Imperial College London
Department of Computing

Optimising reconfigurable systems for real-time applications

Thomas Chun Pong Chau

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the Imperial College
April 2014

Abstract

This thesis proposes novel approaches to design and optimise reconfigurable systems targeting real-time applications. There are three main contributions. First, we propose novel data structures and memory architectures for accelerating real-time proximity queries, with potential application to robotic surgery. We optimise performance while maintaining accuracy by several techniques including mixed precision, function transformation and streaming data flow. Significant speedup is achieved using our reconfigurable accelerator platforms over double-precision CPU, GPU and FPGA designs. Second, we develop a run-time reconfiguration methodology for real-time particle filters. Based on workload over time, different configurations with various performance and power consumption tradeoffs are loaded onto the FPGAs dynamically. Promising energy reduction has been achieved in addition to speedup over CPU and GPU designs. The approach is evaluated in an application to robot localisation. Third, we explore an automated optimisation methodology for real-time sequential Monte Carlo methods. Machine learning algorithms are used to search for an optimal parameter set to produce the highest solution quality while satisfying all timing and resource constraints. The approach is evaluated in an application to air traffic management.

Acknowledgements

I would like to express (whatever feelings I have) to:

- My supervisor: Prof. Wayne Luk
- My second supervisor: Prof. Peter Y.K. Cheung
- Internal researchers: Dr. Ka-Wai Kwok, Dr. Kuen Hung Choi, Gary Chow, Xinyu Niu, Maciej Kurek
- Colleagues: James Arram, Dr. Tobias Becker, Dr. Brahim Betkaoui, Dr. Pavel Burovskiy, Gary Chow, Dr. Bridgette Cooper, Kit Cheung, Dr. Gabriel De Figueiredo Coutinho Stewart Denholm, Paul Grigoras, Ce Guo, Liucheng Guo, Dr. Eddie Hung, Gordon Inggs, Dr. Qiwei Jin, Dr. Andrien Le Masle, Nicholas Ng, Shengjia Shao, Dr. Timothy Todman, Dr. Anson Tse, Jinzhe Yang,
- External collaborators: Dr. Alison Eele, Dr. Benjamin Core, Dr. Kathryn Cobden, Prof. Philip Leong
- Project students: James Targett, Marlon Wijeyasinghe, Jake Humphrey, Georgios Skouroupathis
- ARM internship: William Wang
- Scholarship: Croucher Foundation
- Others: UK EPSRC, FP7 EPiCS and FASTER projects, Maxeler Technologies, Altera and Xilinx

Dedication

To my parents for making me be who I am, and giving me the best education you could;

To my wife, Kaijia, for her patience, understanding and nice food during my hours of research, contemplation and writing.

Contents

Abstract **i**

Acknowledgements **iii**

Dedication **v**

1 Introduction **1**

1.1 Motivation and Objectives 1

1.2 Contributions 1

1.3 Statement of Originality 1

1.4 Publications 1

2 Background Theory **3**

2.1 Introduction 3

3 Functional Transformation & Precision Optimisation for PQ Process **4**

3.1 Introduction 4

3.2 Overview 6

3.3 Formulation of PQ 9

3.4 Optimisation for Reconfigurable Hardware 12

3.4.1 Transformation of Trigonometric and Search Functions 13

3.4.2 Precision Optimisation 13

3.4.3 Dynamic Optimisation 15

3.5 Reconfigurable System Design 16

3.5.1 Streaming Data Structure 16

3.5.2 System Architecture 17

3.5.3 Performance Estimation 20

3.6 Experimental Evaluation 21

3.6.1 General Settings 21

3.6.2 Parallelism versus Precision 22

3.6.3 Ratio of Re-computation versus Precision 22

3.6.4 Comparison: CPU, GPU and FPGA 24

3.7 Summary 25

4 Self-adaption & Run-time Reconfiguration for SMC Applications 27

4.1 Introduction 27

4.2 Overview 28

4.3 Adaptive Particle Filter 30

4.3.1 Stage 1: Sampling and Importance Weighting (line 8 to 9) 30

4.3.2 Stage 2: Lower Bound Calculation (line 10) 32

4.3.3 Stage 3: Particle set size tuning (line 12 to 34) 33
4.3.4 Stage 4: Resampling (line 35) 34
4.4 Heterogeneous Reconfigurable System 34
4.4.1 Mapping adaptive PF to HRS 35
4.4.2 FPGA Kernel Design 35
4.4.3 Timing model for run time reconfiguration 37
4.5 Optimising Transfer of Particle Stream 39
4.6 Experimental Results 42
4.6.1 System Settings 43
4.6.2 Adaptive PF versus Non-adaptive PF 44
4.6.3 Data Compression 46
4.6.4 Performance comparison of HRS, CPUs and GPUs 46
4.7 Summary 49

5 Design Flow for SMC Applications 51

5.1 Introduction 51
5.2 Overview 53
5.2.1 SMC Methods 53
5.2.2 SMC Applications 54
5.3 SMC Design Flow 57
5.3.1 Specifying Application Features 58
5.3.2 Computation Engine Design 59

5.3.3 Performance Model 63

5.4 Optimising SMC Computation Engine 65

5.4.1 Effect of the Design Parameters 65

5.4.2 Parameter Optimisation 66

5.5 Evaluation 68

5.5.1 Design Productivity 68

5.5.2 Application 1: Stochastic Volatility 68

5.5.3 Application 2: Mobile Robot Localisation 69

5.5.4 Application 3: Air Traffic Management 70

5.6 Summary 71

6 Conclusion **73**

6.1 Summary of Thesis Achievements 73

6.2 Applications 73

6.3 Future Work 73

Bibliography **73**

List of Tables

3.1 Parameters of the performance model 21

3.2 Comparison of PQ computation in 1 ms using CPU-based system (CPU), GPU-based system (GPU), double precision FPGA-based system (FPGA DP) and FPGA+CPU system with reduced precision (FPGA RP) 25

4.1 Comparison of adaptive and non-adaptive PF on HRS (MaxWorkstation with one FPGA, no data compression is applied) 45

4.2 Performance comparison of HRS, CPUs and GPU 48

5.1 SMC design parameters. Dynamic: adjustable at run-time; static: fixed at compile-time. 54

5.2 Lines of code for 3 SMC applications under the proposed design flow. 68

5.3 Performance comparison of stochastic volatility. 69

5.4 Performance comparison of robot localisation. 70

5.5 Parameter optimisation of air traffic management system using machine learning approach. 70

5.6 Performance comparison of air traffic management. 71

List of Figures

3.1 (Left) Various sets of vertex points aligned on a series of contours; (Right) A set of vertex points located on an arbitrary form of mesh. 7

3.2 (a) A virtual tube (in green) bounded by a series of contour (in red) denotes the configuration of an endoscope; (b) The corresponding three-dimensional distance map in grids of 86x48x43. 8

3.3 Data structure: N_S points are processed in a group. Each point of a group is iterated for N_C times. Data are streamed in an order as indicated by the arrows. 17

3.4 System architecture: Solid lines represent communication on the FPGA board while dotted lines represent the bus connecting the reduced precision data-path on FPGA to the high precision data-path on CPU. 17

3.5 Memory array stores contour indices for re-computation. 19

3.6 Computation time [dotted line] and the level of parallelism [solid line] vs. different number of mantissa bits. 23

3.7 Ratio of re-computation [dotted line] and the number of points processed in 1 ms [solid line] vs. different number of mantissa bits. 24

3.8 Computation time for a PQ update with 100 contours vs. the number of points. 26

4.1 Particle set reduction 34

4.2 Heterogeneous reconfigurable system (Solid lines: data paths; Dotted lines: control paths) 35

4.3 A particle stream 36

4.4 FPGA kernel design 37

4.5 Power consumption of the HRS over time 40

4.6 After the resampling process, some particles are eliminated and the remaining particles are replicated. Data compression is applied so that every particle is stored and transferred once only. 41

4.7 Number of particles and components of total computation time versus wall-clock time 45

4.8 Localisation error versus wall-clock time 46

4.9 Effect on the data transfer time by particle stream compression 47

4.10 Power consumption of HRS, CPU and GPU in one time-step, notice that the computation time of the CPU system exceeds the 5-second real-time requirement (The lines of HRS(2) and HRS(3) overlap) 49

4.11 Run-time vs. energy consumption of HRS, CPUs and GPUs (5-second time-step, 67M particles; Refer to Table II for system settings) 50

5.1 Design flow for SMC applications. Users only customise the application-specific descriptions inside the dotted box. 57

5.2 (a) Design of the SMC computation engine. Solid lines represent data paths while dotted lines represent control paths; (b) Data structure of particles represented by 3 data streams. 61

5.3 FPGA kernel design. The blocks that require users' customisation are darkened. The dotted box covers the blocks that are optional on FPGAs. 62

5.4 Parameter space of robot localisation system (N_A=8192, S=1). The dark region on the top-right indicates designs which fail localisation accuracy constraints, while those on the bottom-left indicates designs which fail real-time requirements. 66

5.5 Illustration of automatic parameter optimisation: (a) Sampling parameter sets; (b) Building surrogate model; (c) Calculating expected improvement; (d) Moving to the point offering the highest improvement. 67

Chapter 1

Introduction

1.1 Motivation and Objectives

Motivation and Objectives here.

1.2 Contributions

Contributions here.

1.3 Statement of Originality

Statement here. [1] [2] [3]

1.4 Publications

[1] T. C. P. Chau, W. Luk, P. Y. K. Cheung, A. Eele, and J. M. Maciejowski, "Adaptive sequential Monte Carlo approach for real-time applications," in *Proceedings of International Conference Field Programmable Logic and Applications*, 2012, pp. 527–530.

[2] T. C. P. Chau, X. Niu, A. Eele, W. Luk, P. Y. K. Cheung, and J. M. Maciejowski, “Heterogeneous reconfigurable system for adaptive particle filters in real-time applications,” in *Proceedings of International Symposium Applied Reconfigurable Computing*, 2013, pp. 1–12.

[3] T. C. P. Chau, J. S. Targett, M. Wijeyasinghe, W. Luk, P. Y. K. Cheung, B. Cope, A. Eele, and J. M. Maciejowski, “Accelerating sequential Monte Carlo method for real-time air traffic management,” *SIGARCH Computer Architecture News*, vol. 41, no. 5, 2013.

[4] A. Eele, J. M. Maciejowski, T. C. P. Chau, and W. Luk, “Parallelisation of sequential Monte Carlo for real-time control in air traffic management,” in *Proceedings of International Conference Decision and Control*, 2013.

[5] M. Kurek, T. Becker, T. C. P. Chau, and W. Luk, “Automating optimization of reconfigurable designs,” in *Proceedings of International Symposium Field-Programmable Custom Computing Machines*, 2014.

Chapter 2

Background Theory

2.1 Introduction

Text of the Background.

Chapter 3

Functional Transformation and Precision Optimisation for Proximity Query Process

3.1 Introduction

Advanced surgical robots support image guidance and haptic (force-based) feedback for effective navigation of surgical instruments. Such image-guided robots rely on computing in real-time the intersection or the closest point-pair between two objects in three-dimensional space; this computation is known as Proximity Query (PQ).

PQ has been widely studied in areas such as robot motion planning, haptics rendering, virtual prototyping, computer graphics, and animation [4]. Robot motion planning is particularly demanding for the real-time performance of PQ [5]. In the past decade, PQ has also been used as a key task for Active Constraints [6] or Virtual Fixtures [7], a collaborative control strategy mostly applied in image-guided surgical robotics. The clinical potential of this control strategy has been demonstrated by imposing haptic feedback [8] on instrument manipulation based on imaging data [9]. This haptic feedback provides the operator with kinaesthetic perception for sensing positions, velocities, forces, constraints and inertia associated with direct maneuvering of surgical instrument within the target anatomy.

As mentioned above, fast and efficient PQ is a pre-requisite for effective navigation through access routes to the target anatomy [6]. This haptic guidance, rendered based on imaging data, can enable a distinct awareness of the position of the surgical device relative to the target anatomy so as to prevent the operator from feeling disoriented within the surrounding organs. Such disorientation could potentially cause unnoticed major organ damage. This guidance is particularly important during soft tissue surgery, which involves large-scale and rapid tissue deformations. A high update frequency above 1 kHz is required to maintain smooth and steady manipulation guidance. Due to its intrinsic complexity and this real-time requirement, PQ is computationally challenging. Various approaches have been proposed to achieve the required update rate [5, 10], with objects represented in specific formats such as spheres, torus or convex surfaces. The only attempts that apply PQ to haptic rendering, while considering explicitly the interaction of the body with the surrounding anatomical regions, involve modelling the anatomical pathway or the robotic device as a tubular structure [7, 11]. The computation burden is increased by the need to compute the placement of anatomical model relative to the robot whose shape is represented by more than 1 million vertices.

Due to its compute-intensive nature, PQ can greatly benefit from hardware acceleration. However, the massive amount of floating-point computations constitute a long data-path which is resource-demanding. Even if we could implement the data-path in an FPGA, the acceleration would be restricted by low parallelism and clock frequency. This challenge limits the implementation of PQ on an FPGA.

In this paper, we derive a PQ formulation which allows objects to be represented in complex geometry with vertices. To leverage the advantages of FPGAs, function transformation eliminates iterative trigonometric functions such that the algorithm can be fully-pipelined. We increase data-path parallelism by adopting a reduced precision data format which consumes fewer logic resources than high precision. To maintain the accuracy of results, potential incorrect outputs are re-computed in high precision. We design a novel memory architecture for buffering potential outputs and maintaining streaming data-flow. We further exploit the run-time reconfigurability of FPGA to optimise precision dynamically. To the best of our knowledge, our work is the first to apply reconfigurable technology to narrow-phase PQ computation.

The contributions of this paper are as follows:

- A PQ formulation for calculating the relative placement of objects modelled by vertices with complex morphology, which facilitates restructuring of trigonometric and search functions to be amenable to parallel implementation in hardware.
- Enhanced parallelism by treating input points as a novel data structure propagating through pipelines, together with FPGA-specific optimisations such as adapting PQ to reduced precision arithmetic, supporting multiple precisions in a novel memory architecture, and automating precision management with run-time reconfiguration.
- Implementation in a reconfigurable platform with four FPGAs which is shown to be 478 times faster than a single-core CPU, 58 times faster than a 12-core CPU system, 9 times faster than a GPU, and 3 times faster than a 4-FPGA system implemented in double precision.

The rest of the paper is organised as follows. Section 5.2 briefly describes the background of PQ formulation and review bit-width optimisation techniques. Section 3.3 presents our proposed PQ formulation. Section 5.4 discusses the optimisation of PQ for reconfigurable accelerator. Section 3.5 describes the system design that maps PQ to a reconfigurable accelerator. Section 5.5 provides experimental results and Section 5.6 concludes our work.

