature, due to data dependencies between iterations. Thus, it is a potential bottleneck when other steps are implemented in parallel. The main contribution of this chapter is to propose two novel resampling algorithms. There are the reformulated SR and PSR algorithms. The reformulation SR algorithm enables parallel execution in an ASIP with six custom instructions. The PSR algorithm breaks dependencies between iterations. This is accomplished through the introduction of the concept of Cumulative Replication Factors (CRFs). We also propose a fixed-point version of the PSR algorithm that guarantees a correct number of resampled particles even in the presence of finite precision effects.

In this work, to demonstrate the potential of the PSR algorithm in embedded systems, we targeted ASIP [106] implementation. ASIPs aim to strike a balance between GPPs and Custom Processors (CPs) by combining a programmable solution with customized hardware units. With this approach, the design effort can be reduced significantly when compared to custom processor design. Once performance bottlenecks are found, hardware design can be applied with customized hardware units addressing them to improve the application's overall performance. The second contribution of this chapter is an ASIP design with custom instructions supporting the reformulated SR and PSR algorithms in an efficient and accelerated manner.

4.2 Reformulated SR Algorithm and its ASIP Implementation

4.2.1 The Reformulated SR Algorithm

The SR algorithm cannot be parallelized directly due to the data dependency on the index m of Uniformly Distributed Numbers (UDNs) and the number of iterations that is unknown a priori for the while loop in Fig. 2-2. The reformulated SR algorithm combines a group of these while loops into a parallel while loop. A set of Cumulative Weights (CWs) can be processed in parallel with custom instructions in the parallel while loop. The reformulated SR algorithm does not change any properties of the SR algorithm. Its results are identical when the initial U^1 is the same. Fig. 4-2 shows the data flow graph of the reformulated SR algorithm, and Fig. 4-3 shows the pseudo-code of the parallel while loop. The reformulated SR algorithm described in Fig. 4-2 still exhibits inter loop dependencies, but the pseudo-code in the parallel while loop can be executed in parallel with custom instructions generated from loop unrolling [110] in an ASIP.

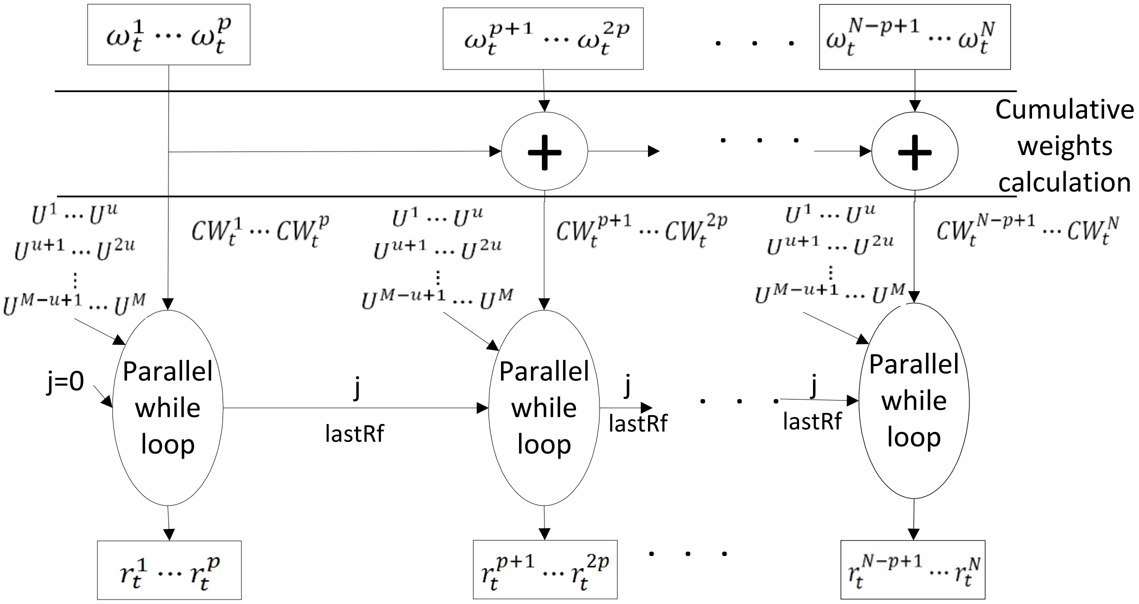

$\omega_t^1 \cdots \omega_t^p$
$\omega_t^{p+1} \cdots \omega_t^{2p}$
$\omega_t^{N-p+1} \cdots \omega_t^N$
Cumulative weights calculation
$U^1 \cdots U^u$
$U^{u+1} \cdots U^{2u}$
$U^{M-u+1} \cdots U^M$
$CW_t^1 \cdots CW_t^p$
$CW_t^{p+1} \cdots CW_t^{2p}$
$CW_t^{N-p+1} \cdots CW_t^N$
j=0
Parallel while loop
j
lastRf
$r_t^1 \cdots r_t^p$
$r_t^{p+1} \cdots r_t^{2p}$
$r_t^{N-p+1} \cdots r_t^N$

Figure 4-2 Data flow graph of the reformulated SR algorithm

The reformulated SR algorithm functions as follows. The CWs and UDNs are separated into N/p groups and N/u groups, where p and u are the number of CWs and UDNs in a group, respectively. Thus, p CWs can be calculated in parallel. In the beginning, the product of the current order of the group of UDNs j and the number of UDNs in a group u is loaded into the Replication Factors (RFs) as shown in lines 1-3 in Fig. 4-3. Each CW is compared with the j^{th} group of u independent UDNs simultaneously. When the CW is greater than the UNDs, the RF value is incremented. The accumulation of the comparisons is added to the RFs as shown in line 7. Lines 9-13 show whether the current executed group of CWs shifts to the next group or is compared with the next group of UDNs. When shifting to the next group of CWs, the RFs in the current group are calculated in accordance with the pseudo-code in lines 15-20. The variable *lastRf* carries the value of the last accumulative comparison results from Line 7 in the previous group to calculate the first RF in the current group. The variable *temp* temporarily reserves the value of the last cumulative comparison results in the current group.

Although the parallel while loops in the reformulated SR algorithm still depend on the order of the group of UDNs, calculating the RFs from the CWs can be executed in parallel with custom instructions as explained in the next section.

```
$\{\{r_t^{ip+1}, r_t^{ip+2}, \dots, r_t^{ip+p}\}, j, temp\} = PWL\{\{CW_t^{ip+1}, CW_t^{ip+2}, \dots, CW_t^{ip+p}\}, j, temp\}$
for k= $1{:}p$
    $r_t^{ip+k} = ju$
end
$nextIteration = true$
while ($nextIteration == true$ )
    for $k = 1{:}p$
        $r_t^{ip+k} = r_t^{ip+k} + (CW_t^{ip+k} > U^{ju+1}) + \cdots\cdots + (CW_t^{ip+k} > U^{ju+u})$
    end
    if $(CW_t^{ip+p} > U^{ju+u+1})$
        $j = j + 1$
    else
        $nextIteration = false$
    end
end
$temp = r_t^{ip+p}$
for k= $p{:}2$
    $r_t^{ip+k} = r_t^{ip+k} - r_t^{ip+k-1}$
end
$r_t^{ip+1} = r_t^{ip+1} - lastRf$
$lastRf = temp$
```

Figure 4-3 Pseudo-code of the parallel while loops

4.2.2 ASIP Implementation

The generation of custom instructions is a popular method to accelerate algorithmic bottlenecks with ASIPs. It involves combining several basic operations into one, often avoiding costly loads and stores. It also facilitates the parallel execution of groups of instructions. Table 4.1 lists and describes the six custom instructions we designed for the reformulated SR algorithm, and Fig. 4-4 shows the resulting pseudo-code.

We do not describe the parallel load and store instructions because they are automatically generated by the implementation tool from the description of the specialized storage. The *cwcalculation* instruction performs the parallel calculation of the CWs from a set of weight inputs. It requires a state register to store the value of the last CW in the current group for further iteration. The execution time to calculate the CWs is reduced when using the *cwcalculation* instruction by a factor equal to the number of weights in a set. The *paradef* instruction is a fusion instruction. It combines a set of instructions to calculate U^1 and U_{size} defined in equation (2.8) and (2.9) into one. Due to the need for U^1 and U_{size} as inputs in the *crfcounter* and *nextiterationdef* instructions, the *paradef* instruction uses two state registers to store the value of U^1 and U_{size} to avoid extra inputs for these two instructions. Since the codes of lines 3 and 7 in Fig. 4-3 are not dependent on the iterations, the *baseadder* and *crfcounter* instructions perform p instructions of lines 3 and 7 per call, respectively. The *nextiterationdef* instruction fuses lines 9-13 into one instruction. The order of the group of UDNs j is stored in a state register to avoid extra load and store instructions. The *rfcalculation* instruction executes the p instructions of Line 17. It uses two temporary state registers to fuse the instructions of lines 15, 19, and 20 with line 17.

Table 4.1 Custom instructions for the reformulated SR algorithm

Name	Description
cwcalculation	Calculate cumulative weights. The delay of its critical path depends on the number of data inputs.
paradef	Calculate U^1 and U_{size}.
baseadder	Calculate the product of the current order of the group of UDNs j and the number of UDNs in a group u to store in the RFs in parallel (lines 1-3 in Fig. 4-3).
crfcounter	Accumulate the results of the comparison between CWs and UDNs in parallel (lines 6-8 in Fig. 4-3).
nextiterationdef	Determine whether to process the next group of the CWs or the next group of UDNs (lines 9-13 in Fig. 4-3).
rfcalculation	Calculate the RFs via obtaining the difference between two successive accumulative comparison results in parallel (lines 15-20 in Fig. 4-3).

According to the pseudo-code in Fig. 4-3, the three for loops (lines 2-4, 6-8, and 16-18) are major bottlenecks in the parallel while loop. We use the *baseadder*, *crfcounter*, and *rfcalculation* instructions to execute these three loops in parallel and thus significantly reduce the execution time.

```
$\{\{r_t^i\}_{i=1}^N\} = reformulated\ SR\{\{\omega_t^i\}_{i=1}^N\}$
Cumulative Weights Calculation:
for $i = 1{:}\,p{:}\,N$
    $\{CW_t^i, \dots, CW_t^{i+p}\} = cwcalculation(\{\omega_t^i, \dots, \omega_t^{i+p}\})$
end
$U^1$ and $U_{size}$ definition:
$paradef(CW_t^N)$
RFs calculation:
for $i = 1{:}\,p{:}\,N$
        $\{r_t^i, \dots, r_t^{i+p}\} = baseadder()$
        $nextIteration = true$
        while ( $nextIteration\ == ture$ )
                $\{r_t^i, \dots, r_t^{i+p}\} = crfcounter(\{r_t^i, \dots, r_t^{i+p}\}, \{CW_t^i, \dots, CW_t^{i+p}\})$
                $nextIteration = nextiterationdef(CW_t^{i+p})$
        end
        $\{r_t^i, \dots, r_t^{i+p}\} = rfcalculation(\{r_t^i, \dots, r_t^{i+p}\})$
end
```

Figure 4-4 Pseudo-code of the reformulated SR algorithm

4.2.3 Performance

The Xtensa LX2 processor [107] was used to implement the reformulated SR algorithm with 512 particles. The Xtensa LX2 is a GPP to which custom instructions can be added to improve performance. Table 4.2 compares the speedup and additional resource requirements of the serial and reformulated SR algorithms. For the basic SR algorithm, we consider the cases without and with a FPU. The PA process is not considered in this experiment. The performance of the reformulated SR algorithm depends on the number of CWs and UDNs calculated in parallel. The speedup is 10.2× when inputting two CWs in parallel and comparing one UDN in each iteration, with a cost of 28.9 K additional gates. When inputting four CWs in parallel and comparing eight UDNs simultaneously, the speedup reaches 23.9× at a cost of an additional 54.0 K gates. Fig. 4-5 plots the various solutions. The ASIP implementation of the proposed reformulated SR algorithm is significantly superior to a GPP implementation with FPU in terms of speedup and extra resources. Further performance improvements can be achieved at the expense of the extra resources when increasing the number of CWs and/or UDNs as inputs in the custom instructions.

Table 4.2 Performance and resource requirements for the reformulated SR algorithm in ASIP implementation

Implementation method	# of CWs	# of UDNs	Clock cycles	Speedup	Additional area (gates)
SR without FPU	-	-	64.1 K	1×	0
SR with FPU	-	-	12.7 K	5.05×	77.0 K
Reformulated SR in ASIP	2	1	6.26 K	10.24×	28.9 K
	2	2	5.46 K	11.74×	30.4 K
	2	4	4.99 K	12.85×	35.5 K
	2	8	4.73 K	13.55×	45.6 K
	4	1	4.10 K	15.63×	35.6 K
	4	2	3.32 K	19.31×	37.6 K
	4	4	2.93 K	21.88×	42.8 K
	4	8	2.68 K	23.92×	54.0 K

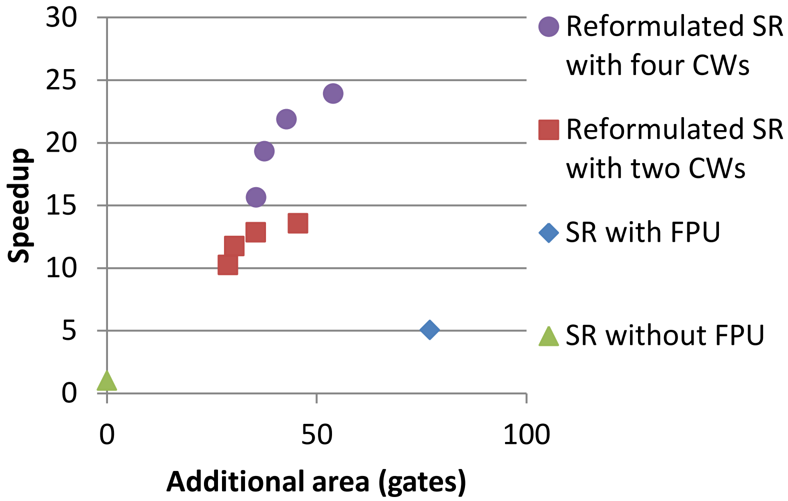


Figure 4-5 Speedup vs additional gates for various solutions

4.3 Parallel Systematic Resampling Algorithm and its ASIP Implementation

4.3.1 Parallel Systematic Resampling Algorithm

In order to parallelize the SR algorithm's loop, the proposed PSR algorithm introduces Cumulative Replication Factors (CRFs) $\{cr_t^i\}_{i=1}^N$ to avoid calculating the RFs directly. The RFs can be recovered by calculating the difference between two successive CRFs.

In the SR algorithm, and as shown by equation (2.8), the UDNs can be considered as M intervals of length U_{size} in the range $[U^1, U^1 + U_{size}M]$. The resultant CRF of the target CW is equivalent to the index of the interval in which the CW falls. Once the interval index is calculated by knowing the interval where the CW is located, the CRFs can be obtained. Equation (4.1) shows the relationship between the CWs and the CRFs.

$$\exists\ cr_t^i : U^1 + U_{size}(cr_t^i - 1) < CW_t^i \leq U^1 + U_{size} cr_t^i, i = 1,2,3, \dots, N \tag{4.1}$$

Once the CWs are available, their CRFs can be obtained independently by equation (4.2).

$$cr_t^i = \left\lceil (CW_t^i - U^1)/U_{size} \right\rceil, i = 1,2,3, \dots, N \tag{4.2}$$

Fig. 4-6 shows the data flow graph of the proposed PSR algorithm. Similarly to the SR algorithm, the PSR algorithm first calculates the CWs $\{\mathrm{CW_t^i}\}_{\mathrm{i=1}}^{\mathrm{N}}$. The sequential computation of the CWs is straightforward and it can be supported by specialized instructions in ASIPs or parallel prefix-sum algorithms for GPUs [111]. The quantities U_{size} and U^1 are calculated with equations (2.8) and (2.9). The CRFs can then be calculated by equation (4.2). The loop iterations for the CRF calculations are independent from each other, and can thus be unfolded and executed in parallel. This is shown in Fig. 4-6, where the operations to calculate the CRFs are shown as independent processes. Next, the RFs are calculated as the difference between two successive CRFs, with a dependency that is limited to a single vertical branch.

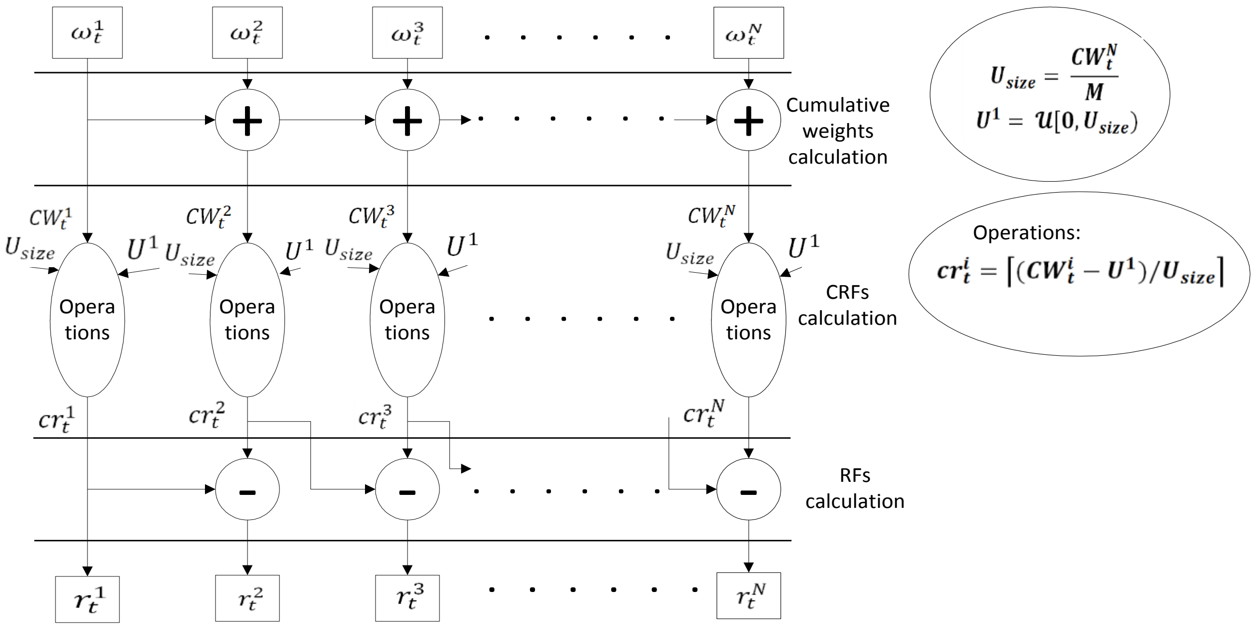


Figure 4-6 Data flow graph of the proposed Parallel Systematic Resampling algorithm

4.3.2 Fixed-point Implementation of the PSR Algorithm

Fixed-point processing can significantly improve the throughput while reducing the amount of necessary hardware resources. However, finite-precision effects may result in errors in the RFs when compared with calculations done with floating-point precision. Moreover, when the calculations are done in fixed-point arithmetic, the division in equation (2.9) requires a floor function. This may result in U_{size} being smaller than its value in floating-point arithmetic. When calculating the last CRF in equation (4.2), U_{size} is used as the divisor. Due to U_{size} being small, the resultant last CRF may be larger than the number of resampled particles, M. This potentially results in an incorrect number of resampled particles.