3.2 Overview

In this section, we first provide a brief introduction to PQ. Then we review the bit-width optimisation techniques that inspired our research.

Fig. 3.1 illustrates two objects acting as inputs to the proposed PQ. The object shown on the left is bounded by a series of contours (cf. Definition 1), each of which is outlined by a set of vertex points. This object can be either a luminal anatomy or a robotic endoscope/catheter. On the right, the mesh comprises vertex points which represent the morphological structure of either the robot or the target anatomy in complex shape. The proposed PQ actually computes how much the mesh deviates beyond the volumetric pathway bounded along the contours.

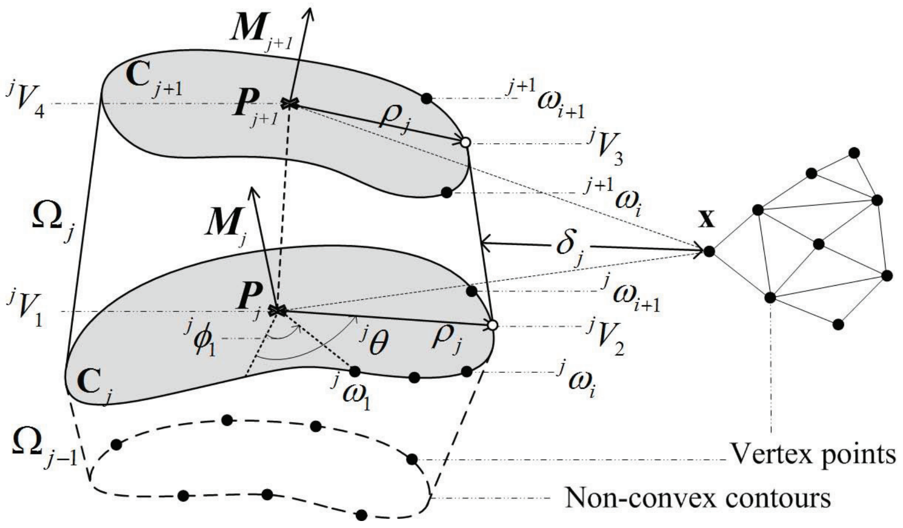
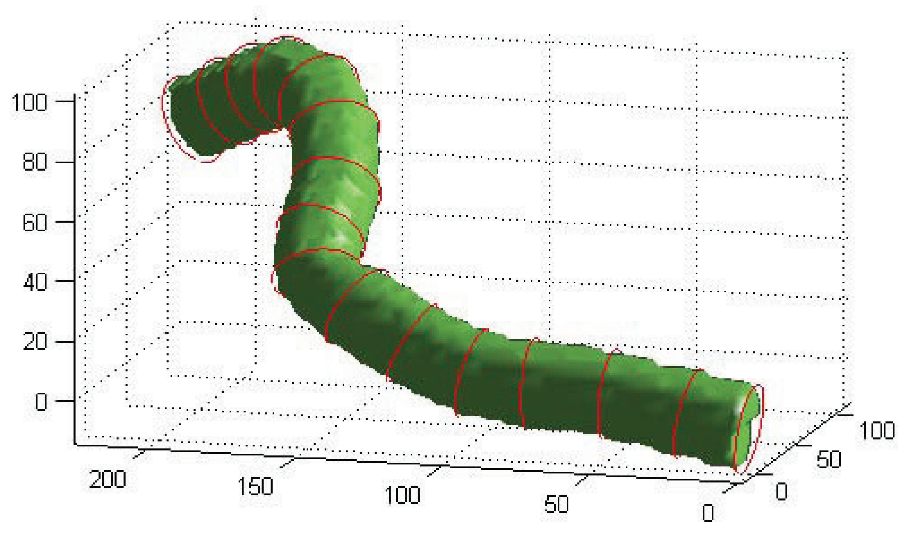


Figure 3.1: (Left) Various sets of vertex points aligned on a series of contours; (Right) A set of vertex points located on an arbitrary form of mesh.

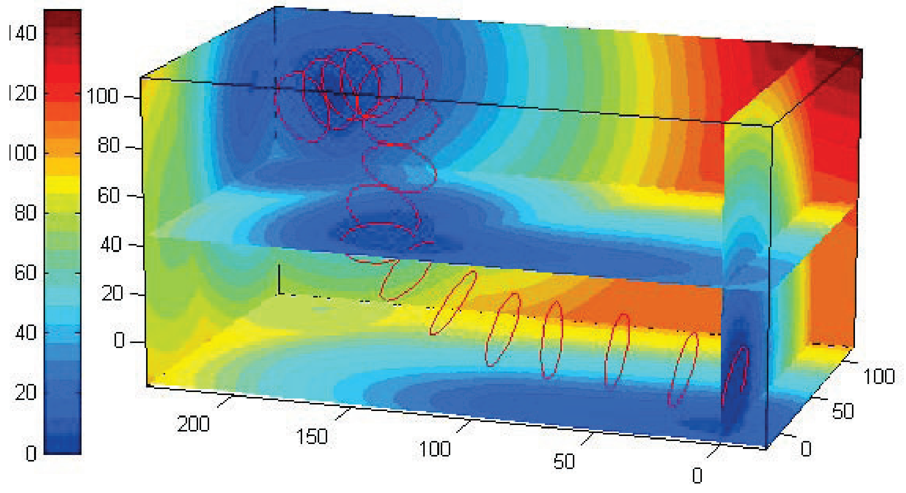
As shown in Fig. 3.2(a), a series of circular contours fitted along a part of an endoscope, which passes through the rectum up to the sigmoid colon. These contours form a constraint pathway. Fig. 3.2(b) shows a distance map in three-dimensional space with 177k grid points. Distance from every grid point to the endoscope is computed by the proposed PQ. The warmer colour, the further the point is located beyond the endoscope.

Definition 1. *Each contour is denoted by* C_j, $\forall j \in [1, ..., N_C]$. *A single segment* Ω_j *comprises two adjacent contours* C_j *and* C_{j+1}. P_j *is the centre of the contour* C_j. M_j *is the tangent of centre line of contour* C_j. ${}^j\omega_i = [{}^j\omega_{xi}, {}^j\omega_{yi}, {}^j\omega_{zi}]^T, (i = 1, ..., W)$ *are the contour vertices, where* W *is the number of vertex points outlining each contour.*

There has been previous work on hardware acceleration of board-phase PQ, which involves detecting collisions between primitive objects, e.g. spheres [10] or boxes [12]. Such an object



(a)



(b)

Figure 3.2: (a) A virtual tube (in green) bounded by a series of contour (in red) denotes the configuration of an endoscope; (b) The corresponding three-dimensional distance map in grids of 86x48x43.

can be a bounding volume tightly containing a union of multiple complex-shaped objects. On FPGA, the most relevant work is covered by Chow el at. [13]; however, to the narrow-phase PQ which computes the further detailed information, for instance, the shortest distance or penetration depth between polyhedra, GJK [14], V-Clip [15] and Lin-Canny [16] are the few well-established approaches, but their hardware acceleration is difficult due to algorithmic complexity. There is, thus far, no attempt of using FPGA. Such approaches are also restricted to the object represented in convex polyhedra. To this end, we have proposed a PQ approach for complex-morphology object [11] but how it can be incorporated with FPGA is not elaborated.

To leverage the advantages of FPGAs for hardware acceleration, Chow et al. [13] proposed a mixed precision methodology. They assume the data-path is short such that both the reduced precision and high precision implementations can be fitted in an FPGA. For complicated applications where the level of parallelism is limited by FPGA resource, their approach is not applicable. There are other studies about bit-width optimisation which uses minimum precision in a data-path given a required output accuracy. Examples include interval arithmetic [17], affine arithmetic [18, 19] and polynomial algebraic approach [20]. However, a reduction of precision in any stage within a data-path will result in a loss in output accuracy which is uncorrectable. They require using accuracy models to relate output accuracy with the precisions of data-path. Our work is different from these work by deriving an automatic way to find an optimal precision using run-time reconfiguration.

3.3 Formulation of PQ

In this section, we derive our modified PQ process which was originally proposed in our previous work [11]. The significance of this modification is to formulate the PQ capable of processing the contours in complex shapes. As a result, it allows the analytical measure of the shortest Euclidean distance between an arbitrary set of vertices and a series of segments Ω_j (cf. Definition 1) which has been a well-known representation of a complex three-dimensional object [21]. Each segment is enclosed by two adjacent contours which are outlined by vertices arranged in polar coordinates; hence, it outperforms the existing narrow-phase PQs which are only compatible with convex objects.

In consideration of the point-to-segment distance, as shown in Fig. 3.1, four steps are taken to calculate the shortest distance δ_j between a point $\boldsymbol{x}$ and the corresponding edge ${}^jV_2 \rightarrow^j V_3$.

Before these steps, we capture the computation using polar coordinates. Given a contour C_j, ${}^j\phi_i$ are the polar angles corresponding to each contour vertex ${}^j\omega_i$. The polar angles of all the ${}^j\omega_i$ along the contour have to be computed. This computation can be further simplified by ignoring an axis coordinate. The poles and the contour vertices are then projected either on

X-Y, Y-Z or X-Z plane based on the following conditions:

$$\begin{aligned}
&\text{if } |M_{zj}| = \max\left(|M_{xj}|, |M_{yj}|, |M_{zj}|\right) \\
&\quad {}^j\omega'_i = [{}^j\omega_{1i}, {}^j\omega_{2i}]^T = [{}^j\omega_{xi}, {}^j\omega_{yi}]^T, P'_j = [P_{xj}, P_{yj}]^T \\
&\text{if } |M_{xj}| = \max\left(|M_{xj}|, |M_{yj}|, |M_{zj}|\right) \\
&\quad {}^j\omega'_i = [{}^j\omega_{1i}, {}^j\omega_{2i}]^T = [{}^j\omega_{yi}, {}^j\omega_{zi}]^T, P'_j = [P_{yj}, P_{zj}]^T \\
&\text{if } |M_{yj}| = \max\left(|M_{xj}|, |M_{yj}|, |M_{zj}|\right) \\
&\quad {}^j\omega'_i = [{}^j\omega_{1i}, {}^j\omega_{2i}]^T = [{}^j\omega_{zi}, {}^j\omega_{xi}]^T, P'_j = [P_{zj}, P_{xj}]^T
\end{aligned} \tag{3.1}$$

Then ${}^j\phi_i$ is calculated as follows:

$$\overline{{}^j\omega'_i} = {}^j\omega'_i - P'_j \ , \qquad {}^j\phi_i = atan2\left(\overline{{}^j\omega_{2i}}, \overline{{}^j\omega_{1i}}\right) \tag{3.2}$$

We will explain the details of $atan2$ is Section 3.4.1.

Step 1: Find the normal of a plane containing points $\boldsymbol{x}$, P_j and P_{j+1}. The symbol $\times$ denotes a cross product of two vectors in three-dimensional space.

$$n_j = (P_j - x) \times (P_{j+1} - x) \tag{3.3}$$

Step 2: Calculate vectors ρ_j and ρ_{j+1} which are respectively perpendicular to tangents M_j and M_{j+1} and are both parallel to the plane with normal n_j.

$$\rho_j = n_j \times M_j \ , \qquad \rho_{j+1} = n_j \times M_{j+1} \tag{3.4}$$

Step 3: Determine a 4-vertex polygon outlined by ${}^jV_{i=1...4} \in \Re^{3\times 1}$ which is a part of the cross-section of segment Ω_j. This section is cut by a plane containing the point $\boldsymbol{x}$ and the line segment $P_j \rightarrow P_{j+1}$.

$$\begin{aligned} {}^jV_1 &= P_j & {}^jV_2 &= P_j + t_j \cdot \rho_j \\ {}^jV_4 &= P_{j+1} & {}^jV_3 &= P_{j+1} + t_{j+1} \cdot \rho_{j+1} \end{aligned} \tag{3.5}$$

At this stage, we need to calculate t_j and t_{j+1}. This can be achieved by mapping the values of ρ_j to a two-dimensional plane.

$$\begin{aligned} &\text{if } |M_{zj}| = \max\left(|M_{xj}|, |M_{yj}|, |M_{zj}|\right) \\ &\quad \rho'_j = [\rho_{1j}, \rho_{2j}]^T = [\rho_{xj}, \rho_{yj}]^T \\ &\text{if } |M_{xj}| = \max\left(|M_{xj}|, |M_{yj}|, |M_{zj}|\right) \\ &\quad \rho'_j = [\rho_{1j}, \rho_{2j}]^T = [\rho_{yj}, \rho_{zj}]^T \\ &\text{if } |M_{yj}| = \max\left(|M_{xj}|, |M_{yj}|, |M_{zj}|\right) \\ &\quad \rho'_j = [\rho_{1j}, \rho_{2j}]^T = [\rho_{zj}, \rho_{xj}]^T \end{aligned} \tag{3.6}$$

Then we calculate ${}^j\theta$, the corresponding polar angle of ρ'_j by Equation 3.7.

$${}^j\theta = atan2\left(\rho_{2j}, \rho_{1j}\right) \tag{3.7}$$

A search is performed to find ${}^j\phi_i$ and ${}^j\phi_{i+1}$ which embrace ${}^j\theta$. The polar angles ${}^j\phi_i$ and ${}^j\phi_{i+1}$ are calculated from Equation 3.2.

Based on the value i obtained from the search, t_j is calculated.

$$\begin{aligned} a &= [(P_j - {}^j\omega_i)({}^j\omega_{i+1} - {}^j\omega_i)][({}^j\omega_{i+1} - {}^j\omega_i)\rho] \\ b &= [(P_j - {}^j\omega_i)\rho]\|{}^j\omega_{i+1} - {}^j\omega_i\|^2 \\ c &= \|\rho\|^2\|{}^j\omega_{i+1} - {}^j\omega_i\|^2 - \|({}^j\omega_{i+1} - {}^j\omega_i)\rho\|^2 \\ t_j &= \frac{a-b}{c} \end{aligned} \tag{3.8}$$

Step 4: Define the shortest distance to be zero if the point $\boldsymbol{x}$ lies inside the polygon ${}^jV_{i=1...4}$ on the same plane. Referring to [22], it can be determined by three variables $\lambda_{i=1,...,3}$ calculated as follows:

$$\begin{aligned} \lambda_i &= n_j \cdot \psi_i, i = 1, ..., 3 \\ \text{s.t. } \psi_i &= ({}^jV_i - x) \times ({}^jV_{i+1} - x). \end{aligned} \tag{3.9}$$

Here n_j denotes the normal defined in Equation 3.3 and ψ_i denotes the normal of the plane containing ${}^jV_{i=1...4}$. For all $\lambda_{i=1,...,3} \geq 0$, the shortest distance δ_j from point x to the segment Ω_j is assigned to zero such that $\delta_j(x) = 0$. Otherwise $\delta_j(x)$ will be considered as the distance from the point $\boldsymbol{x}$ to the line segment ${}^jV_2 \rightarrow^j V_3$. Such a point-line distance in three-dimensional space can be calculated simply referring to [23].