For example, consider the case of 1024 particles, a value of 10000 for CW_t^N , and a value of 2 for U^1. According to equation (2.9), U_{size} is equal to 9 in fixed-point arithmetic, from its value of 9.77 in floating-point precision. The last CRF calculated according to equation (4.2) would thus be equal to 1111, while the correct number of resampled particles should be 1024.

In order to mitigate the problems due to the quantization of U_{size}, we propose to merge equations (2.9) and (4.2) to calculate the CRFs instead of calculating U_{size} as the divisor in equation (4.2). Its calculation is given by equation (4.3):

$$cr_t^i = \left\lceil M\left(CW_t^i - U^1\right)/CW_t^N \right\rceil, i = 1,2,3,\dots,N \tag{4.3}$$

When calculating the last CRF, equation (4.3) can be transformed into equation (4.4):

$$cr_t^N = \lceil M(CW_t^N - U^1)/CW_t^N \rceil = \lceil M - MU^1/CW_t^N \rceil \tag{4.4}$$

In order for the number of resampled particles to be equal to *M*, the last CRF should be equal to *M*. According to equation (4.4), the last CRF being equal to *M* can be ensured when the condition expressed in equation (4.5) is satisfied.

$$MU^1/CW_t^N < 1 \tag{4.5}$$

Due to the fact that U^1 is in the range of $[0, U_{size})$ and CW_t^N is equal to MU_{size}, equation (4.5) can always be satisfied, whether calculations are done in fixed or floating-point precision. Using equation (4.3) to calculate the CRFs instead of equation (4.2) guarantees correct results in the estimated number of resampled particles in fixed-point arithmetic at the expense of an additional multiplication and the relevant additional bits to represent the product of the multiplication.

Fixed-point implementations normally imply that a custom processor is being designed. Special attention must thus be paid to the nature of the computations. The PSR algorithm requires the division and ceiling functions, but these more complex operations can be implemented efficiently in hardware. The CRFs can be calculated by an integer division with a conditional statement: if the remainder is not equal to zero, the quotient, which is equivalent to the CRF, is incremented.

4.3.3 ASIP Implementation of the Fixed-Point PSR Algorithm

A hardware implementation is an appealing method to meet high throughput requirement for PFs. The ASIP approach can be a good choice since it balances the trade-offs between GPPs and custom processors. ASIPs allow a programmable solution with customized hardware units to accelerate application bottlenecks and improve throughput. They can significantly reduce the design effort when compared to fully custom processors. Custom instruction generation is the most popular method to accelerate the algorithmic bottlenecks in ASIPs. Custom instructions can combine several basic operations into a single one, and multiple instructions can be executed in parallel.

Based on the data flow graph of Fig. 4-6, we designed four custom instructions to accelerate the implementation of the PSR algorithm. They are described in Table 4.3.

Table 4.3 Custom instructions for the proposed PSR algorithm

Name	Description
cwcalculation	Calculate the CWs in parallel. The delay of its critical path depends on the number of data inputs.
paradef	Calculate U^1 defined. The value is stored in a state register in order to avoid memory accesses for further instructions.
Integerdivisionforcrf	Calculate CRFs in parallel using equation (4.3). Because the quotient should be smaller than or equal to the number of particles, we use a subtraction/shift integer division with a known number of bits for the quotient [112].
rfcalculation	Calculate the RFs as the differences between two adjoining CRFs in parallel.

Fig.4-7 gives the PSR algorithm pseudo-code after inclusion of these four custom instructions. Parameter p is the number of data calculated in parallel. The operations inside each step can be executed in parallel, except for the definition of U^1, which is merged into the instruction *paradef.*

```
Input: $\{\omega_t^i\}_{i=1}^N$                Output: $\{r_t^i\}_{i=1}^N$
Cumulative Weights Calculation
for $i = 1{:}p{:}N$
    $\{CW_t^i, CW_t^{i+1}, \dots, CW_t^{i+p}\} = cwcalculation(\{\omega_t^i, \omega_t^{i+1}, \dots, \omega_t^{i+p}\})$
end
$\mathbf{U^1}$ definition
$paradef(CW_t^N)$
CRFs calculation
for $i = 1{:}p{:}N$
    $\{cr_t^i, cr_t^{i+1}, \dots, cr_t^{i+p}\} = Integerdivisionforcrf(\{CW_t^i, CW_t^{i+1}, \dots, CW_t^{i+p}\})$
end
RFs Calculation
for $i = 1{:}p{:}N$
    $\{r_t^i, r_t^{i+1}, \dots, r_t^{i+p}\} = rfcalculation(\{cr_t^i, cr_t^{i+1}, \dots, cr_t^{i+p}\})$
end
```

Figure 4-7 Pseudo-code of the PSR algorithm in ASIP implementations

4.3.4 Implementation Results

4.3.4.1 Accuracy for Fixed-Point Processing

When implemented with floating-point precision, the PSR algorithm produces exactly the same results as the SR algorithm, with the same accuracy. However, in a fixed-point implementation, the accuracy varies with the number of bits used to represent the weights. Due to the stochastic nature of the algorithm, we counted the number of times that the set of RFs was different between the fixed-point PSR and single-precision floating-point SR algorithms for 10000 simulations with 128, 256, 512 or 1024 particles. Results are shown in Table 4.4. The weights in the fixed-point PSR implementation are represented with a varying number of bits. For each experiment, we used a single value for U^1 for the two algorithms. It is important to note that the number of resampled particles was always the same in both algorithms as discussed in Section 4.3.2.

Table 4.4 Number of times that incorrect RFs were generated by fixed-point PSR algorithm for 10000 simulations.

M	Number of bits to represent the weights							
	2	8	16	20	22	24	28	32
128	9996	2024	9	1	0	0	0	0
256	10000	4484	22	2	0	0	0	0
512	10000	7857	62	3	0	0	0	0
1024	10000	9739	90	17	2	1	0	0

From Table 4.4, we observe that increasing the number of bits to represent the weights reduces the frequency of observed errors in the calculation of the RFs. For 128, 256, and 512 particles, using 22 bits resulted in no observed error for 10000 simulations. This represents a reduction in data path width of 31% over the 32-bit case. With 1024 particles, using only 16 bits for the weights would result in an error in the number of RFs in less than 1% of cases. For 128 particles, it would be in less than 0.1% of cases. Due to the stochastic nature of PFs, we believe that these levels of accuracy would be acceptable for most applications.

4.3.4.2 Performance of Fixed-Point Processing and ASIP Implementation

The proposed PSR algorithm was implemented in an Xtensa LX2 processor which runs at 300MHz with 512 particles. The weights were represented with 32 bits because it is not possible to change the Xtensa's basic datapath. The Xtensa LX2 is a general-purpose processor to which custom instructions can be added to improve its performance. In its basic form it occupies 79 K gates. It can include a FPU when necessary, at a cost of an additional 77 K gates. We implemented the floating-point SR algorithm in this processor without and with FPU as references. For comparison purposes, we used the RSR algorithm sequentially implemented on a TI TMS320C54x DSP [5] as reference to evaluate our fixed-point PSR algorithm. This DSP supports floating-point operations, including the multiplication and ceiling functions. The maximum clock rate for this DSP C54 is 160MHz. Table 4.5 compares the speedup and additional resource requirements for different scenarios.

Table 4.5 Performance and resource requirements of various implementation methods

Implementation method	Parallelism for the weights	Clock cycles	Clock Frequency (Hz)	Speedup	Additional area (gates)
Floating-point SR on Xtensa LX2	-	64.1K	300M	1.0×	0
Floating-point SR on Xtensa LX2 with FPU	-	12.7K	300M	5.05×	77.0 K
Floating point RSR on DSP [5]	-	9.2 K	160M	3.71×	N/A
Fixed-point PSR ASIP with *Integerdivisionforcrf* instruction	-	5.1 K	300M	12.57×	15.5 K
Fixed-point PSR ASIP	2	2.2 K	300M	29.14×	27.5 K
	4	1.2 K	300M	53.42×	47.7 K

The fixed-point PSR implementation requires the fixed-point division operation, which the Xtensa LX2 processor does not support. Its performance is thus inferior to that of the floating-point SR implementation, because executing fixed-point division with a software library requires a large number of clock cycles. When using the specific *Integerdivisionforcrf* instruction without any parallelism, the fixed-point PSR implementation can achieve 12.57× speedup over the floating-point SR implementation on the unmodified Xtensa LX2 with a resource overhead of 15.5 K gates. We observe in Table 4.5 that this sequential implementation has better performance than the RSR algorithm sequentially implemented on a DSP. This indicates that the PSR algorithm has less complexity than the RSR algorithm, which is the least complex of the three standard RAs mentioned in Section 2.1. When compared to the floating-point SR implementation executed over an Xtensa with a FPU, the fixed-point PSR implementation is still better in terms of throughput and extra resources. This experiment shows that fixed-point processing can be valid for software PSR implementation when the target processor supports fixed-point division.