In consideration of many points and segments, Equation 3.10 generally expresses the deviation in distance from a single coordinate x_i to a series of constraint segments $(\Omega_1, ..., \Omega_{N_C-1})$, where $i = 1, ..., N_P$ and N_P is the total number of vertex points belong to the mesh model and $N_C - 1$ is the number of segments involved in the calculation.

$${}_i\delta_{N_C-1} = \min\left(\delta_1(\boldsymbol{x}_i), \delta_2(\boldsymbol{x}_i), ..., \delta_{N_C-1}(\boldsymbol{x}_i)\right) \tag{3.10}$$

The point with the maximum deviation, also known as penetration depth, is obtained below.

$$d^{N_C-1} = \max_{i=1,...,N_P} \left({}_i\delta_{N_C-1}(\boldsymbol{x}_i)\right) \tag{3.11}$$

3.4 Optimisation for Reconfigurable Hardware

The PQ formulation sketched in the previous section is not entirely hardware-friendly. In this section we discuss several techniques to allow PQ to benefit from FPGA technology.

3.4.1 Transformation of Trigonometric and Search Functions

The search process in step 3 of PQ checks whether ${}^{j}\phi_i \leq {}^{j}\theta$.

$$ {}^{j}\phi_i = atan2\left(\overline{{}^{j}\omega_{2i}}, \overline{{}^{j}\omega_{1i}}\right) \ , \ {}^{j}\theta = atan2\left(\rho_{2j}, \rho_{1j}\right) \tag{3.12}$$

$atan2(a, b)$ is not a hardware-friendly operator. It requires the calculation of $tan^{-1}(a, b)$ and then determines the appropriate quadrant of the computed angle based on the signs of a and b. $tan^{-1}(a, b)$ is expensive and is often not available in FPGA libraries, therefore, we transform Equation 3.12 to another form as shown below:

$$\begin{aligned} {}^{j}\phi_i &= tan^{-1}\left(\frac{\overline{{}^{j}\omega_{2i}}}{\sqrt{\overline{{}^{j}\omega_{1i}}^2 + \overline{{}^{j}\omega_{2i}}^2} + \overline{{}^{j}\omega_{1i}}}\right) \\ {}^{j}\theta &= tan^{-1}\left(\frac{\rho_{2j}}{\sqrt{\rho_{1j}^2 + \rho_{2j}^2} + \rho_{1j}}\right) \end{aligned} \tag{3.13}$$

$atan2$ is transformed to tan^{-1} which is then cancelled out on both sides. As a result, the comparison becomes:

$$\frac{\overline{{}^{j}\omega_{2i}}}{\sqrt{\overline{{}^{j}\omega_{1i}}^2 + \overline{{}^{j}\omega_{2i}}^2} + \overline{{}^{j}\omega_{1i}}} \leq \frac{\rho_{2j}}{\sqrt{\rho_{1j}^2 + \rho_{2j}^2} + \rho_{1j}} \tag{3.14}$$

In this case, square root calculation is much easier to be mapped to hardware.

3.4.2 Precision Optimisation

Reduced precision data-paths consume less logic resource at the expense of lower accuracy of results. To benefit from reduced precision data-paths without compromising accuracy, we partition the computation into two data-paths:

- Reduced precision data-path: Compute the deviations based on Equation 3.3 to 3.10.
- High precision data-path: Re-compute those deviations which are not accurate enough and calculate the penetration depth according to Equation 3.11.

In Equation 3.10, there are $\Delta m - 1$ comparisons involved to find the minimum value. The only item of interest is the minimum value ${}_i\delta_{\Delta m}$, rather than the exact values of every $\delta_j(\boldsymbol{x}_i)$. Based on this insight, we define the comparison operation:

$$\begin{aligned} \delta^{min}_{1,\ldots,j} &= \min\left(\delta_1(\boldsymbol{x}_i), ..., \delta_j(\boldsymbol{x}_i)\right) \\ D &= \delta^{min}_{1,\ldots,j} - \delta_{j+1}(\boldsymbol{x}_i) \end{aligned} \tag{3.15}$$

The values of D when computed in reduced and high precision are denoted as D_{p_L} and D_{p_H}, respectively. D_{p_L} might have a flipped sign compared with D_{p_H}. We use the following three steps to make sure the results of Equation 3.10 is correct.

1. Evaluate Equation 3.15 using a reduced precision data format.
2. Estimate the maximum and minimum values of the value in high precision, i.e. $min(D_{p_H})$ and $max(D_{p_H})$, as shown in Equation 3.16.

   $E_{p_L}(\delta_{j+1}(\boldsymbol{x}_i))$ is the absolute error of $\delta_{j+1}(\boldsymbol{x}_i)$ in reduced precision p_L. It is computed at run-time and the details will be discussed later.

$$\begin{aligned} E_{p_L}(D_{p_L}) &= E_{p_L}(\delta^{min}_{1,\ldots,j}) + E_{p_L}(\delta_{j+1}(_i)) \\ \min\left(D_{p_H}\right) &= D_{p_L} - E_{p_L}(D_{p_L}) \\ \max\left(D_{p_H}\right) &= D_{p_L} + E_{p_L}(D_{p_L}) \end{aligned} \tag{3.16}$$

3. Determine whether the comparison result should be re-computed or dropped.

   Case A: $\min\left(D_{p_H}\right) > 0$, $\delta_{j+1}(\boldsymbol{x}_i)$ is smaller.

   Case B: $\max\left(D_{p_H}\right) < 0$, $\delta^{min}_{1,\ldots,j}$ is smaller.

   Case C: Cannot determine which value is smaller. Store both values for re-computation using high precision p_H.

In case A and B, the difference between the values is large enough to distinguish the sign of D_{p_H} even in the presence of errors introduced by reduced precision computations. In case C, the difference is small compared with the uncertainty introduced and therefore re-computation in high precision is necessary. The frequency of case C is lower than case A and B, therefore the performance gain from using reduced precision outweighs the re-computation overhead.

3.4.3 Dynamic Optimisation

We optimise the error bound based on feedback from run-time environment. Although the error bound $E_{p_L}(D_{p_L})$ can be derived statically [18], the estimated error bound grows pessimistically as it propagates along the data-path. Thus, we calculate the error bound using run-time data y and relative error RE_{p_L}. RE_{p_L} is profiled using a number of test vectors relative to a double precision data-path.

$$E_{p_L}(y) = y \cdot RE_{p_L} \tag{3.17}$$

On the other hand, we need to decide the precision used in the reduced precision data-paths. A lower precision increases the level of parallelism and hence increases the throughput of reduce precision data-path. However, it increases the ratio of re-computation and the total run-time. It is important to find an optimal for the best performance. The ratio of re-computation is data-dependent which changes over time and cannot be computed in advance.

We propose a method to search for the optimal precision at run-time. When a new data set is applied or the ratio of re-computation exceeds a threshold, Algorithm 1 is invoked on the CPU to reconfigure the FPGA with a higher precision. The computation overhead of the algorithm is negligible. $T(p_L)$ is the run-time measured computation time when using precision p_L as the reduced precision.

Algorithm 1 Run-time tuning of precision

```
1: Get the list of precisions $P$
2: $T(p_{test}) \leftarrow \infty$
3: repeat
4:     $T(p_L) \leftarrow T(p_{test})$
5:     $p_{test} \leftarrow \min(p \in P)$
6:     Remove $p_{test}$ from $P$
7:     Configure the FPGA with precision $p_{test}$
8:     Compute PQ and get $T(p_{test})$
9: until $T(p_{test}) > T(p_L)$
```

3.5 Reconfigurable System Design

In this section, we present our design which treats input points as a data stream that propagates through the customised system architecture. We also propose an analytical model for performance estimation.

3.5.1 Streaming Data Structure

In PQ, there are N_P points to represent a mesh. PQ computes the shortest distance from each point to the segment boundary defined by N_C contours. An intuitive implementation is to stream one point into the FPGA at the beginning, then the contours are streamed in the subsequent N_C iterations. In other words, Equation 3.3 to 3.10 are iterated for $N_C - 1$ times. However, since every comparison operation in Equation 3.10 takes $L_{Cmp} > 1$ clock cycles to compute, the next comparison can only start after the current one completes. This significantly reduces the FPGA's throughput for L_{Cmp} times because the pipeline is not fully-filled.

To tackle this problem, we propose a data structure for efficient streaming. As shown in Fig. 5.2(b), data are streamed in an order as indicated by the arrows. In each iteration of N_S cycles, $N_S > L_{Cmp}$ points are processed together as a group. A new contour value is streamed in at the beginning of each iteration. In this manner, N_S points are being processed together in the pipeline to retain one output per clock cycle.

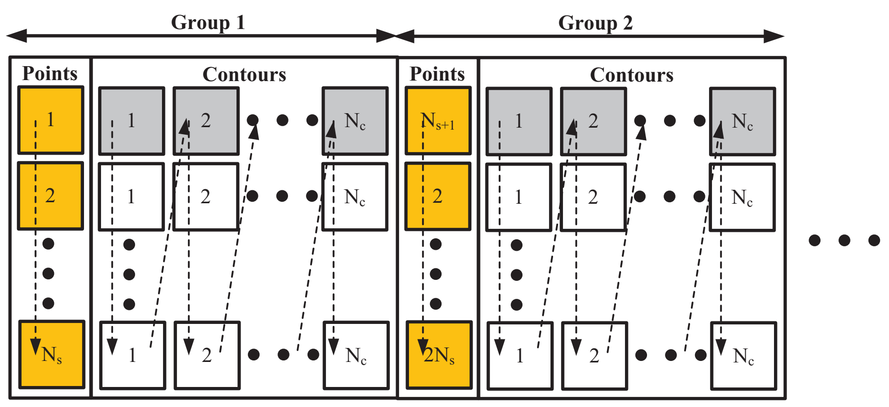


Figure 3.3: Data structure: N_S points are processed in a group. Each point of a group is iterated for N_C times. Data are streamed in an order as indicated by the arrows.

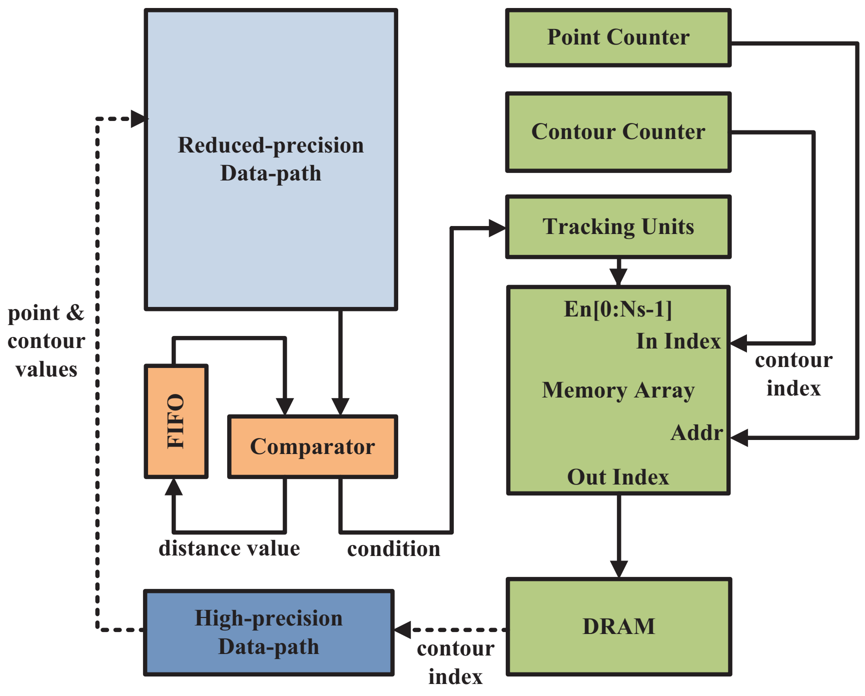


Figure 3.4: System architecture: Solid lines represent communication on the FPGA board while dotted lines represent the bus connecting the reduced precision data-path on FPGA to the high precision data-path on CPU.

3.5.2 System Architecture

Fig. 4.2 shows our proposed system architecture which consists of three major components.

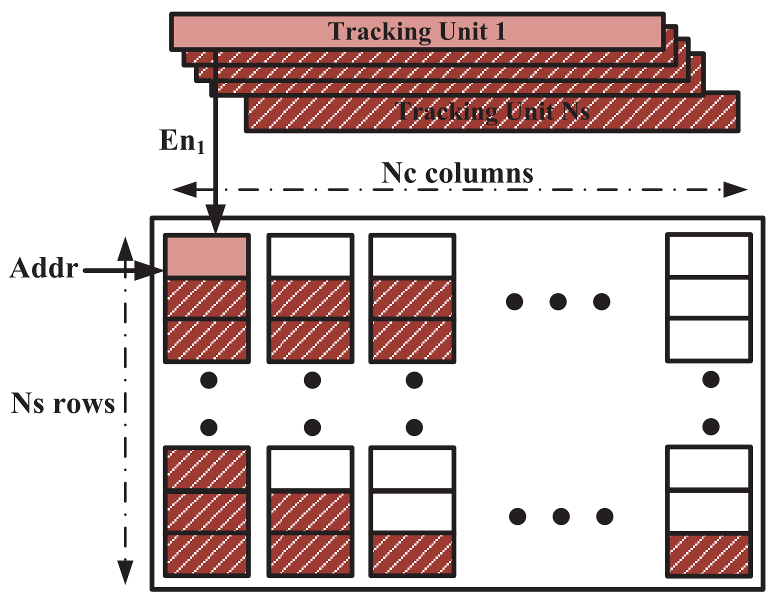
Data-paths: As mentioned in Section 5.4, we employ reduced precision on FPGA to compute the deviations. The high precision data-path on CPU re-computes the deviations which are not sufficiently accurate, and then it calculates the penetration depth based on the minimum deviation. The reduced precision and high precision data-paths are interfaced by a comparator and a memory architecture as described below.

Comparator: The comparator compares the values of two deviations and determines which one is smaller. The FIFO stores the latest minimum deviation which corresponds to a group of points. The FIFO has N_S slots because N_S points are processed together. Since the deviations are calculated in reduced precision, according to Section 3.4.2, either one of the three conditions happens: (A) The distance from the data-path is smaller; (B) The distance stored in the FIFO is smaller; (C) The difference between the two distances is too small, so re-computation in high precision is necessary.

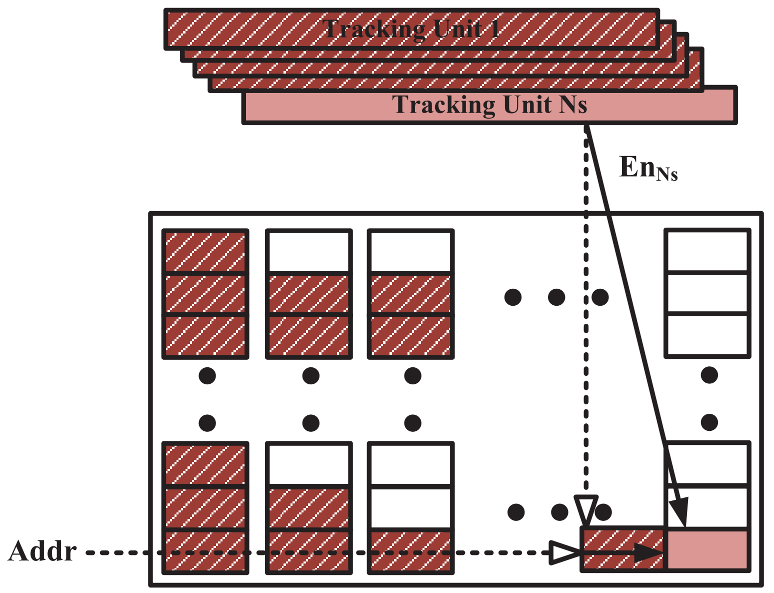
Memory Architecture: The purpose of the memory architecture is to store the contours that require re-computation. We design a memory array as shown in Fig. 3.5. There are N_S rows, each of which corresponds to the computation of one point which is addressed by a *point counter*. Each row consists of N_C elements and it serves as a buffer for contours that may need re-computation. Instead of storing the contours in three-dimensional coordinate, we store their indices to save memory space. The indices are counted by a *contour counter*. There are N_S *tracking units*, each for one row, to keep track of the latest elements where the indices should be written.

To understand the mechanism of memory architecture, consider the example in Fig. 3.5(a). First, the deviation in distance of *point 1* is being calculated. If the comparator indicates *condition A*, the value from the reduced precision data-path is the smallest, and all previous values stored in that row will be cleared. Second, the index corresponding to the new value is written to *element 1* of *row 1*. Third, *tracking unit 1* is updated to point to that element. If *condition B* is indicated, the minimum value is already stored in the memory and no update is required. Consider another example in Fig. 3.5(b) where the calculation of *point* N_S indicates *condition C*. Both the indices in the memory and from the data-path should be stored. Thus, a contour index is written to the next element and *tracking unit* N_S advances one element further.

After a group of points are processed, the contour indices stored in the memory array are transferred to the high precision data-path. To fully utilise the memory bandwidth, only non-



(a) Condition A: the value from the reduced precision data-path is the smallest, *tracking unit 1* points to the *element 1* of *row 1*. Previous vales stored in *row 1* are cleared.



(b) Condition C: both the value in the memory and the index from the data-path should be stored. A contour index is written to the next element and *tracking unit* N_S advances one element further.

Figure 3.5: Memory array stores contour indices for re-computation.

empty memory columns are transferred in burst to the DRAM on the FPGA board.

3.5.3 Performance Estimation

We derive a performance model to make the most effective use of the FPGA's resources. The results will be presented in Section 3.6.2 and 3.6.3. The total computation time T_{Comp} is affected by the time spent on three parts: (1) the reduced precision data-path on FPGA, (2) the high precision data-path on CPU, (3) the data transfer through the bus connecting the CPU to FPGA. Equation 3.18 shows the three parts respectively.

$$T_{Comp} = T_{p_L} + T_{p_H} + T_{Tran} \tag{3.18}$$

As shown in Equation 3.19, the computation time of FPGA depends on the number of points N_P and the number of contours N_C. L_{p_L} is the length of the data-path but this term is usually negligible when compared with the amount of data being processed. Each point needs L_{Output} cycles to output indices on the memory array to DRAM. L_{Output} is affected by the bit-width available between the FPGA and the DRAM and their relations are shown in Equation 3.20.

$$T_{p_L} = \frac{N_P \cdot (N_C + L_{Output})}{freq_{p_L} \cdot N_{p_L}} + L_{p_L} \tag{3.19}$$

$$L_{Output} = \frac{N_C}{N_{Output}} \; , \quad N_{Output} = \frac{W_{DRAM}}{W_{Idx} \cdot N_{p_L}} \tag{3.20}$$

The computation time of CPU is related to the amount of data and the ratio of re-computation.

$$T_{p_H} = \alpha \cdot R \cdot N_P \cdot N_C \tag{3.21}$$

The data transfer time from the DRAM to CPU is judged by the amount of data, the ratio of

Table 3.1: Parameters of the performance model

N_P	Num. of points
N_C	Num. of contours
N_{p_L}	Num. of reduced precision data-path
N_{p_H}	Num. of high precision data-path
L_{p_L}	Length of the data-path
N_{Output}	Num. of outputs per data-path per cycle
L_{Output}	Num. of output cycles
R	Ratio of re-computation
W_{DRAM}	Bit-width of FPGA-DRAM connection
W_{Idx}	Bit-width of one contour index
$freq_{p_L}$	Clk. freq. of reduced precision data-path
α	Empirical constant of CPU speed
BW_{bus}	Bandwidth of the bus connecting the CPU to FPGA

re-computation and the bandwidth of the bus connecting the CPU to FPGA.

$$T_{Tran} = \frac{R \cdot N_P \cdot N_C \cdot W_{Idx}}{BW_{bus}} \tag{3.22}$$

3.6 Experimental Evaluation

3.6.1 General Settings

We use the MPC-C500 reconfigurable system from Maxeler Technologies for our evaluation. The system has four MAX3 cards, each of which has a Virtex-6 XC6VSX475T FPGA with 476,100 logic cells and 2,016 DSPs. The cards are connected to two Intel Xeon X5650 CPUs and each card communicates with the CPUs via a PCI Express gen2 x8 link. The CPUs have 12 physical cores and are clocked at 2.66 GHz. We develop the FPGA kernels using MaxCompiler which adopts a streaming programming model and it supports customisable floating-point data formats.

We also build a CPU-based system by implementing the PQ formulation on a platform with two Intel Xeon X5650 CPUs running at 2.66 GHz. The code is written in C++ and compiled by Intel C compiler with the highest optimisation. OpenMP library is used to parallelise the program for multiple cores. IEEE double precision floating point numbers are used.

For the GPU-based system, we use an NVIDIA Tesla C2070 GPU which has 448 cores running at 1.15 GHz.

Our PQ implementation supports 100 contours and we set an update rate of 1 kHz as the real-time requirement.

3.6.2 Parallelism versus Precision

Fig. 3.6 shows the overall computation time (T_{Comp}) and the degree of parallelism of PQ versus different number of mantissa bits. Please note that all different configurations of mantissa bits have the same output accuracy. The data set includes 73k points and 100 contours. The computation times are obtained using our analytical model in Section 5.3.3 and they are verified experimentally using the implementation. The degree of parallelism is obtained by filling the FPGA with data-paths until the logic cell utilisation exceeds 80% after the placement and routing process. The degree of parallelism is the highest when we start with four mantissa bits. Using more mantissa bits decreases the parallelism as well as the ratio of re-computation, therefore T_{p_L} increases but T_{p_H} decreases. As shown by the dotted line in the figure, a minimum computation time is achieved when 10 mantissa bits are used. Note that when the number of mantissa bits is more than 36, only one data-path can be mapped onto the FPGA. In such cases, we can implement the data-path in double precision directly which does not require any re-computation on CPU. This is indicated by the last data points of both curves.

3.6.3 Ratio of Re-computation versus Precision

The dotted line in Fig. 3.7 shows the ratio of re-computation versus the number of mantissa bits. The results are obtained from a software version of PQ implementation with precisions adjusted using MPFR library [24]. For each point, 100 computations of deviation in distance are required. The ratio of re-computation drops exponentially as the number of mantissa bits

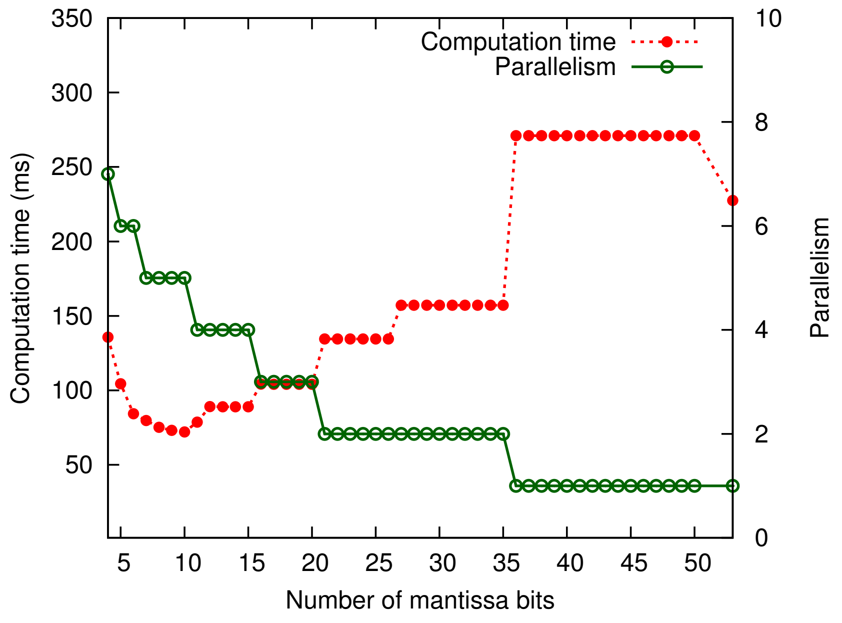


Figure 3.6: Computation time [dotted line] and the level of parallelism [solid line] vs. different number of mantissa bits.

increases. From the performance perspective, to the left the ratio of re-computation is too high, to the right the decrease of re-computation cannot offset the impact brought by the decrease in parallelism. When the number of mantissa bits is four, in average 2.66 out of 100 computations need to be re-computed using high precision, i.e. the ratio of re-computation is 2.66%. When the number of mantissa bits is greater then 15, the ratio of re-computation drops to 1% which is the minimum value as only one out of 100 values is re-computed. The last data points of both curves indicate the situation when double precision is used on the FPGA and no re-computation is necessary.

The solid line in Fig. 3.7 shows the number of point processed in 1 ms versus the number of mantissa bits. The application has a real-time update requirement of 1 kHz so the results are updated every 1 ms. The number of required vertex points is based on the user specification of the model resolution in three-dimensional space. When the number of mantissa bits is 10, the maximum number of points can be processed. It is because the throughput is the highest by balancing the ratio of re-computation and the degree of parallelism. Since more points can be processed in real-time, we can handle a more complex robot model with a finer resolution.

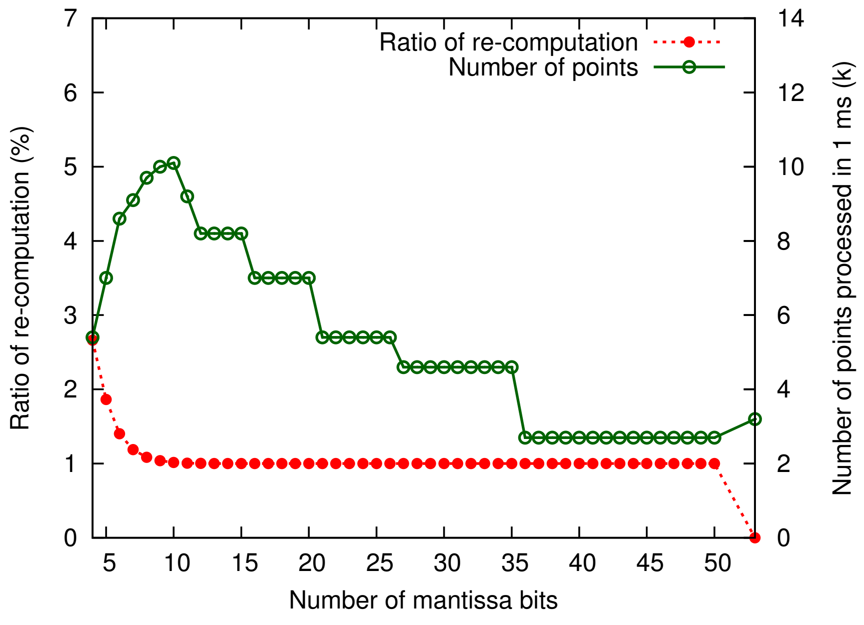


Figure 3.7: Ratio of re-computation [dotted line] and the number of points processed in 1 ms [solid line] vs. different number of mantissa bits.

3.6.4 Comparison: CPU, GPU and FPGA

Table 3.2 compares the performance of PQ running on CPU, GPU and FPGA in double precision arithmetic, and our proposed reconfigurable system with CPUs and FPGAs.

In 1 ms, our proposed system is able to process 58 times more points than a 12-core CPU system, and 9 times more points than a GPU system. Without any optimisation, we can only implement one double precision data-path on an FPGA. Our proposed approach can support five reduced precision data-paths to be implemented in parallel on one chip, i.e. 20 data-paths in total on the 4-FPGA system. The clock frequency is also higher because reduced precision simplifies routing of signals. The performance gain over a double precision FPGA implementation is over 3 times.

Fig. 3.8 shows the computation time for a PQ update against the number of vertex points. The black solid line indicates the real-time bound of 1 ms. In the CPU-based system, even with the fastest configuration (12 cores), only 173 points can be processed in real-time. Meanwhile, the performance of our proposed 1-FPGA system is on-par with a 4-FPGAs system in double precision. Our proposed 4-FPGAs system can process 10,094 points within the 1 ms interval.

Table 3.2: Comparison of PQ computation in 1 ms using CPU-based system (CPU), GPU-based system (GPU), double precision FPGA-based system (FPGA DP) and FPGA+CPU system with reduced precision (FPGA RP)

	CPU	GPU	FPGA DP	FPGA RP
Clock freq. (MHz)	2,660	1,150	80	130 & 2,660 [a]
Num. of cores	12	448	4	20
Num. of mantissa bits	53	53	53	10 & 53 [b]
Num. of p_L eval. (k)	0	0	0	1009.4
Num. of p_H eval. (k)	173	106	320	10.1
Num. of total eval. (k)	173	106	320	1019.5
Eval. in p_H (%)	100	100	100	1
Num. of points in 1 ms	173	1,064	3,200	10,094
Normalised speedup	1x	6.15x	18.5x	58.35x
Reduced precision gain	-	-	1x	3.15x

[a] FPGA and CPU clock frequencies.
[b] Reduced precision and high precision.