In addition, the fixed-point PSR algorithm can be easily implemented in parallel. In its ASIP implementation, its performance can be significantly improved with our proposed four special custom instructions. From Table 4.5, with two parallel paths and all four custom instructions, the speedup is 29.1×. This is achieved at a cost of 27.5 K additional gates, or 35% of the basic processor. With four parallel paths, the speedup is 53.4× and the cost is 47.3 K additional gates, or 60% of the basic processor.

This speedup can be further improved when calculating more than four particles in parallel. However, this would require that the memory bandwidth support the data throughput and that silicon area be available.

4.4 Conclusion

In this chapter, we proposed two new resampling algorithms to replace the sequential SR algorithm in PFs. The reformulated SR algorithm makes the sequential SR algorithm feasible for ASIP parallel implementation. The PSR algorithm makes the iterations in the SR algorithm independent, and thus allowing the resampling algorithm to perform the iterations in parallel. The reformulated SR algorithm is faster than the PSR algorithm when implemented in a GPP. This is because the former has relatively simple operations such as comparator and addition, and the latter has division and ceiling functions. The advantage of the PSR algorithm is that it is a pure parallel resampling algorithm and thus it can be used in any parallel implementation technology. The reformulated SR algorithm is only suitable for ASIP parallel implementation. Furthermore, the PSR algorithm has the least complexity among the standard RAs. When they are implemented in a processor supporting for the integer division, the PSR algorithm outperforms all of the other standard RAs including the reformulated SR algorithm.

Chapter 5 PARALLEL ARRAY HISTOGRAM ARCHITECTURE FOR EMBEDDED IMPLEMENTATIONS

In histogram-based Particle Filters (PFs) for video tracking, the bottleneck is often in the Particle Measurement Processing (PMP) block, shown in Fig. 3-1. It is therefore critical to accelerate the execution of this block. To that effect, this chapter proposes a parallel array histogram architecture (PAHA) suitable for embedded implementations. PAHA uses a register array instead of a memory block to store the histogram bins. In each step, M inputs can be processed in parallel to update the histogram bins without any additional latency. This chapter also describes a second version of PAHA with a flexible number of inputs, potentially avoiding the need for multiple PAHAs in a single application. Implementation results show that the architecture can achieve a super-linear speed-up of 43.75× for a 16-way PAHA when compared to a software implementation in a general-purpose processor.

The material of this chapter was published in my paper [10].

5.1 Introduction

Histograms or binning functions are representations of frequency distributions. Their calculation is a relatively simple operation and it is often performed by GPPs. Nonetheless, hardware acceleration may be necessary to satisfy high throughput requirements for some real-time applications, such as histogram-based particle filters for video object tracking [11] [84].

Shahbahrami et al. [113] have discussed possible software implementations for calculating histograms. The Scalar Histogram Algorithm (SHA) is the fastest reported algorithm when compared to the privatization or merge-sort algorithms. It is described as follows:

```
For $i=1...K$
        Histogram [data[$i$]] +=1;
```

The SHA is the basis for the histogram implementation described in this chapter. Concerning hardware implementation of SHA, Muller [114] showed that two basic architectures can be used. The first one addresses the memory array with a read-modify-write architecture. Due to its efficient resource usage, most techniques [115] [116] have investigated such a read-modify-write architecture. However, its performance depends on the number of access ports of the memory array. In practice, a dual-port memory array is often used to implement the SHA but the maximum speed-up is 2× when compared to its implementation using a single port memory array. The second basic architecture uses an array of counters, with one counter for each histogram bin. However, this architecture is costly in resources for the register array and that is why it is seldom used. In order to overcome the limitation of 2× speed-up for the first architecture, Cadenas et al. [117] [118] employed the second architecture, the array of counters, in a fully pipelined manner to process the inputs in parallel. However, their pipelined histogram array (PHA) has a large latency when using only one or two histogram bins per cell. It occupies a large amount of resource due to the pipeline technique. When using a single cell for all the bins with multiple inputs, their architecture becomes complex and its delay is increased.

This chapter firstly presents that histogram calculation is a major bottleneck in histogram-based PFs for video tracking and then introduces a parallel array histogram architecture (PAHA) that does not involve the use of pipelining techniques. It has better resource efficiency when compared to the recent work by Cadenas et al. [117] [118]. The chapter also introduces a version of PAHA with a flexible number of inputs, called Flexible PAHA (FPAHA), for applications where the number of items is not known in advance or can take several known values.

5.2 Bottleneck in Histogram-based PFs for Video Tracking

As we described in subsection 2.2.2, in the field of computer vision systems, video tracking is an important issue in many applications. It can provide spatial information of target objects for further analysis. Consequently, many approaches have been developed for video tracking. Among these methods, PFs for video tracking employing the color models is one such successful approach [3] [65] [119]. It performs sequential Monte Carlo method [17] based on particle representation of probability density to search the region of interest (ROI) whose color content expressed in histogram matches the color content of the reference target. The results reported in [3] [65] show that it is independent of the object deformation and robust to occlusion and to variations in the color of the background. This makes it suitable for general visual tracking systems.

Histogram-based PFs have shown great promise as a powerful method for video tracking. However, they suffer from a vital drawback: heavy computation. With the increase of the ROI size, the number of particles required, and the dimensions of color channels as the measurement, the phenomenon for heavy computation becomes even worse. Table 5.1 shows the profiling results in an Xtensa LX2 processor for an object (120 by 140 pixels) being tracked by using a general histogram-based PF with 256 particles. Under these circumstances, more than 50 million clock cycles are required to complete the tracking in a frame. Assuming that the Xtensa LX2 processor runs at 300MHz clock frequency, the resulting 6 frames per second for this case are not sufficient for most of real-time video tracking systems. We also observe from Table 5.1 that the PMP block takes a significant portion of the total number of clock cycles (97.6%). Obviously, it is necessary to accelerate the PMP block in order to achieve the throughput requirement of real-time video tracking. The only calculations performed in the PMP block are for the histogram. As discussed in Section 5.1, the SHA is the fastest reported histogram calculation algorithm. It is difficult to be optimized in the algorithm level. This motivates us to accelerate the SHA in the architectural level.

Table 5.1 Profiling results of histogram-based PF with 256 particles tracking an object (120 by 140 pixels) in the Xtensa LX2 processor

	Clock cycles per frame	%
Transition Processing (TP)	0.00 M	0%
Random Number Addition (RNA)	0.01 M	0.02%
Particle Measurement Processing (PMP)	52.19 M	97.61%
Distance Calculation (DC)	0.36 M	0.67%
Likelihood Evaluation (LE)	0.86 M	1.61%
Resampling Algorithm (RA)	0.05 M	0.09%
Total	53.47 M	100%

5.3 Parallel Array Histogram Architecture

Fig. 5-1 shows the PAHA. It has M N-bit parallel inputs. Each input is fed to a $N{:}2^N$ decoder. The decoders' 2^N 1-bit outputs are each fed to one of 2^N accumulators. The Accumulators are each associated with one register bin. The role of the Accumulators is to accumulate the M 1-bit signals coming from the decoders, to add them to the register bin contents, and then to update the register bins. The width P of the register bins is selected to avoid overflow and depends on the maximum number of inputs that can be presented to the PAHA. In each clock cycle, PAHA thus processes M inputs in parallel and updates the register bins accordingly.

The architecture of one of the M-bit accumulators is shown in Fig. 5-2. It consists of a Compressor Tree (CT) [120] and a two-operand adder. The CT counts the numbers of 1's in the M 1-bit signals and generates a R-bit operand. The relation between R and M is given by equation (5.1):

$$R = \lceil \log_2(M+1) \rceil \tag{5.1}$$

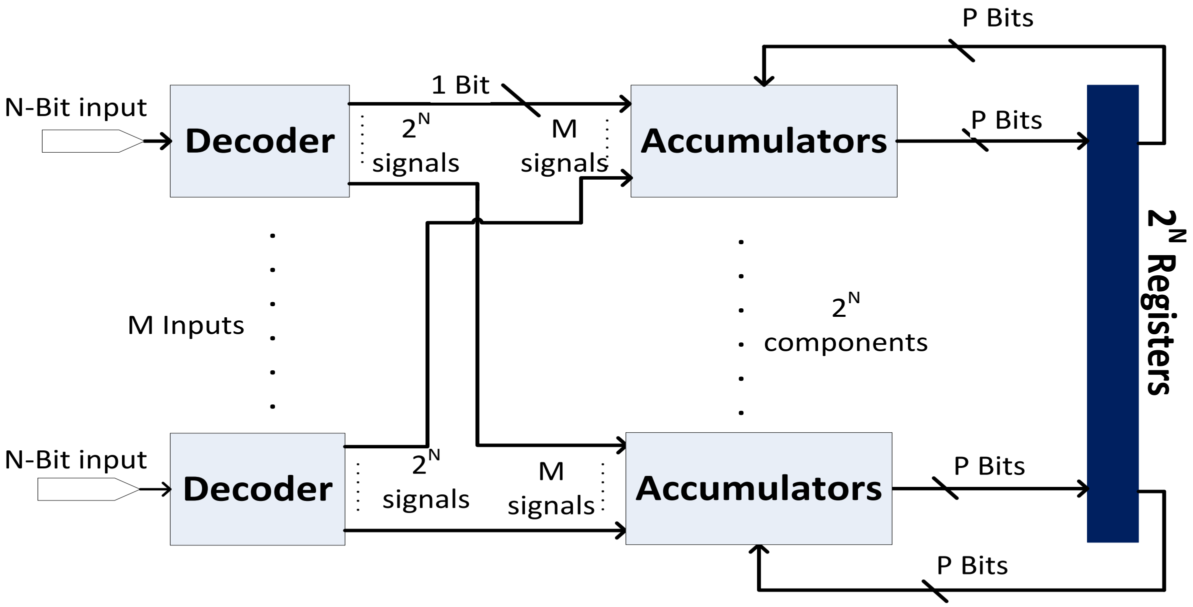


Figure 5-1 Parallel Array Histogram Architecture

The CT's area and delay increase linearly with M. The two-operand adder's area and delay depends on P and M. When P is fixed, its area and delay increase logarithmically with M.