3.7 Summary

This paper presents a reconfigurable computing solution to proximity query computation. To the best of our knowledge, our approach is the first to apply FPGAs to this problem.

We transform the algorithm to enable pipelining and apply reduced precision methodology to maximise parallelism. Run-time reconfiguration is employed to optimise precision automatically. We then map the optimised algorithm to a reconfigurable system with four Virtex-6 FPGAs and 12 CPU cores. Our proposed system achieves 478 times speedup over a single-core CPU, 58 times speedup over a 12-core CPU system, 9 times speedup over a GPU, and 3 times speedup over an FPGA implementation in double precision. Since more points can be processed in real-time, we can handle a more complex robot model with a finer resolution.

The work shows the potential of reconfigurable computing for PQ. Real-time performance is the pre-requisite to enable Dynamic Active Constraints, which has drawn increasing attention for effective human-robot collaborative control. Future work includes extending the current run-time reconfigurable architecture to cover other real-time applications that can benefit from the reduced precision approach. These applications, such as the real-time PQ between a robotic device and a rapidly deforming anatomy, will help us to evaluate the impact of run-time reconfiguration on various data sets. We are currently extending this work to cover imaged-guided

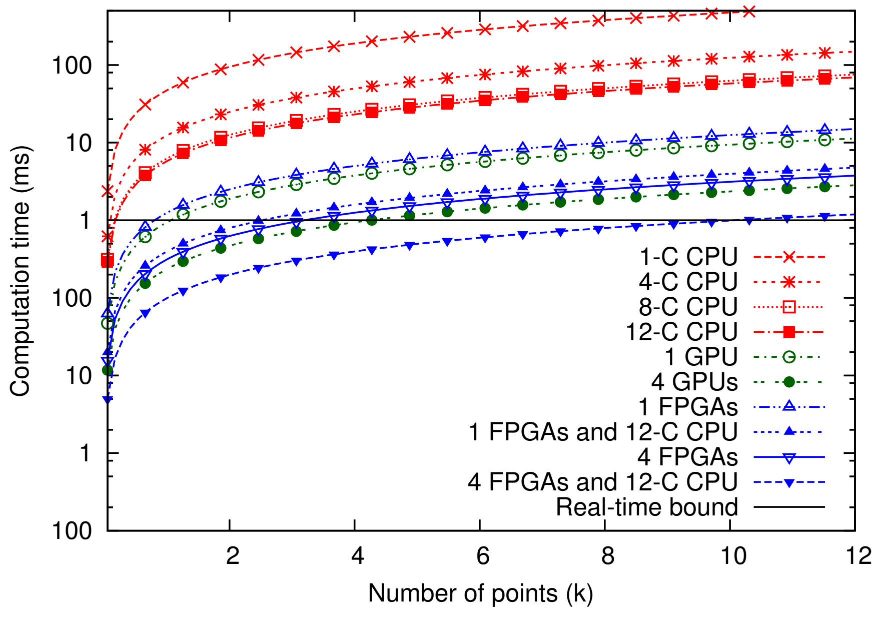


Figure 3.8: Computation time for a PQ update with 100 contours vs. the number of points.

catheterisation, particularly for cardiac electrophysiology intervention. To deal with the rapid deformation of the heart and the associated vessels, it is vitally important to provide the operator of a surgical robot online intra-operative guidance in real time, for which fast and efficient PQ computation is essential.

Chapter 4

Self-adaption and Run-time Reconfiguration for Sequential Monte Carlo Applications

4.1 Introduction

Particle filter (PF), also known as sequential Monte Carlo (SMC) method, is a statistical method for dynamic systems involving non-linear and non-Gaussian properties. PF has been studied in various application areas including object tracking [25], robot localisation [26], speech recognition [27] and air traffic management [28].

PF keeps track of a large number of particles where each contains information about how a system would evolve. The underlying concept is to approximate a sequence of states as a collection of particles. Each particle is weighted to reflect the quality of an approximation. The more complex the problem, the larger the number of particles that are needed. One drawback of PF is its long execution times that limit its practical use.

This paper presents an efficient solution to PF. We derive an adaptive algorithm that adjusts its computation complexity at run time based on the quality of results. To map our algorithm to a heterogeneous reconfigurable system (HRS) consisting of multiple FPGAs and CPUs, we design a pipeline-friendly data structure to make effective use of the stream computing model. Moreover, we effectively accelerate the algorithm with a data compression scheme and data control separation.

The key contributions of this paper include:

1. An adaptive PF algorithm which adapts the size of particle set at run-time. The algorithm is able to reduce computation workload while maintaining quality of results.
2. Mapping the proposed algorithm to a scalable and reconfigurable system by following the stream computing model. A novel data structure is designed to take advantage of the architecture and to alleviate the data transfer bottleneck. The system uses the run-time reconfigurability of FPGA to switch between computation mode and low-power mode.
3. An implementation of a robot localisation application targeting the proposed system. Compared to a non-adaptive and non-reconfigurable implementation, the idle power of our proposed system is reduced by 25-34% and the overall energy consumption decreases by 17-33%. Our system with four FPGAs is up to 169 times faster than a single core CPU, 41 times faster than a 1U CPU server with 12 cores, and 3 times faster than a modelled four-GPU system.

4.2 Overview

This section briefly outlines the PF algorithm. A more detailed description can be found in [29]. PF estimates the state of a system by a sampling-based approximation of the state probability density function. The state of a system in time-step t is denoted by X_t. The control and observation are denoted by U_t and Y_t respectively. Three pieces of information about the system are known a-priori:

- $p(X_0)$ is the probability of the initial state of the system,

- $p(X_t|X_{t-1}, U_{t-1})$ is the state transition probability of the system's current state given a previous state and control information,
- $p(Y_t|X_t)$ is the observation model describing the likelihood of observing the measurement at the current state.

PF approximates the desired posterior probability $p(X_t|Y_{1:t})$ using a set of P particles $\{\chi_t^{(i)}\}_{i=1}^P$ with their associated weights $\{w^{(i)}\}_{i=1}^P$. X_0 and U_0 are initialised. This computation consists of three iterative steps.

1. **Sampling**: A new particle set $\{\widetilde{\chi}_t^{(i)}\}_{i=1}^P$ is drawn from the distribution $p(X_t|X_{t-1}, U_{t-1})$, forming a prediction of the distribution of X_t.
2. **Importance weighting**: The likelihood $p(Y_t|\widetilde{\chi}_t^{(i)})$ of each particle is calculated. The likelihood indicates whether the current measurement Y_t matches the predicted state $\{\widetilde{\chi}_t^{(i)}\}_{i=1}^P$. Then each particle is assigned a weight $w^{(i)}$ with respect to the likelihood.
3. **Resampling**: Particles with higher weights are replicated and the number of particles with lower weights is reduced. With resampling, the particle set has a smaller variance. The particle set is used in the next time-step to predict the posterior probability subsequently. The distribution of the resulting particles $\{\chi_t^{(i)}\}_{i=1}^P$ approximates $p(X_t|Y_{1:t})$.

The particles in PF are independent of each other. It means the algorithm can be accelerated using specialised hardware with massive parallelism and pipelining. In [25], an approach for PF on a hybrid CPU/FPGA platform is developed. Using a multi-threaded programming model, computation is switched between hardware and software during run-time to react to performance requirements. Resampling algorithms and architectures for distributed PFs are proposed in [30].

Adaptive PFs have been proposed to improve performance or quality of state estimation by controlling the number of particles dynamically. Likelihood-based adaptation controls the number of particles such that the sum of weights exceeds a pre-specified threshold [31]. Kullback

Leibler distance (KLD) sampling is proposed in [32], which offers better quality results than likelihood-based approach. KLD sampling is improved in [33] by adjusting the variance and gradient of data to generate particles near high likelihood regions. The above methods introduce data dependencies in the sampling and importance weighting steps, so they are difficult to be parallelised. An adaptive PF is proposed in [34] that changes the number of particles dynamically based on estimation quality. In [1], adaptive PF is extended to a multi-processor system on FPGA. The number of particles and active processors change dynamically but the performance is limited by soft-core processors. In [35], a mechanism and theoretical lower bound for adapting the sample size of particles is presented. Our previous work [2] presents a hardware-friendly adaptive PF. The algorithm is mapped to an accelerator system which consists of an FPGA and a CPU. However, the system suffers from a large communication overhead when the particles are transferred between the FPGA and CPU. Moreover, the scalability of the adaptive PF algorithm to multi-FPGA is not well-studied. In this paper, we extend our previous work to address the problems mentioned above.

4.3 Adaptive Particle Filter

This section introduces an adaptive PF algorithm which changes the number of particles at each time-step. The algorithm is inspired by [35] and we transform it to a pipeline-friendly version for mapping to the stream computing architecture. This algorithm is shown in Algorithm 2 which consists of four stages.

4.3.1 Stage 1: Sampling and Importance Weighting (line 8 to 9)

At the initial time-step ($t = 0$), the maximum number of particles are used, i.e. $P_0 = P_{max}$. At the subsequent time-steps, the number of particles is denoted as P_t. Initially, the particle set $\{\chi_t^{(i)}\}_{i=1}^{P_t}$ is sampled to $\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}$. Then a weight from $\{w^i\}_{i=1}^{P_t}$ is assigned to each particle. As a result, $\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}$ and $\{w^{(i)}\}_{i=1}^{P_t}$ give an estimation of the next state.

Algorithm 2 Adaptive PF algorithm

```
$P_0 \leftarrow P_{max}$
$\{X_0^{(i)}\}_{i=1}^{P_0}$ $\leftarrow$random set of particles
$t = 1$
for each step $t$ do
    $r = 0$
    while $r \leq itl\_repeat$ do
        —On FPGAs—
        Sample a new state $\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}$ from $\{\chi_t^{(i)}\}_{i=1}^{P_t}$
        Calculate unnormalised importance weights $\{\widetilde{w}^{(i)}\}_{i=1}^{P_t}$ and accumulate the weights as $w_{sum}$
        Calculate the lower bound of sample size $\widetilde{P}_{t+1}$ by Equation 4.1
        —On CPUs—
        Sort $\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}$ in descending $\{\widetilde{w}^{(i)}\}_{i=1}^{P_t}$
        if $\widetilde{P}_{t+1} < P_t$ then
            $P_{t+1} = max\left(\lceil \widetilde{P}_{t+1} \rceil, P_t/2\right)$
            Set $a = 2P_{t+1} - P_t$ and $b = P_{t+1}$
            –Do the following loop in parallel–
            for $i$ in $P_t - P_{t+1}$ do
                $\widetilde{\chi}_{t+1}^{(i)} = \frac{\chi_{t+1}^{(a)}\widetilde{w}^{(a)} + \chi_{t+1}^{(b)}\widetilde{w}^{(b)}}{\widetilde{w}^{(a)} + \widetilde{w}^{(b)}}$
                $\widetilde{w}^{(i)} = \widetilde{w}^{(a)} + \widetilde{w}^{(b)}$
                $a = a + 1$ and $b = b - 1$
            end for
        else if $\widetilde{P}_{t+1} \geq P_t$ then
            $a = 0$ and $b = 0$
            for $i$ in $P_{t+1} - P_t$ do
                if $\widetilde{w}^{(a)} < \widetilde{w}^{(a+1)}$ and $a < P_{t+1}$ then
                    $a = a + 1$
                end if
                $\widetilde{\chi}_{t+1}^{(P_t+b)} = \widetilde{\chi}_{t+1}^{(a)}/2$
                $\widetilde{\chi}_{t+1}^{(a)} = \widetilde{\chi}_{t+1}^{(a)}/2$
                $\widetilde{w}^{(P_t+b)} = \widetilde{w}^{(a)}/2$
                $\widetilde{w}^{(a)} = \widetilde{w}^{(a)}/2$
                $b = b + 1$
            end for
        end if
        Resample $\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}$ to $\{\chi_{t+1}^{(i)}\}_{i=1}^{P_{t+1}}$
        $r = r + 1$
    end while
end for
```

During sampling and importance weighting, the computation of every particle is independent of each other. The mapping of computation to FPGAs will be described in Section 4.4.

4.3.2 Stage 2: Lower Bound Calculation (line 10)

This stage derives the smallest number of particles that are needed in the next time-step in order to bound the approximation error. The adaptive algorithm seeks a value which is less than or equal to P_{max}. This number, denoted as $\widetilde{P}_{t+1}$, is referred to as the lower bound of sampling size. It is calculated by Equation 4.1 to 4.4.

$$\widetilde{P}_{t+1} = \sigma^2 \cdot \frac{P_{max}}{Var(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t})} \tag{4.1}$$

$$\begin{aligned} \sigma^2 =& \sum_{i=1}^{P_t} \left(w^{(i)} \cdot \widetilde{\chi}_{t+1}^{(i)} \right)^2 - 2 \cdot E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}) \cdot \sum_{i=1}^{P_t} \left((w^{(i)})^2 \cdot \widetilde{\chi}_{t+1}^{(i)} \right) \\ &+ \left(E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}) \right)^2 \cdot \sum_{i=1}^{P_t} (w^{(i)})^2 \end{aligned} \tag{4.2}$$

$$Var(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}) = \sum_{i=1}^{P_t} \left(w^{(i)} \cdot (\widetilde{\chi}_{t+1}^{(i)})^2 \right) - \left(E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}) \right)^2 \tag{4.3}$$

$$E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}) = \sum_{i=1}^{P_t} w^{(i)} \cdot \widetilde{\chi}_{t+1}^{(i)} \tag{4.4}$$

As shown in Equation 4.2 to 4.4, $w^{(i)}$ is a normalised term. To calculate $w^{(i)}$, a traditional software-based approach is to iterate through the set of particles twice. The sum of weights w_{sum} and unnormalised weight $\widetilde{w}^{(i)}$ are calculated in the first iteration. Then $w^{(i)}$ is obtained by dividing $\widetilde{w}^{(i)}$ by w_{sum} in the second iteration. However, this method is inefficient for FPGA implementation. Since $2P_t$ cycles are needed to process P_t pieces of data, the throughput is reduced to 50%.

To fully utilise the deep pipeline on FPGA, we perform function transformation. Given $w^{(i)} = \frac{\widetilde{w}^{(i)}}{w_{sum}}$, we extract w_{sum} out of Equation 4.2 to 4.4. By doing so, we obtain a transformed form as shown in Equations 4.5 to 4.7. w_{sum} and $\widetilde{w}^{(i)}$ are computed simultaneously in two separate data paths. At the last clock cycle of the particle stream, σ^2, $Var(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t})$ and $E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t})$ are obtained. The details of the FPGA kernel design will be explained in Section 4.4.

$$\begin{aligned}\sigma^2 =& \frac{1}{(w_{sum})^2} \cdot (\sum_{i=1}^{P_t} \left(\widetilde{w}^{(i)} \cdot \widetilde{\chi}_{t+1}^{(i)}\right)^2 - 2 \cdot E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}) \cdot \sum_{i=1}^{P_t} \left((\widetilde{w}^{(i)})^2 \cdot \widetilde{\chi}_{t+1}^{(i)}\right) \\ &+ \left(E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t})\right)^2 \cdot \sum_{i=1}^{P_t} (\widetilde{w}^{(i)})^2)\end{aligned} \tag{4.5}$$

$$Var(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}) = \frac{1}{w_{sum}} \cdot \sum_{i=1}^{P_t} \left(\widetilde{w}^{(i)} \cdot (\widetilde{\chi}_{t+1}^{(i)})^2\right) - \left(E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t})\right)^2 \tag{4.6}$$

$$E(\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}) = \frac{1}{w_{sum}} \cdot \sum_{i=1}^{P_t} \widetilde{w}^{(i)} \cdot \widetilde{\chi}_{t+1}^{(i)} \tag{4.7}$$

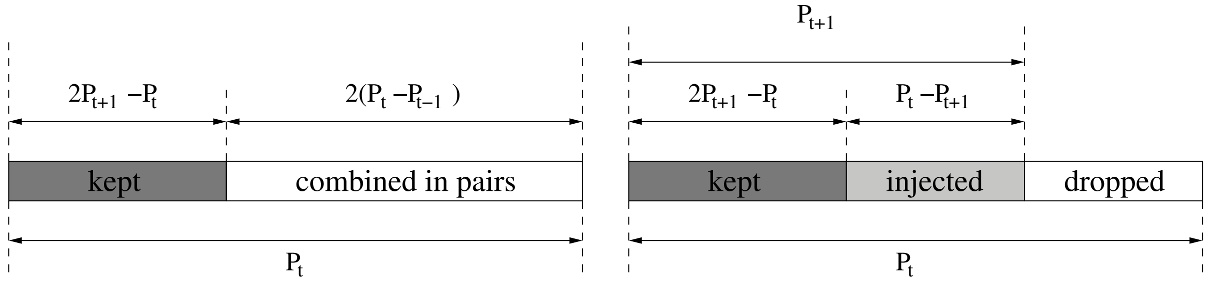
4.3.3 Stage 3: Particle set size tuning (line 12 to 34)

The adaptive approach tunes the particle set size to fit the lower bound P_{t+1}. This stage is done on the CPUs because the operations involve non-sequential data access that cannot be mapped efficiently to FPGAs.

The particles are sorted in descending order according to their weights. As the new sample size can increase or decrease, there are two cases:

- **Case I: Particle set reduction when $\widetilde{P}_{t+1} < P_t$**

  The lower bound P_{t+1} is set to $max\left(\lceil \widetilde{P}_{t+1} \rceil, P_t/2\right)$. Since the new size is smaller than the old one, some particles are combined to form a smaller particle set. Figure 4.1 illustrates the idea of particle reduction. The first $2P_{t+1} - P_t$ particles with higher weights are kept and the remaining $2(P_t - P_{t+1})$ particles are combined in pairs. As a result, there are



(a) Combining the last $2(P_t - P_{t+1})$ particles with lower weights

(b) P_{t+1} new particles are formed

Figure 4.1: Particle set reduction

$P_t - P_{t+1}$ new particles injected to form the target particle set with P_{t+1} particles. We combine the particles deterministically to keep the statements in the loop independent of each other. As a result, loop unrolling is undertaken to execute the statements in parallel. The complexity of the loop is in $\mathcal{O}\left(\frac{P_t - P_{t+1}}{N_{parallel}}\right)$, where $N_{parallel}$ indicates the level of parallelism.

- **Case II: Particle set expansion when $\widetilde{P}_{t+1} \geq P_t$**

  The lower bound P_{t+1} is set to $\widetilde{P}_{t+1}$. Some particles are taken from the original set and are inserted to form a larger set. The particles with larger weight would have more descendants. As shown in line 22 to 34, the process requires picking the particle with the largest weight at each iteration of particle incision. Since the particle set is pre-sorted, the complexity of particle set expansion is $\mathcal{O}(P_{t+1} - P_t)$.

4.3.4 Stage 4: Resampling (line 35)

Resampling is performed to pick P_{t+1} particles from $\{\widetilde{\chi}_{t+1}^{(i)}\}_{i=1}^{P_t}$ to form $\{\chi_{t+1}^{(i)}\}_{i=1}^{P_{t+1}}$. The process has a complexity of $\mathcal{O}(P_{t+1})$.

4.4 Heterogeneous Reconfigurable System

This section describes the proposed heterogeneous reconfigurable system (HRS). It is scalable to cope with different FPGA devices and applications. HRS also takes advantage of the run-time

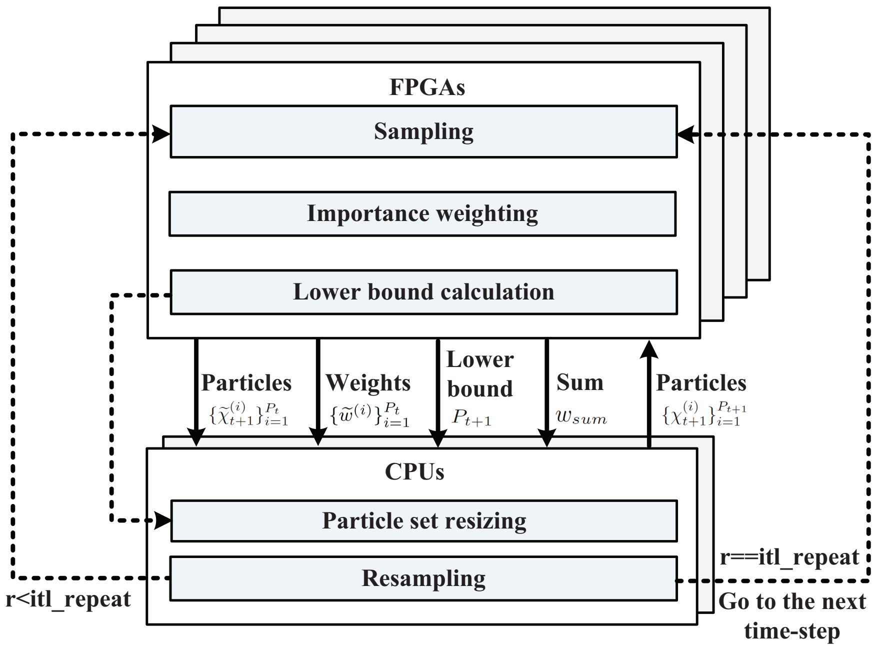


Figure 4.2: Heterogeneous reconfigurable system (Solid lines: data paths; Dotted lines: control paths)

reconfiguration feature for power and energy reduction.

4.4.1 Mapping adaptive PF to HRS

The system design of HRS is shown in Figure 4.2. A heterogeneous structure is employed to make use of multiple FPGAs and CPUs. FPGAs and CPUs communicate through high bandwidth buses. FPGAs are responsible for (1) sampling, (2) importance weighting, and (3) lower bound calculation. The data paths on the FPGAs are fully-pipelined. Each FPGA has its own on-board dynamic random-access memory (DRAM) to store the large amount of particle data. On the other hand, the CPUs gather all the particles from FPGAs to perform particle set size tuning and resampling.

4.4.2 FPGA Kernel Design

Sampling, importance weighting and lower bound calculation are the most computation intensive stages. In each time-step, these three stages are iterated for *itl_repeat* times. An FPGA

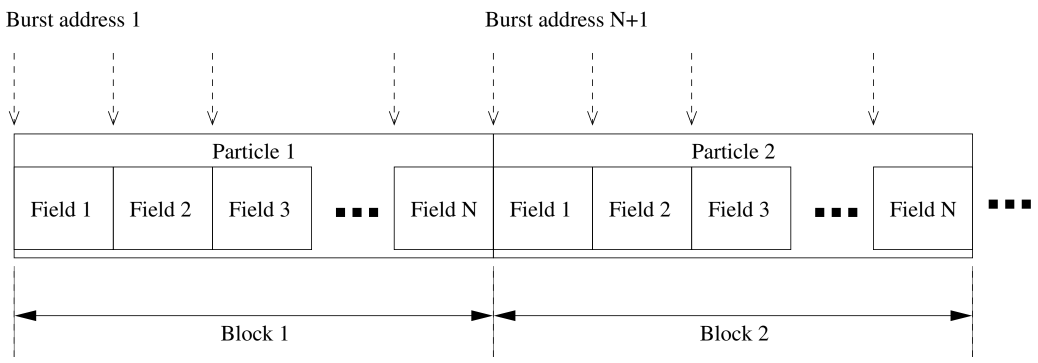


Figure 4.3: A particle stream

kernel is designed to enable acceleration of them.

Figure 5.3 shows the components of the FPGA kernel. The kernel is fully pipelined to achieve one output per clock cycle. It can also be replicated as many times as FPGA resource allow and the replications can be split across multiple FPGA boards. The kernel takes three inputs from the CPUs or on-board DRAM: (1) states, (2) controls, and (3) seeds. Application specific parameters are stored in ROMs. Three building blocks correspond to the sampling, importance weighting and lower bound calculation stages as described in Section 4.3.

For sampling and importance weighting, the computation of each particle is independent of each other. Particles are fed to the FPGAs as a stream shown in Figure 4.3. Each block of the particle stream consists of a number of data fields which store information of a particle. The number of data fields is application dependent. In every clock cycle, one piece of data is transferred from the onboard memory to an FPGA data path. Each FPGA data path has a long pipeline where each stage is filled with a piece of data, and therefore many particles are processed simultaneously. Fixed-point data representation is customised at each pipeline stage to reduce the resource usage.

Meanwhile, the accumulation of w_{sum} introduces a feedback loop. A new weight comes along every cycle which is more quickly than the floating-point unit to perform addition of the previous weight. In order to achieve one result per clock cycle, fixed-point data-path is implemented while ensuring no overflow or underflow occurs.

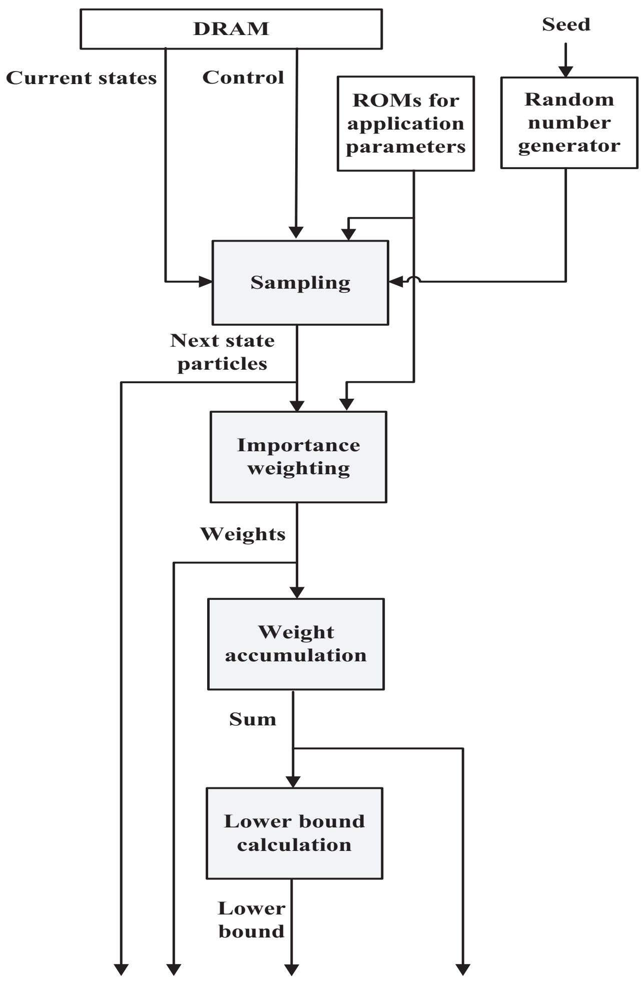


Figure 4.4: FPGA kernel design

4.4.3 Timing model for run time reconfiguration

We derive a model to analyse the computation time of HRS. The model helps us to design a configuration schedule that satisfies the real-time requirement and, if necessary, amend the application's specification. The model will be validated by experiments in Section 4.6.

The computation time (T_{comp}) of HRS consists of three components: (1) Data path time $T_{datapath}$, (2) CPU time T_{CPU}, and (3) Data transfer time T_{tran}. The sampling, importance weighting and resampling processes are repeated for *itl_repeat* times in every time-step.

$$T_{comp} = itl\_repeat \cdot (T_{datapath} + T_{CPU} + T_{tran}) \tag{4.8}$$

Data path time, $T_{datapath}$, denotes the time spent on the FPGAs. P_t denotes the number of particles at the current time-step and f_{FPGA} denotes the clock frequency of the FPGAs. L is the length of the pipeline. $N_{datapath}$ denotes the number of data paths on one FPGA board. N_{FPGA} is the number of FPGA boards in the system.

$$T_{datapath} = \left(\frac{P_t}{f_{FPGA} \cdot N_{datapath}} + L - 1 \right) \frac{1}{N_{FPGA}} \tag{4.9}$$

CPU time, T_{CPU}, denotes the time spent on the CPUs. The clock frequency and number of threads of the CPUs are represented by f_{CPU} and N_{thread} respectively. par is an application-specific parameter in the range of $[0, 1]$ which represents the ratio of CPU instructions that are parallelisable, and α is a scaling constant derived empirically.

$$T_{CPU} = \alpha \cdot \frac{P_t}{f_{CPU}} \cdot \left(1 - par + \frac{par}{N_{thread}} \right) \tag{4.10}$$

Data transfer time, T_{tran}, denotes the time of moving a particle stream between the FPGAs and the CPUs. df is the number of data fields of a particle. For example, if a particle contains the information of coordinates (x, y) and heading h, $df = 3$. Given that the constant 1 represents the weight and the constant 2 accounts for the movement of data in and out of the FPGAs, and bw_{data} is the bit-width of one data field, the expression $(2 \cdot df + 1) \cdot bw_{data}$ is regarded as the size of a particle.

f_{bus} is the clock frequency of the bus connecting the CPUs to FPGAs and $lane$ is the number of bus lanes connected to one FPGA. Since many buses, such as the PCI Express Bus, encode data during transfer, the effective data are denoted by eff (in PCI Express Gen2 the value is 8/10). In our previous work [2], the data transfer time has a significant performance impact on HRS. To reduced the data transfer overhead, we introduce a data compression technique that will be described in Section 4.5.

$$T_{tran} = \frac{(2 \cdot df + 1) \cdot bw_{data} \cdot P_t}{f_{bus} \cdot lane \cdot eff \cdot N_{FPGA}} \tag{4.11}$$

In real-time applications, each time-step is fixed and is known as the real-time bound T_{rt}. The derived model helps system designers to ensure that the computation time T_{comp} is shorter than T_{rt}. An idle time T_{idle} is introduced to represent the time gap between the computation time and real-time bound.

$$T_{idle} = T_{rt} - T_{comp} \tag{4.12}$$

Figure 4.5(a) illustrates the power consumption of HRS without run-time reconfiguration. It shows that the FPGAs are still drawing power after the computation finishes. By exploiting run-time reconfiguration as shown in Figure 4.5(b), the FPGAs are loaded with a low-power configuration during the idle period. Such configuration minimises the amount of active resources and clock frequency. Equation 4.13 describes the sleep time when the FPGAs are idle and being loaded with the low-power configuration. If the sleep time is positive, reconfiguration would be helpful in these situations.