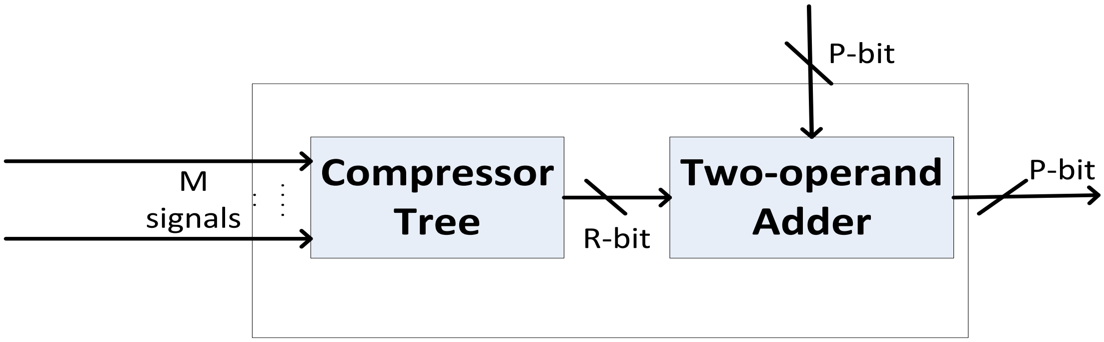


Figure 5-2 M-bit Accumulator

5.4 Flexible Parallel Array Histogram Architecture

Fig. 5-3 shows a modified version of PAHA with a flexible number of active inputs up to *M*. The number of inputs can vary in applications where the data set size is not an integer multiple of *M*, such as when the data is not aligned with the cache size. In such a case, the first or last group of data or the misaligned data presented to the PAHA will consist of $m < M$ inputs. It is then necessary to disregard the $M–m$ least or most significant items in the list. The FPAHA can accommodate such a situation.

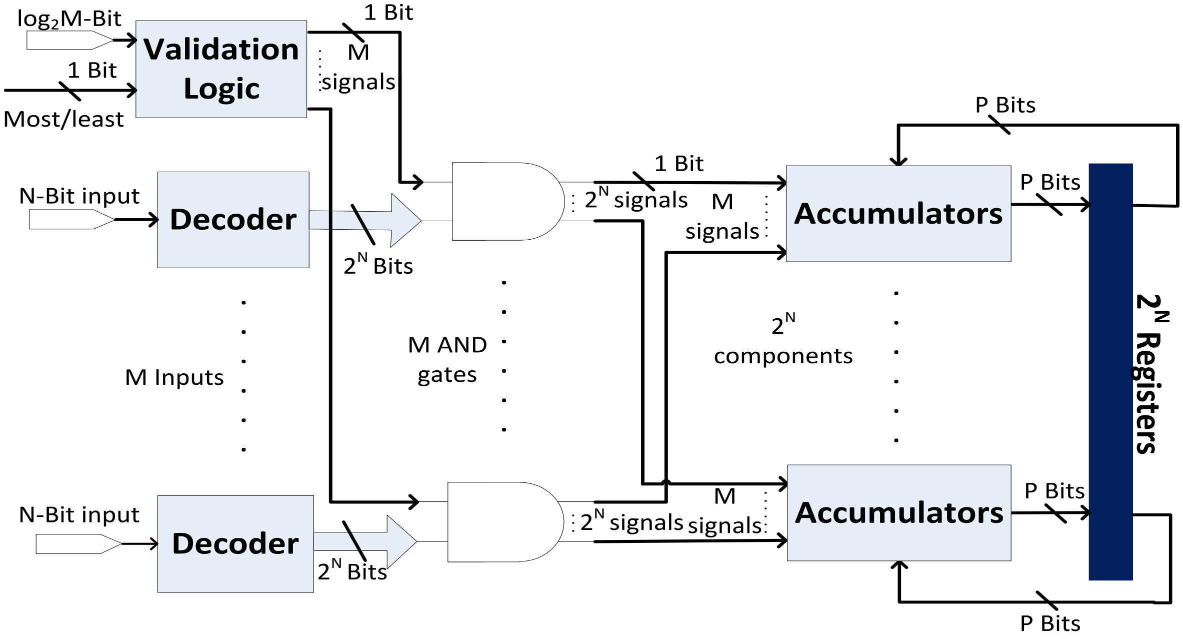


Figure 5-3 Flexible Parallel Array Histogram Architecture

Compared to the PAHA, the FPAHA requires an additional validation logic block and $M \times 2N$ AND gates. The validation logic determines which inputs must be considered to update the register bins. Its inputs are the number of inputs that must be used and a 1-bit most/least signal to determine whether the most or least significant items are to be used. For each unused input, the validation logic generates a '0' on its corresponding control signal. In conjunction with the AND gates, this nullifies the corresponding input to the accumulators.

5.5 Architecture Comparisons

Table 5.1 compares delay, latency, parallelism and resource requirements for the array of counters [114], the parallel histogram with dual-ported memory [115], the PHA [117], the PAHA, and the FPAHA for the specific case of $(M, N, P) = (8, 8, 20)$. I only use 1-bin PHA to represent the PHA as reference because it is difficult to design multiple-bin PHA with large number of parallel inputs. In Table 5.2, τ is the basic unit delay and the evaluation of the resources is based on NAND gates. The delay for the array of counters consists of a $N{:}2^N$ decoder and a P-bit counter. The parallel histogram's delay consists of a two-operand P-bit adder. The delay for the 1-bin PHA consists of an N-bit comparator, logic to determine the count for the one bin and a two-operand adder with N-bit and R-bit operands. The logic for count determination in the 1-bin PHA can be considered as an M-bit CT. However, with the increase of the number of bins per cell for the PHA, the logic for count determination becomes complex. Consequently, its delay is longer than the delay of the CT. The delay for the PAHA consists of an $N{:}2^N$ decoder and an M-input accumulator which includes an M-bit CT and a two-operand adder with P-bit and R-bit operands. As shown in Table 5.2, the PAHA and PHA are the only architectures which can execute multiple inputs in parallel. In addition, the PAHA has lower delay and latency than the PHA. The throughput is used to combine the delay and input parallelism information of each architecture to show the performance, which is given by equation (5.2).

$$\text{Throughtput} = \frac{\text{Parallelism}}{\text{Delay}} \tag{5.2}$$

Table 5.2 also presents resource usage based on NAND gates for the considered architectures when $(M, N, P) = (8, 8, 20)$ except for the parallel histogram because it depends on its memory architecture. The array of counters only requires a $N{:}2^N$ decoder and 2^N P-bit counters. Hence, it only spends 74.9 K NAND gates in our case. Due to the use of pipelining, the PHA uses additional registers to store the register bin indices and the value of inputs. It thus requires $2^N \times M$ N-bit comparators, 2^N logic for count determination, 2^N Two-operand adder, 2^N P-bit registers, 2^N N-bit registers and 2^N N-bit counters. The PAHA uses M $N{:}2^N$ decoders, 2^N accumulators and 2^N P-bit registers. From Table 5.2, we see that the PHA uses more resources than the PAHA because the PAHA does not use pipeline registers. Also, the M decoders in PAHA are smaller than $2^N \times M$ N-bit comparators in the PHA. Furthermore, the resources of the PHA increase with the number of bins due to the complexity of the logic for count determination. The penalty for using the FPAHA is the additional delay of an AND gate, the resources for the validation logic and $M \times 2^N$ AND gates. This may be acceptable in order to gain more flexibility.

Table 5.2 Comparison of PAHA and FPAHA against previous work for the case of $(M, N, P) = (8, 8, 20)$

Architecture	Delay	Latency	Parallelism	Throughput	Resources
Array of counters [114]	72τ	1	1	0.0139	74.9 K
Parallel histogram [115]	120τ	258	2	0.0167	—
1-bin PHA [117]	103τ	256	8	0.0777	234.2 K
PAHA	99τ	1	8	0.0808	109.3 K
FPAHA	101τ	1	8	0.0792	115.6 K

5.6 Implementation Results

The Xtensa LX2 processor [107], an ASIP, was used to implement the SHA for an image of size 640×480. The Xtensa LX2 is a general-purpose processor to which customizable instructions can be added to improve performance. Custom instructions were generated for the PAHA and FPAHA. Table 5.3 shows the speedup and additional resource requirements for M-way PAHAs when compared to the software implementation of SHA. We observe that the 1-way PAHA can achieve 3.23× speedup with a resource overhead of 329.9 K gates. The speedup is due to the reduction in memory access instructions since state registers are used to store the histogram. When the number of inputs *M* is increased to 4 and 16, the speedup attains 13.13× and 43.75×, respectively. The 16-way FPAHA requires only 0.7 K more gates than the 16-way PAHA.

Table 5.3 Performance and resource requirements of various *M*-way PAHAs on the Xtensa LX2 processor

	Performance		Resource requirement (Number of gates)		
	number of clock cycle (M)	Speedup	Base processor	Overhead	Total
SHA	2.15	1×	79 K	0 K	79 K
1-way PAHA	0.65	3.23×	79 K	329.9 K	408.9 K
4-way PAHA	0.16	13.13×	79 K	356.1 K	435.1 K
16-way PAHA	0.048	43.75×	79 K	468.4 K	547.4 K
16-way FPAHA	0.048	43.75×	79 K	469.1 K	548.1 K

5.7 Conclusion

This chapter has presented a parallel array histogram architecture for embedded implementations. A register array is used to overcome the limited speed-up due to memory access. The proposed architecture achieves better performance in terms of throughput when compared to previous work. Implementation results show the speed-up can attain 3.2×, 13.1× and 43.7× for 1-way, 4-way and 16-way PAHAs, respectively, when compared to a traditional software implementation.

Chapter 6 CONCLUSIONS AND FUTURE WORK

6.1 Thesis Conclusions

PFs, which are computationally very intensive, are popular in addressing a wide range of complex applications due to the fact that they outperform most of the traditional filters in practice. It remains a challenge to simultaneously achieve high throughput and energy efficiency in most of these applications.

In this thesis, we investigated a promising and popular design platform, ASIPs, to implement PFs in embedded systems. We analyzed the characteristics of PFs and then identified the Likelihood Evaluation (LE) and Resampling Processing (RP) as the major bottlenecks in the algorithm-specific blocks. The optimization of these two blocks was explored at the algorithm and architecture level.

We proposed UQLE instead of ELE in PFs and then designed its custom instruction and relevant hardware unit to further accelerate the LE block. Experiments showed that the proposed ASIP UQLE instruction can achieve an average speedup of 805× in comparison with ELE implemented in a GPP. This comes at a cost of only 3.8K additional gates for UQLE with 32 intervals, or 4.7 % overhead to be added to a base processor with 79K gates.

Concerning the RP block, we proposed two resampling algorithm instead of the sequential SR algorithm in PFs. They are the reformulated SR and the PSR algorithms. The former is a new form of the SR algorithm to facilitate its parallel ASIP implementation. We designed six custom instructions for the reformulated SR algorithm. Experimental results showed that ASIP implementation of the reformulated SR algorithm achieves a 23.9× speedup over the sequential SR algorithm in a GPP. This is for the case of four weights calculated in parallel and eight categories defined by uniformly distributed numbers that are compared simultaneously. The additional size for these six instructions occupies 68% of a base processor.

The PSR algorithm makes iterations independent through the introduction of the concept of Cumulative Replication Factors. In addition, a fixed-point version of the PSR algorithm was proposed to ensure that the correct number of particles is generated. Four customized instructions were designed to accelerate the proposed PSR algorithm in ASIPs. When supporting four weights inputting in parallel, its ASIP implementation with four relevant custom instructions lead to a 53.4× speedup over a floating-point SR implementation in a GPP at a cost of 47.3 K additional gates.

In order to accelerate histogram-based PFs for video tracking, we proposed a Parallel Array Histogram Architecture (PAHA) for embedded implementations. M inputs can be processed in parallel to update the histogram bins without any additional latency. Implementation results show that 16-way PAHA can achieve a 43.75× over a software histogram implementation in a GPP.

6.2 Future Work

There are several topics that could be interesting to investigate in future research.

- UQLE can achieve a significant speedup for PFs by replacing ELE in the LE block and relevant custom hardware architecture. However, in order to achieve the equivalent accuracy to ELE, UQLE needs an appropriate number of intervals. Due to the stochastic nature of PFs, finding this number requires a large amount of experiments. A possible future work would be to automatically identify the number of intervals in UQLE for a given application.

- The UQLE calculation is independent for each particle. Therefore, it could be implemented in parallel to achieve higher throughput than our sequential implementation.

- Our two novel algorithms make the resampling algorithm feasible for parallel implementation. They can calculate the RFs in a parallel manner. However, after calculating the RFs, we did not do the parallel implementation for the PA step since such a parallel implementation depends on the implementation technology. A possible future work would be to exploit the parallel implementation of the PA step in accordance with the target implementation technology.

- The PAHA can achieve a significant speedup when compared to a software histogram calculation executed in a GPP. However, special registers must be instantiated for the histogram bins, and they must be wide enough to store the largest possible value that a bin can take. Their large size makes them important contributions to overall processor power consumption. Therefore, it would be interesting to apply low power technique to their design.

- In this thesis, the research mainly focuses on PF algorithms. Their bottlenecks at the algorithm-specific blocks are the LE and RP blocks. However, we mentioned in Section 2.1 and Section 2.2 that there are various types of PFs commonly used and there are various applications using PFs. Depending on the application and target model, the bottlenecks may be shifted to other blocks. A possible future work would involve the analysis of different types of PFs from the point of view of the computational complexity of the target model and its impact on the various PF steps. The main emphasis, then, should be in identifying and resolving these bottlenecks at the algorithm and architecture levels. For example, for histogram-based PFs for video tracking, only optimizing histogram calculation in the PMP block may not satisfy real-time constraints. Other potential bottlenecks in other blocks would be evaluated and optimized in ASIPs to improve the performance of video tracking.

REFERENCES

[1] B. Ristic, S. Arulampalam and N. Gordon, *Beyond the Kalman filter: particle filters for tracking applications*, Boston, MA, USA: Artech House, 2004.

[2] N. J. Gordon, D. J. Salmond and . A. F. M. Smith, “Novel-Approach to Nonlinear Non-Gaussian Bayesian State Estimation,” *IEEE Proceedings-F Radar and Signal Processing,* vol. 140, no. 2, pp. 107-113, 1993.

[3] K. Nummiaro, E. Koller-Meier and L. Van Gool, "A Color-based Particle Filter," *in First International Workshop on Generative-Model-Based Vision*, vol. 1, pp. 53-60, 2002.

[4] A. Doucet, N. De Freitas and N. Gordon, *Sequential Monte Carlo methods in practice*, New York, NY, USA: Springer, 2001.

[5] M. Bolic, *Architectures for efficient implementation of particle filters*, New York, USA: State University of New York at Stony Brook, 2004.

[6] F. Gustafsson, F. Gunnarsson, N. Bergman, U. Forssell, J. Jansson, R. Karlsson and P. Nordlund, “Particle filters for positioning, navigation, and tracking,” *IEEE Transactions on Signal Processing,* vol. 50, no. 2, pp. 425-437, 2002.

[7] Q. Gan, J. P. Langlois and Y. Savaria, "Efficient Uniform Quantization Likelihood Evaluation for Particle Filters in Embedded Implementations," *accepted by Journal of Signal Processing Systems,* 29th Oct. 2013.

[8] Q. Gan, J. P. Langlois and Y. Savaria, "A reformulated systematic resampling and its parallel implementation on Application-Specific Instruction-set Processors," *accepted by MWSCAS*, 2013.

[9] Q. Gan, J. P. Langlois and Y. Savaria, "Parallel Systematic Resampling algorithm for particle filters," *submitted to Journal of Circuits, Systems and Signal Processing,* 10th Apr. 2013.

[10] Q. Gan, L. P. Langlois and Y. Savaria, "Parallel Array Histogram Architecture for

Embedded Implementations," *Electronics Letters,* vol. 49, no. 2, pp. 99-101, 2013.

[11] R. Farah, Q. Gan, J. M. P. Langlois, G. A. Bilodeau and Y. Savaria, "A tracking algorithm suitable for embedded systems implementation," *18th IEEE International Conference on Electronics, Circuits and Systems*, pp. 256-259, 2011.

[12] N. Metropolis and S. Ulam, "The Monte Carlo Method," *Journal of the American Statistical Association,* vol. 44, no. 247, pp. 335-341, 1949.

[13] N. Gordon, D. Salmond and C. Ewing, "Bayesian State Estimation for Tracking and Guidance Using the Bootstrap Filter," *Journal of Guidance Control and Dynamics,* vol. 18, no. 6, pp. 1434-1443, 1995.

[14] G. Kitagawa, "Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models," *Journal of Computational and Graphical Statistics,* vol. 5, pp. 1-25, 1996.

[15] J. Carpenter, P. Clifford and P. Fearnhead, "Improved particle filter for nonlinear problems," *IEE Proceedings Radar, Sonar and Navigation,* vol. 146, no. 1, pp. 2-7, 1999.

[16] M. S. Arulampalam, S. Naskell, N. Gordon and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking," *IEEE Transactions on Signal Processing,* vol. 50, no. 2, pp. 174-188, 2002.

[17] O. Cappe, S. J. Godsill and E. Moulines, "An overview of existing methods and recent advances in sequential Monte Carlo," *Proceedings of the IEEE,* vol. 95, no. 5, pp. 899-924, 2007.

[18] P. M. Djuric, J. H. Kotecha, J. Zhang, Y. Huang, T. Gbirmai, M. F. Bugallo and J. Miguez, "Particle filtering," *IEEE Signal Processing Magazine,* vol. 20, no. 5, pp. 19-38, 2003.

[19] A. Doucet, S. Godsill and C. Andrieu, "On sequential Monte Carlo sampling methods for Bayesian filtering," *Statistics and computing,* vol. 10, no. 3, pp. 197-208, 2000.

[20] J. V. Candy, "Bootstrap particle filtering," *IEEE Signal Processing Magazine,* vol. 24, no. 4, pp. 73-85, 2007.

[21] O. Aydogmus and M. Talu, "Comparison of Extended-Kalman- and Particle-Filter-Based

Sensorless Speed Control," *IEEE Transactions on Instrumentation and Measurement,* vol. 61, no. 2, pp. 402-410, 2012.

[22] M. Bolic, P. Djuric and S. Hong, "Resampling algorithms and architectures for distributed particle filters," *IEEE Transactions on Signal Processing,* vol. 53, no. 7, pp. 2442-2450, 2005.