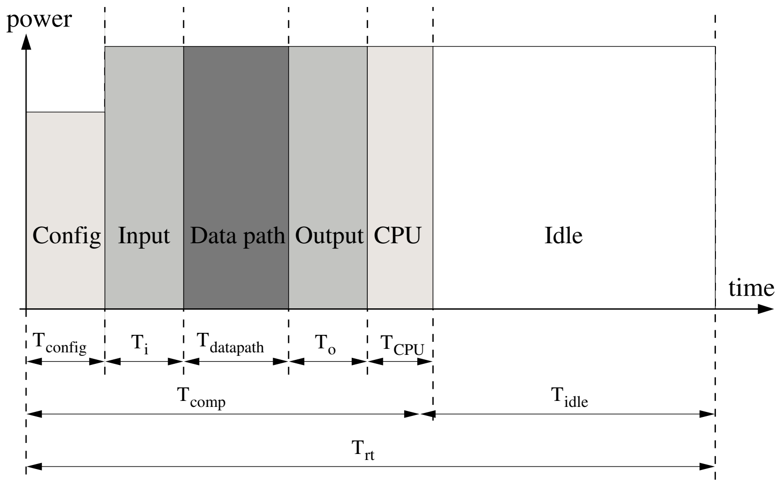
$$T_{sleep} = T_{idle} - T_{config} \tag{4.13}$$

Configuration time, T_{config}, denotes the time needed to download a configuration bit-stream to the FPGAs. $size_{bs}$ represents the size of bitstream in bits. f_{config} is the configuration clock frequency in Hz and bw_{config} is the width of the configuration port.

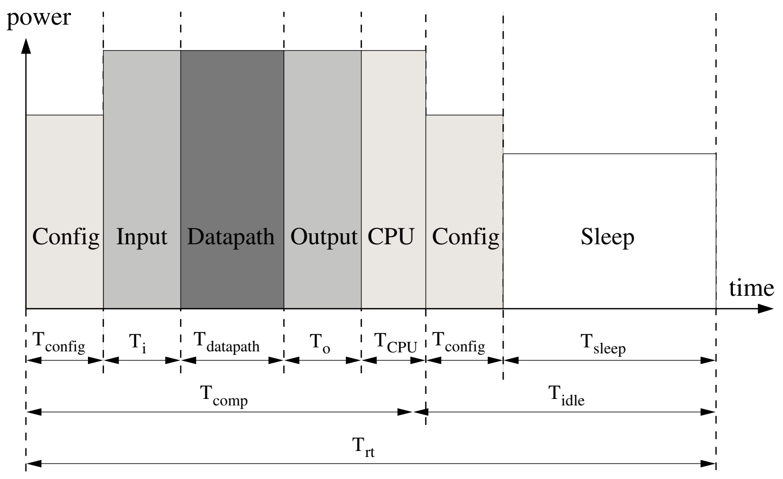
$$T_{config} = \frac{size_{bs}}{f_{config} \cdot bw_{config}} \tag{4.14}$$

4.5 Optimising Transfer of Particle Stream

In Section 4.4, the data transfer time depends on the number of particles and the bus bandwidth between the CPUs and FPGAs. It can be a major performance bottleneck as depicted in [2]. Refer to Figure 4.6(a), each block stores the data of a particle. When the CPUs finish processing, all data are transferred from the CPUs to the FPGAs. The data transfer time



(a) Without reconfiguration



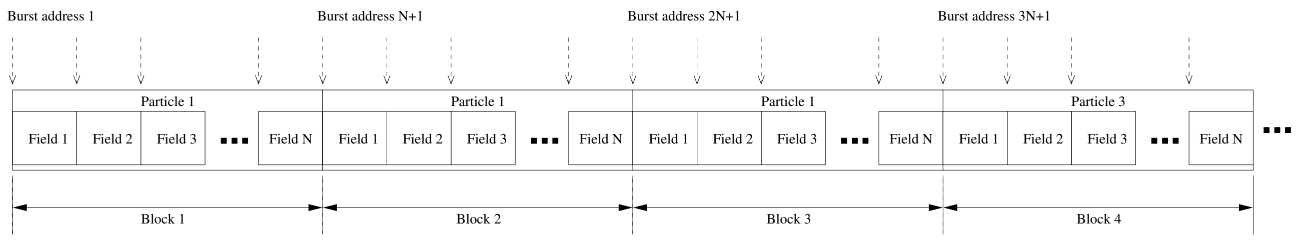
(b) With reconfiguration to low-power mode during idle

Figure 4.5: Power consumption of the HRS over time

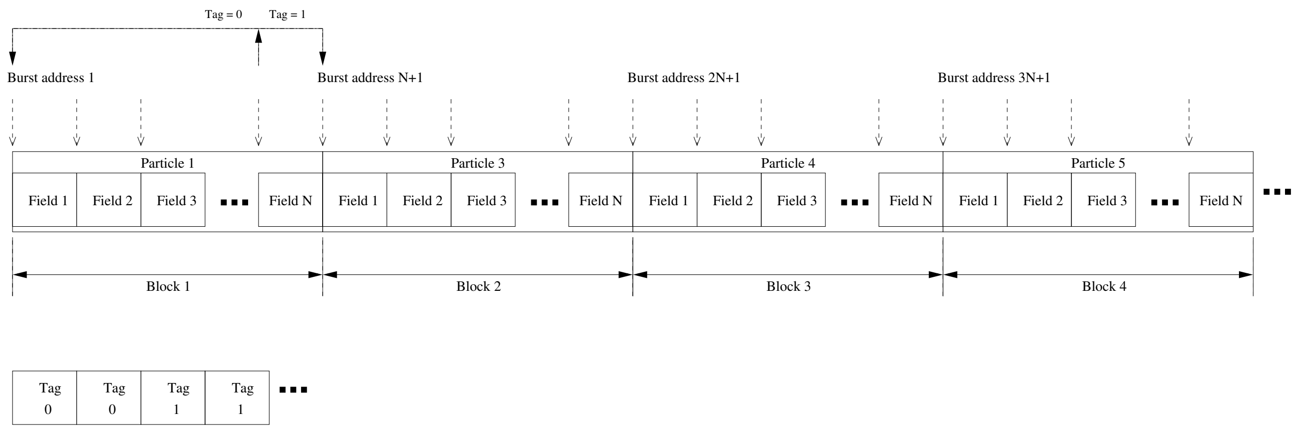
cannot be reduced by implementing more FPGA data paths or increasing the FPGAs' clock frequency because the bottleneck is at the bus connecting the CPUs and FPGAs.

To improve the data transfer performance, we design a data structure which facilitates compression of particles. The idea comes from an observation of the resampling process - some particles are eliminated and the vacancies are filled by replicating non-eliminated particles. Replication means data redundancy exists. For example, in the original data structure shown in Figure 4.6(a), particle 1 has three replicates and particle 2 is eliminated, therefore, particle 1 is stored and transferred for three times.

By using the data structure in Figure 4.6(b), data redundancy is eliminated by storing every particle once. Each particle is also transferred once. As a result, the data transfer time and



(a) Particle stream before compression



(b) Compressed particle stream

Figure 4.6: After the resampling process, some particles are eliminated and the remaining particles are replicated. Data compression is applied so that every particle is stored and transferred once only.

memory space are reduced.

A HRS often contains DRAM which transfers data in burst in order to maximise the memory bandwidth. This works fine with the original data structure where the data are organised as a sequence from the lower address space to the upper. However, using the new data structure, the data access pattern is not sequential anymore, the address can go back and forth. The DRAM controller needs to be modified so that the transfer throughput would not be affected by the change of data access pattern. As illustrated in Figure 4.6(b), a tag sequence is used to indicate the address of the next block. For example, after reading the data of particle 1, the burst address is at N. If the tag is one, the next burst address will point to the address of the next block at $N+1$. Otherwise, the burst address will point to the start address of the current block (which is 1). The data are still addressed in burst so the performance is not degraded.

The data transfer time with compression is shown below. *Rep* is the average number of replication of the particles, and therefore the size of the resampled particle stream is reduced by a ratio of *Rep*. The range of *Rep* is from 1 to P_t, depending on the distribution of particles after the resampling process. The effect of *Rep* on data transfer time will be evaluated in the next section.

$$T_{tran} = \frac{(\frac{df}{Rep} + df + 1) \cdot bw_{data} \cdot P_t}{f_{bus} \cdot lane \cdot eff \cdot N_{FPGA}} \tag{4.15}$$

4.6 Experimental Results

To evaluate the performance of the HRS and make comparison with the other systems, we implement an application which uses PF for localisation and tracking of mobile robot. The application is proposed in [26] to track location of moving objects conditioned upon robot poses over time. Given a priori learned map, a robot receives sensor values and moves at regular time intervals. Meanwhile, M moving objects are tracked by the robot. The states of the robot and objects at time t are represented by a state vector X_t:

$$X_t = \{R_t, H_{t,1}, H_{t,2}, ..., H_{t,M}\} \tag{4.16}$$

R_t denotes the robot's pose at time t, and $H_{t,1}, H_{t,2}, ..., H_{t,M}$ denote the locations of the M objects at the same time.

The following equation is used to represent the posterior of the robot's location:

$$p(X_t|Y_t, U_t) = p(R_t|Y_t, U_t) \prod_{m=1}^{M} p(H_{t,m}|R_t, Y_t, U_t) \tag{4.17}$$

Y_t is the sensor measurement and U_t is the control of the robot at time t. The robot path posterior $p(R_t|Y_t, U_t)$ is represented by a set of robot-particles. The distribution of an object's location $p(H_{t,m}|R_t, Y_t, U_t)$ is represented by a set of object-particles, where each object-particle set is attached to one particular robot-particle. In other words, if there are P_r robot-particles representing the posterior over robot path, there are P_r object-particle sets, each has P_h particles.

In the application, the area of the map is 12m by 18m. The robot makes a movement of 0.5m every five seconds, i.e. $T_{rt} = 5$. The robot can track eight moving objects at the same time. A maximum of 8192 particles are used for robot-tracking and each robot-particle is associated with 1024 object-particles. Therefore, the maximum number of data path cycles is 8*8192*1024=67,108,864. Each particle being streamed into the FPGAs contains coordinates (x,y) and heading h which are represented by three single precision floating-point numbers. For the particle being streamed out of the FPGAs, it also contains a weight in addition to the coordinates. From Equation 4.11, the size of a particle is $(2 \cdot 3 + 1) \cdot 32$ bits $= 224$ bits.

4.6.1 System Settings

HRS: Two reconfigurable accelerator systems from Maxeler Technologies are used. The system is developed using MaxCompiler, which adopts a stream computing model.

- *MaxWorkstation* is a microATX form factor system which is equipped with one Xilinx Virtex-6 XC6VSX475T FPGA. The FPGA has 297,600 lookup tables (LUTs), 595,200 flip-flops (FFs), 2,016 digital signal processors (DSPs) and 1,064 block RAMs. The FPGA board is connected to an Intel i7-870 CPU (4 physical cores, 8 threads in total, clocked at 2.93 GHz) via a PCI Express Gen2 x8 bus. The maximum bandwidth of the PCI Express bus is 2 GB/s according to the specification provided by Maxeler Technologies.

- *MPC-C500* is a 1U server accommodating four FPGA boards, each of which has a Xilinx Virtex-6 XC6VSX475T FPGA. Each FPGA board is connected to two Intel Xeon X5650 CPUs (12 physical cores, 24 threads in total, clocked at 2.66 GHz) via a PCI Express Gen2 x8 bus.

To support run-time reconfigurability, there are two FPGA configurations:

- *Sampling and importance weighting configuration* is clocked at 100 MHz. Two data paths are implemented on one FPGA to process particles in parallel. The total resource usage is 231,922 LUTs (78%), 338,376 FFs (56%), 1,934 DSPs (96%) and 514 block RAMs (48%).
- *Low-power configuration* is clocked at 10 MHz, with 5,962 LUTs (2%), 6,943 FFs (1%) and 12 block RAMs (1%). It uses minimal resources just to maintain communication between the FPGAs and CPUs.

CPU: The CPU performance results are obtained from a 1U server that hosts two Intel Xeon X5650 CPUs. Each CPU is clocked at 2.66 GHz. The program is written in C language and optimised by Intel Compiler with SSE4.2 and flag *-fast* enabled. OpenMP is used to utilise all the processor cores.

GPU: An NVIDIA Tesla C2070 GPU is hosted inside a 4U server. It has 448 cores running at 1.15 GHz and has a peak performance by 1288 GFlops. The program is written in C for CUDA and optimised to use all the cores available. To get more comprehensive results for comparison, we also estimate the performance of multiple GPUs. The estimation is based on the fact that the first three stages (sampling, importance weighting, lower bound calculation) can be evenly distributed to every GPU and be computed independently, so the data path and data transfer speedup scales linearly with the number of GPUs. On the other hand, the last two stages (particle set resizing, resampling) are computed on the CPU no matter how many GPUs are used, therefore, the CPU time does not scale with the number of GPUs.

4.6.2 Adaptive PF versus Non-adaptive PF

The comparison of adaptive and non-adaptive PF is shown in Table 4.1. Both model estimation and experimental results are listed. Initially, the maximum number of particles are instantiated for global localisation. For the non-adaptive scheme, the particle set size does not change. The total computation time estimated and measured are 1.328 seconds and 1.885 seconds, respectively. The difference is due to the difference between the effective and maximum bandwidth of the PCI Express bus.