[23] K. Hwang and W. Sung, "Load Balanced Resampling for Real-Time Particle Filtering on Graphics Processing Units," *IEEE Transactions on Signal Processing,* vol. 61, no. 2, pp. 411-419, 2013.

[24] S. Hong, Z. Shi, J. Chen and K. Chen, "Compact resampling algorithm and hardware architecture for particle filters," in *IEEE International Conference Communications Circuits and Systems*, vol. 2, pp. 886-890, 2008.

[25] S. Hong, Z. Shi, J. Chen and K. Chen, "A Low-Power Memory-Efficient Resampling Architecture for Particle Filters," *Journal of Circuits Systems and Signal Processing,* vol. 29, no. 1, pp. 155-167, 2010.

[26] J. S. Liu and R. Chen, "Sequential Monte Carlo Methods for Dynamic Systems," *Journal of the American Statistical Association,*vol. 93, pp. 1032-1044, 1998.

[27] H. A. A. El-Halym, I. I. Mahmoud and S. E.-D. Habib, "Proposed hardware architectures of particle filter for object tracking," *EURASIP Journal on Advances in Signal Processing,* vol. 2012, no. 1, pp. 1-19, 2012.

[28] S. Hong, M. Bolic and P. M. Djuric, "An Efficient Fixed-Point Implementation of Residual Resampling Scheme for High-Speed Particle Filters," *IEEE Signal Processing Letters,* vol. 11, no. 5, pp. 482-485, 2004.

[29] S. Hong and P. M. Djuric, "High-Throughput Scalable Parallel Resampling Mechanism for Effective Redistribution of Particles," *IEEE Transactions on Signal Processing,* vol. 54, no. 3, pp. 1144-1155, 2006.

[30] A. C. Sankaranarayanan, A. Srivastava and R. Chellappa, "Algorithmic and Architectural Optimizations for Computationally Efficient Particle Filtering," *IEEE Transaction on*

Image Processing, vol. 17, no. 5, pp. 737-748, 2008.

[31] L. Miao, J. J. Zhang, C. Chakrabarti and A. Papandreou-Suppappola, "A new parallel implementation for particle filters and its application to adaptive waveform design," *IEEE Workshop on Signal Processing Systems*, pp. 19-24, 2010.

[32] L. Miao, J. J. Zhang, C. Chakrabarti and A. Papandreou-Suppappola, "Algorithm and Parallel Implementation of Particle Filtering and its Use in Waveform-Agile Sensing," *Journal of Signal Process Systems,* vol. 65, no. 2, pp. 211-227, 2011.

[33] M. K. Pitt and N. Shephard, "Filtering via Simulation: Auxiliary Particle Filters," *Journal of the American Statistical Association,* vol. 94, no. 446, pp. 590-599, 1999.

[34] J. Kotecha and P. Djuric, "Gaussian particle filtering," *IEEE Transactions on Signal Processing,* vol. 51, no. 10, pp. 2592-2601, 2003.

[35] A. Doucet, N. de Freitas, K. Murphy and S. Russell, "Rao-blackwellised particle filtering for dynamic Bayesian networks," *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pp. 176-183, 2000.

[36] X. Xu and B. Li, "Rao-Blackwellised particle filter for tracking with application in visual surveillance," *2nd Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance*, pp. 17-24, 2005.

[37] C. Boucher and J.-C. Noyer, "A Hybrid Particle Approach for GNSS Applications with," *IEEE Transactions on Instrumentation and Measurement,* vol. 59, no. 3, pp. 498-505, 2010.

[38] J.-S. Kim, E. Serpedin and D.-R. Shin, "Improved Particle Filtering-Based Estimation of the Number of Competing Stations in IEEE 802.11 Networks," *IEEE Signal Processing Letters,* vol. 15, pp. 87-90, 2008.

[39] Q. Wen, J. Gao, A. Kosaka, H. Iwaki, K. Luby-Phelps and D. Mundy, "A particle filter framework using optimal importance function for protein molecules tracking," *IEEE International Conference on Image Processing*, pp.1161-1164, 2005.

[40] F. Benboujja, "Suivi automatique d'instruments dans les sequences d'images thoracoscopiques," *M.Sc.A. Dissertation*, Polytechnique Montreal, Canada, 2010.

[41] Y. Bar-Shalom, X. Li and T. Kirubarajan, *Estimation with Applications to Tracking and Navigation*, John Wiley, 2001.

[42] Y. Bar-Shalom and T. Fortmann, *Tracking and Data Association*, New York: Academic Press, 1988.

[43] Y. Bar-Shalom, *Multitarget-Multisensor Tracking: Applications and Advances*, Artech House, 1991.

[44] Y. Bar-Shalom and X. Li, *Estimation and Tracking: Principles, Techniques, and Software,* Artech House, 1993.

[45] X. Li and V. Jilkov, "A survey of maneuvering target tracking-part III: measurement models," *Proceedings of SPIE Conference on signal and data processing of small targets*, vol. 4473, pp. 423-446, 2001.

[46] R. Karlsson and F. Gustafsson, "Recursive Bayesian estimation: bearings-only applications," *IEEE Proceedings of Radar, Sonar and Navigation,* vol. 29, no. 1, pp. 305-313, 2005.

[47] M. S. Arulampalam, R. Ristic, N. Gordon and T. Mansell, "Bearings-only tracking of manoeuvring targets using particle filters," *Journal of Applied Signal processing,* vol. 15, pp. 2351-2365, 2004.

[48] M. S. Arulampalam and B. Ristic, "Comparison of the particle fillter with range-parameterized and modified polar EKFs for angle-only tracking," *Proceedings of SPIE, Signal and Data Processing of Small Target*, 2000.

[49] F. Gustafsson, “Particle Filter Theory and Practice with Positioning Applications,” *IEEE Aerospace and Electronic Systems Magazine,* vol. 25, no. 7, pp. 53-81, 2010.

[50] A. Hampapur, L. Brown, J. Connell, A. Ekin, N. Haas, M. Lu, H. Merkl, S. Pankanti, A. Senior, C. Shu and Y. Tian, "Smart video surveillance: exploring the concept of multiscale

spatiotemporal tracking," *IEEE Signal Processing Magazine,* vol. 22, no. 2, pp. 38-51, 2005.

[51] R. Yogesh, N. Vaswani, A. Tannenbaum and A. Yezzi, "Particle filtering for geometric active contours with application to tracking moving and deforming objects," *Computer Vision and Pattern Recognition,* vol. 2, pp. 2-9, 2005.

[52] M. Black and A. Jepson, "A Probabilistic Framework for Matching Temporal Trajectories: CONDENSATION-Based Recognition of Gestures and Expressions," in *Proceedings of the 5th European Conference on Computer Vision*, 1998.

[53] R. Chellappa and S. Zhou, "Face Tracking and Recognition from Video," *Handbook of Face Recognition*, New York, Springer, 2005.

[54] K. Okuma, A. Taleghani, N. Freitas, J. Little and D. Lowe, "A Boosted Particle Filter: Multitarget Detection and Tracking," *European Conference on Computer Vision*, vol. 3021, pp. 28-39, 2004.

[55] A. Yilmaz, O. Javed and M. Shah, “Object Tracking: A Survey,” *Journal ACM Computing Surveys,* vol. 38, no. 4, 2006.

[56] Z. Zivkovic and F. van der Heijden, "Improving the selection of feature points for tracking," *Pattern Analysis and Applications,* vol. 7, no. 2, pp. 144-150, 2004.

[57] K. Nummiaro, E. Koller-Meier and L. J. Van Gool, "Object Tracking with an Adaptive Color-Based Particle Filter," in *Proceedings of the 24th DAGM Symposium on Pattern Recognition*, 2002.

[58] P. Smith, T. Drummond and R. Cipolla, "Motion segmentation by tracking edge information over multiple frames," in *European Conference on Computer Vision*, pp. 396-410, 2000.

[59] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol.1, pp886-893, 2005.

[60] D. Comanuciu, V. Ramesh and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 25, no. 5, pp. 564-577, 2003.

[61] R. E. Kalman, "A new approach to linear filtering and prediction problem," *Journal of Basic Engineering Transactions,* vol. 82, pp. 34-45, 1960.

[62] M. Isard and A. Blake, "CONDENSATION—Conditional Density Propagation for Visual Tracking," *International Journal of Computer Vision,* vol. 29, no. 1, pp. 5-28, 1998.

[63] M. Isard and A. Blake, "Icondensation: Unifying low-level and high-level tracking in a stochastic framework," *European Conference on Computer Vision*, pp.893-908, 1998.

[64] J. Deutscher, A. Blake and I. Reid, "Articulated body motion capture by annealed particle filtering," *Computer Vision and Pattern Recognition,* vol. 2, pp. 126-133, 2000.

[65] P. Pérez, C. Hue, J. Vermaak and M. Gangnet, "Color-based probabilistic tracking," *Proceedings of the 7th European Conference on Computer Vision-Part I*, pp. 661-675, 2002.

[66] P. Perez, J. Vermaak and A. Blake, "Data fusion for visual tracking with particles," *Proceedings of the IEEE,* vol. 92, no. 3, pp. 495-513, 2004.

[67] P. Dunne and B. Matuszewski, "Choice of similarity measure, likelihood function and parameters for histogram based particle filter tracking in CCTV grey scale video," *Image and Vision Computing,* vol. 29, pp. 178-189, 2011.

[68] G. De Micheli, R. Ernst and W. Wolf, *Readings in Hardware/Software Codesign,* San Mateo, CA: Morgan Kaufmann, 2001.

[69] K. Keutzer, S. Malik and A. Newton, "From ASIC to ASIP: the next design discontinuity," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 84-90, 2002.

[70] M. Imai, Y. Takeuchi, K. Sakanushi and N. Ishiura, "Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP)," *IPSJ Transactions on*

System LSI Design Methodology, vol. 3, pp. 161-178, 2010.

[71] A. Hoffman and A. Nohl, "The Dusk of ASIC, the Dawn of ASIP," *Coware company white paper*, 2006.

[72] M. Bolic, A. Athalye, P. Djuric and S. Hong, “Algorithmic modification of particle filters for hardware implementation,” in *Proceedings of the European Signal Processing Conference*, Vienna, Austria, pp. 1641-1644, 2004.

[73] S. Hong, S. Chin, P. Djuric and M. Bolic, “Design and implementation of flexible resampling mechanism for high-speed parallel particle filters,” *Journal of VLSI Signal Processing,* vol. 44, no. 1-2, pp. 47-62, 2006.

[74] A. Athalye, M. Bolic, S. Hong and P. Djuric, “Generic Hardware Architectures for Sampling and Resampling in Particle Filters,” *EURASIP Journal on Applied Signal Processing,* vol. 17, pp. 2888-2902, 2005.

[75] H. A. A. El-Halym, I. I. Mahmoud and S. E.-D. Habib, "Efficient hardware architecture for particle filter based object tracking," *Proceedings of 2010 IEEE 17th International Conference on Image Processing*, Hong Kong, pp. 4497-4500, 2010.

[76] M. Sadasivam and S. Hong, "Application specific coarse-grained FPGA for processing element in real time parallel particle filters," *Proceedings of 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, pp.116-119, 2003.

[77] R. Velmurugan, V. Cevher and J. McClellan, "Implementation of Batch-Based Particle Filters for Multi-Sensor Tracking," *2nd IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing*, pp. 257-260, 2007.

[78] R. Velmurugan, S. Subramanian, V. Cevher, J. McClellan and D. Anderson, "Mixed-mode Implementation of Particle Filters," *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 617-620, 2007.

[79] B. Minch, "Construction and transformation of multiple-input translinear element networks," *IEEE Transaction Circuits Systems I,* vol. 50, pp. 1530-1537, 2003.

[80] S. Maskell, B. Alun-Jones and M. Macleod, "A Single Instruction Multiple Data Particle Filter," *IEEE Workshop on Nonlinear Statistical Signal Processing*, pp. 51-54, 2006.

[81] G. Hendeby, R. Karlsson and F. Gustafsson, “Particle Filtering: The Need for Speed,” *Eurasip Journal on Advances in Signal Processing,* vol. 2010, No. 22, 2010.

[82] H. Medeiros, X. Gao, J. Park, R. Kleihorst and A. Kak, "A parallel implementation of the color-based particle filter for object tracking," *Proceedings of the ACM Sensys Workshop on Applications, Systems and Algorithms for Image Sensing*, 2008.

[83] H. Medeiros, J. Park and A. Kak, "A parallel color-based particle filter for object tracking," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp.1-8 2009.

[84] H. Medeiros, G. Holguin, P. Shin and J. Park, "A parallel histogram-based particle filter for object tracking on SIMD-based smart cameras," *Journal of Computer Vision and Image,* vol. 114, no. 11, pp. 1264-1272, 2010.

[85] G. Li, B. Li, Z. Liu and X. Chen, "Implementation and optimization of Particle Filtertracking algorithm on Multi-DSPs system," *IEEE Conference on Cybernetics and Intelligent Systems*, pp.152-157, 2008.

[86] M. Gries and K. Keutzer, *Building ASIPs: The MESCAL Methodology*, Springer, 2005.

[87] M. Jain, M. Balakrishnan and A. Kumar, "ASIP design methodologies: survey and issues," *Fourteenth International Conference on VLSI Design*, pp.76-81, 2001.

[88] L. Pozzi and P. Paulin, "A Future of Customizable Processors: Are We There Yet?," *Design, Automation & Test in Europe Conference*, pp.1-2, 2007.

[89] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*, Morgan Kaufmann, 2006.

[90] A. Hoffmann, H. Meyr and R. Leupers, *Architecture Exploration for Embedded Processors With LISA*, Springer, 2002.

[91] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt and A. Nicolau, "EXPRESSION: a

language for architecture exploration through compiler/simulator retargetability," *Design, Automation and Test in Europe Conference and Exhibition*, pp. 31-45, 1999.

[92] A. Fauth, J. Van Praet and M. Freericks, "Describing instruction set processors using nML," *European Design and Test Conference*, pp. 503-507, 1995.

[93] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takenchi, A. Kitajima and M. Imai, "PEAS-III: an ASIP design environment," *Proceeding of International Conference Computer Design*, pp. 430-436, 2000.

[94] A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takenchi and M. Imai, "Effectiveness of the ASIP design system PEAS-III in design of pipelined processors," in *Proceeding of Asia South Pacific Design Automation Conference*, pp.649-654, 2001.

[95] S. Leibson, *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores*, San Francisco: Elsevier Morgan-Kaufmann, 2006.

[96] P. Ienne, L. Pozzi and M. Vuletic, "On the limits of automatic processor specialisation by mapping dataflow sections on ad-hoc functional units," *Technical Report 01/376, Swiss Federal Institute of Technology Processor Architecture Lab*, Lausanne, 2001.

[97] P. Yu and T. Mitra, "Characterizing embedded applications for instruction-set extensible processors," *Proceeding of Design Automation Conference*, pp. 723-728, 2004.

[98] K. Atasu, L. Pozzi and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," *Proceedings of the 40th Design Automation Conference*, pp. 411-428, 2003.

[99] K. Atasu, G. Dündar and C. Ozturan, "An integer linear programming approach for identifying instruction-set extensions," *Proceedings of the International Conference on Hardware - Software Codesign and System Synthesis*, pp. 172-177, 2005.

[100] K. Atasu, R. Dimond, O. Mencer, W. Luk, C. Ozturan and G. Dundar, "Optimizing instruction-set extensible processors under data bandwidth constraints," *Proceedings of Design,Automation and Test in Europe Conference*, pp.1-6, 2007.

[101] K. Atasu, C. Ozturan, G. Dundar, O. Mencer and W. Luk, "CHIPS: Custom Hardware Instruction Processor Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 3, pp. 528-541, 2008.

[102] K. Atasu, O. Mencer, W. Luk, C. Ozturan and G. Dundar, "Fast Custom Instruction Identification by Convex Subgraph Enumeration," *Proceedings of the 19th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp.1-6, 2008.

[103] L. Pozzi, K. Atasu and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 7, pp. 1209-1229, 2006.

[104] P. Bonzini and L. Pozzi, "Polynomial-time subgraph enumeration for automated instruction set extension," *Proceedings of DATE*, pp. 1331-1336, 2007.

[105] D. E. Culler, J. P. Singh and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers Inc., 1999.

[106] H. Noori, F. Mehdipour, K. Murakami, K. Inoue and M. Zamani, “An architecture framework for an adaptive extensible processor,” *Journal of Supercomputing,* vol. 45, no. 3, pp. 313-340, 2008.

[107] Tensilica Inc., “Xtensa LX microprocessor data book for Xtensa LX2 processor cores,” 2007.

[108] Tensilica Inc., “Tensilica Instruction Extension (TIE) Language User’s Guide,” 2002.

[109] B. Anderson and J. Moore, *Optimal Filtering*, New Jersey, USA: Prentice-Hall Englewood Cliffs, 1979.

[110] M. M. Mbaye, N. Belanger, Y. Savaria and P. Samuel, "Loop Acceleration Exploration for ASIP Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 20, no. 4, pp. 684-696, 2012.

[111] M. Harris, S. Sengupta and J. Owens, "Parallel prefix sum (scan) with CUDA," *GPU*

Gems, vol. 3, no. 39, pp. 851-876, 2007.

[112] B. Parhami, *Computer arithmetic: Algorithms and hardware designs*, Oxford University Press, 2000.

[113] A. Shahbahrami,, B. Juurlink and S. Vassiliadis, "SIMD vectorization of histogram functions," *Proceedings of ASAP*, pp. 174-179, 2007.

[114] S. Muller, "A new programmable VLSI architecture for histogram and statistics computation in different windows," *Proceedings of ICIP*, vol.1, pp. 73-76, 1995.

[115] A. Shahbahrami, J. Hur, B. Juulink and S. Wong, "FPGA implementation of parallel histogram computation," *2nd HiPEAC Workshop on Reconfigurable Computing*, pp. 63-72, 2008.

[116] E. Jamro, M. Wielgosz and K. Wiatr, "FPGA implementation of the strongly parallel histogram equalization," *Proceedings of IEEE Workshop on DDECS*, pp.1-6, 2007.

[117] J. Cadenas, R. Sherratt and P. Huerta, "Parallel pipelined histogram architectures," *Electronics Letters,* vol. 47, no. 20, pp. 1118-1120, 2011.

[118] J. Cadenas, R. Sherratt, P. Huerta and W. Kao, "Parallel pipelined array architectures for real-time histogram computation in consumer devices," *IEEE Transactions on Consumer Electronics,* vol. 57, no. 4, pp. 1460-1464, 2011.

[119] J. Czyz, B. Ristic and B. Macq, "A particle filter for joint detection and tracking of color objects," *Image and Vision Computing,* vol. 25, no. 8, p. 1271–1281, 2006.

[120] A. Verma and P. Ienne, "Automatic synthesis of compressor trees: re-evaluating large counters," *Proceedings of DATE*, pp. 443-448, 2007.