Table 4.1: Comparison of adaptive and non-adaptive PF on HRS (MaxWorkstation with one FPGA, no data compression is applied)

	Non-adaptive PF		Adaptive PF	
	Model	Exp.	Model	Exp.
No. of particles	67M		573k	
Data path time $T_{datapath}$ (s)	0.336	0.336	0.003	0.003
CPU time T_{CPU} (s)	0.117	0.117	0.001	0.001
Data time T_{tran} (s)	0.875	1.432	0.007	0.012
Total comp. time T_{comp} (s)	1.328	1.885	0.011	0.016
Comp. speedup (higher is better)	1x	1x	120.7x	117.8x

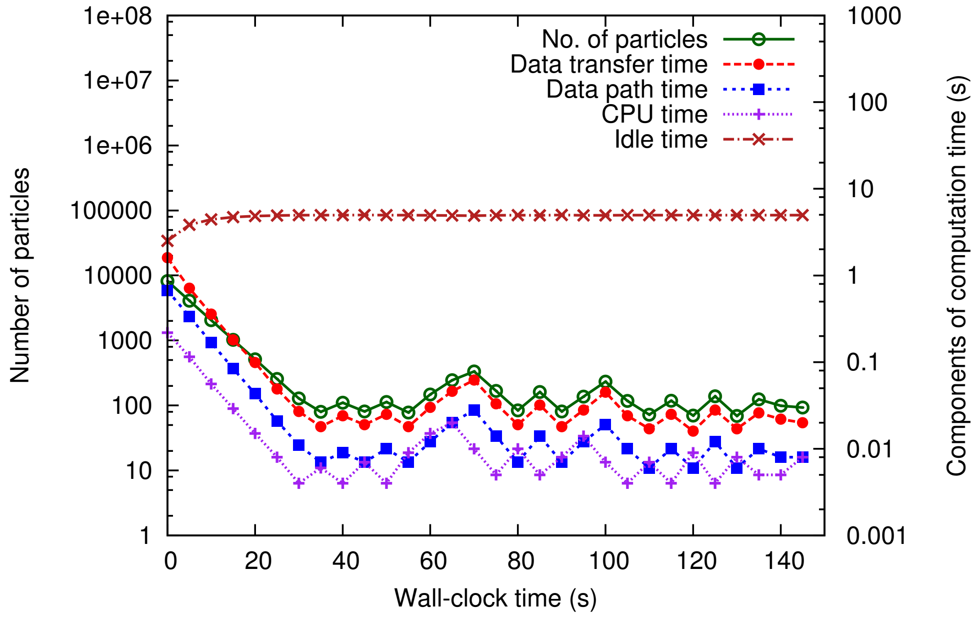


Figure 4.7: Number of particles and components of total computation time versus wall-clock time

For the adaptive scheme, the number of particles varies from 573k to 67M, and the computation time scales linearly with the number of particles. From Table 4.1, both the model and experiment show 99% reduction in computation time.

Figure 4.7 shows how the number of particles and the components of total computation time vary over the wall-clock time (passage of time from the start to the completion of the application). Although the number of particles is reduced in the proposed design, the results in Figure 4.8 show that the localisation error is not adversely affected. The error is the highest during initial global localisation and it is reduced when the robot moves.

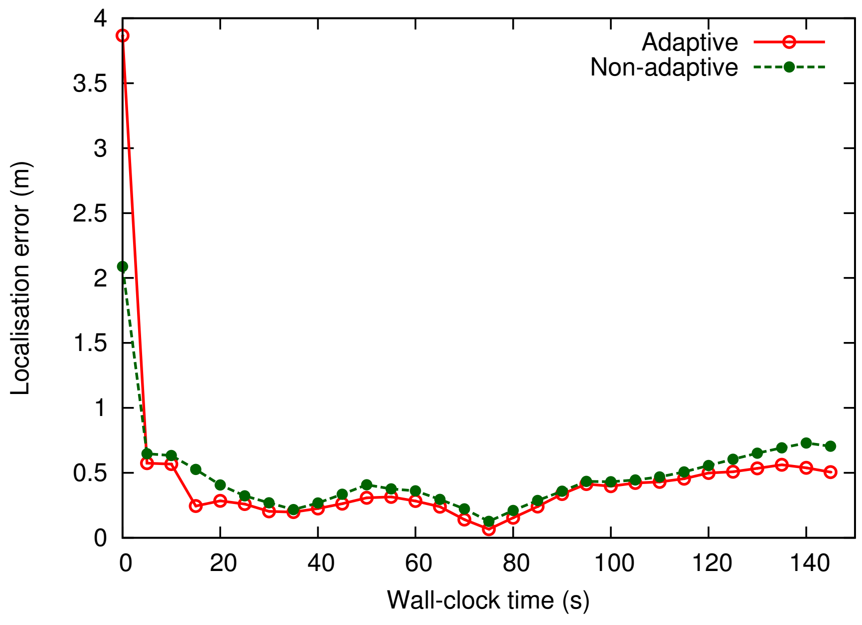


Figure 4.8: Localisation error versus wall-clock time

4.6.3 Data Compression

Figure 4.9 shows the reduction in data transfer time after applying data compression. A higher number of replications means a lower data transfer time. The data transfer time has a lower bound of 0.212 seconds because the data from the FPGAs to the CPUs are not compressible. Only the particle stream after the resampling process is compressed when it is transferred from the CPUs to the FPGAs.

4.6.4 Performance comparison of HRS, CPUs and GPUs

Table 4.2 shows the performance comparison of the CPUs, GPUs and HRS.

Data path time: Considering the time spent on the data paths only, HRS is up to 328 times faster than a single-core CPU and 76 times faster than a 12-core CPU system with 24 threads. In addition, it is 12 times and 3 times faster than one GPU and four GPUs, respectively.

Data transfer time: The data transfer time of HRS is shown in three rows. The first row shows the situation when the PCI Express bandwidth is 2 GB/s. The second row shows the performance when PCI Express gen3 x8 (7.88 GB/s) is used such that the bandwidth is

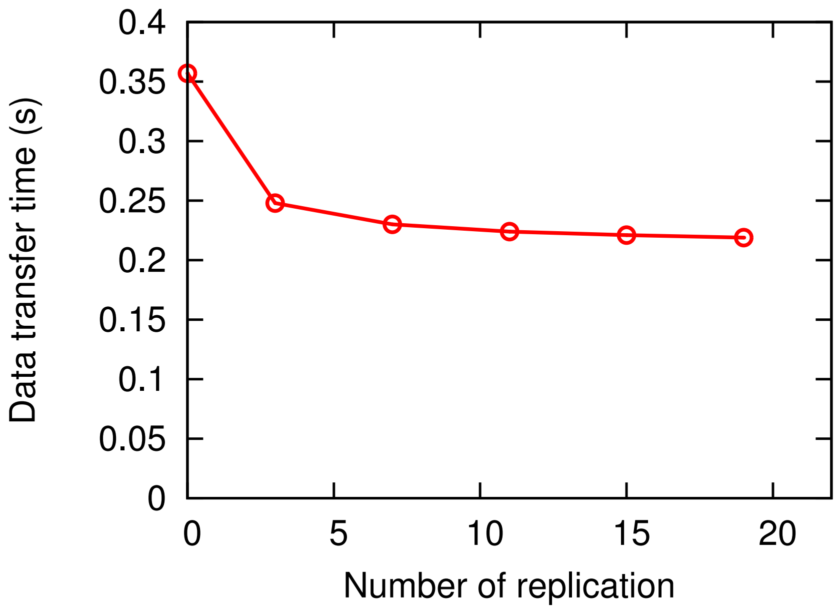


Figure 4.9: Effect on the data transfer time by particle stream compression

comparable with that of the GPU system. When multiple FPGA boards are used, the data transfer time decreases because multiple PCI Express buses are utilised simultaneously. The third row shows the performance when data compression is applied and it is assumed that each particle is replicated for 20 times in average.

CPU time: The CPU time of HRS is shorter than that of the CPU and GPU systems because part of the resampling process of object-particles is performed on the FPGA using Independent Metropolis-Hastings (IMH) resampling algorithm [36]. IMH resampling algorithm is optimised for the deep pipeline architecture where each particle occupies a single stage of the pipeline. On the CPUs and GPU, the computation of the particles are shared by threads and therefore IMH resampling algorithm is not applicable.

Total computation time: Considering the overall system performance, HRS is up to 169 times faster than a single-core CPU, 41 times faster than a 12-core CPU system. In addition, it is 9 times faster than one GPU, and 3 times faster than four GPUs. Notice that the CPUs violate the real-time constraint of 5 seconds.

Power and energy consumption: In real-time applications, we are interested in the energy consumption per time-step. Figure 4.10 shows the power consumption of HRS, CPUs and

Table 4.2: Performance comparison of HRS, CPUs and GPU

	CPU(1) [a]	CPU(2) [a]	GPU(1) [b]	GPU(2) [b]	GPU(3) [b]	HRS(1) [c]	HRS(2) [d]	HRS(3) [d]
Clock freq. (MHz)	2660	2660	1150	1150	1150	100	100	100
Precision	single	single	single	single	single	single + custom	single + custom	single + custom
Level of parallelism	1	24	448	896	1792	2+8 [e]	4+24 [e]	8+24 [e]
Data path time (s)	27.530	6.363	1.000	0.500	0.250	0.336	0.168	0.084
Data path speedup	1x	4.3x	27.5x	55.1x	110.1x	81.9x	163.9x	327.7x
						1.432 [f]	0.716 [f]	0.358 [f]
Data tran. time (s)	0	0	0.360	0.180	0.090	0.363 [g]	0.182 [g]	0.091 [g]
						0.223 [h]	0.111 [h]	0.056 [h]
CPU time (s)	0.420	0.334	0.117	0.117	0.117	0.030	0.025	0.025
Total comp. time (s)	27.95	6.697	1.477	0.797	0.457	0.589	0.304	0.165
Overall speedup	1x	4.2x	18.9x	35.1x	61.2x	47.5x	91.9x	169.4x
Comp. power (W)	183	279	287	424	698	145	420	480
Comp. power eff.	1x	0.7x	0.6x	0.4x	0.3x	1.3x	0.4x	0.4x
Idle power (W)	133	133	208	266	382	95	360	360
Idle power eff.	1x	1x	0.6x	0.5x	0.4x	1.4x	0.4x	0.4x
Energy. (J) [i]	677/5115	673/1868	1041/1157	1331/1456	1911/2054	489/595	1896/1914	1994/2012
Energy eff.	1x	1x/2.7x	0.7x/4.4x	0.5x/3.5x	0.4x/2.5x	1.4x/8.6x	0.4x/2.7x	0.3x/2.5x

[a] 2 Intel Xeon X5650 CPUs @2.66 GHz (12 cores supporting 24 threads).
[b] 1/2/4 NVIDIA Tesla C2070 GPUs and 1 Intel Core i7-950 CPU @3.07 GHz (4 cores supporting 8 threads).
[c] 1 Xilinx XC6VSX475T FPGA and 1 Intel Core i7-870 CPU @2.93 GHz (4 cores supporting 8 threads).
[d] 4 Xilinx XC6VSX475T FPGAs and 2 Intel Xeon X5650 CPUs @2.66 GHz (12 cores supporting 24 threads).
[e] Number of FPGA data paths and number of CPU threads.
[f] Each FPGA communicates with CPUs via a PCI Express bus with 2 GB/s bandwidth.
[g] Each FPGA communicates with CPUs via a PCI Express Gen3 x8 bus with 7.88 GB/s bandwidth.
[h] Each FPGA communicates with CPUs via a PCI Express Gen3 x8 bus with data compression.
[i] Cases for 573k and 67M particles in a 5-second interval.

GPU over a period of 10 seconds (2 time-steps). The system power is measured using a power meter which is connected directly in-line between the power source and the system. All the curves of HRS show peaks when HRS is at the computation mode and troughs when it is at the low power mode. The power during the configuration period lies between the two modes. On the HRS with one FPGA, run-time reconfiguration reduces the idle power consumption by 34% from 145W to 95W. In other words, over a 5-second time-step, the energy consumption is reduced by up to 33%. On the HRS with four FPGAs, the idle power consumption is reduced by 25% from 480W to 360W, and hence the energy consumption decreased by up to 17%.

The run-time reconfiguration methodology is not limited to the Maxeler systems, it can be applied to other FPGA platforms. The resource management software of our system (MaxelerOS) simplifies the effort of performing run-time reconfiguration, and hence we can focus on studying the impact of run-time reconfiguration on energy saving.

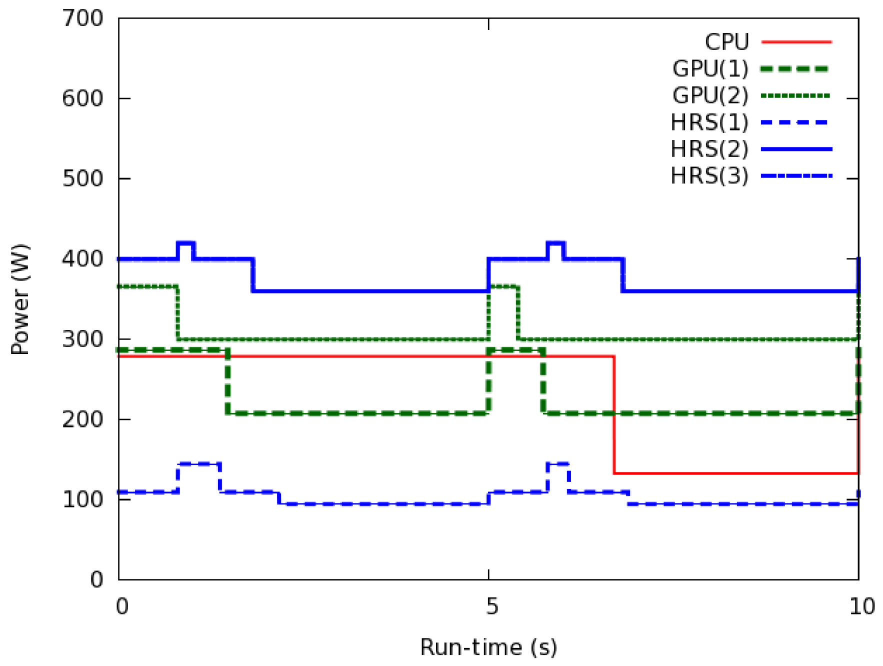


Figure 4.10: Power consumption of HRS, CPU and GPU in one time-step, notice that the computation time of the CPU system exceeds the 5-second real-time requirement (The lines of HRS(2) and HRS(3) overlap)

To identify the speed and energy trade-off, we produce a graph as shown in Figure 4.11. Each data point represents the computation time versus energy consumption of a system setting. Among all the systems, the HRS with one FPGA has the computation speed that satisfies the real-time requirement, while at the same time consumes the smallest amount of energy. All the configurations of CPU system cannot meet the real-time requirement. HRS(3), the HRS with four FPGAs, is the fastest among all the systems in comparison, therefore it is able to handle larger problems and more complex applications.

4.7 Summary

This paper presents an adaptive particle filter for real-time applications. The proposed heterogeneous reconfigurable system demonstrates a significant reduction in power and energy consumption compared with CPU and GPU. The adaptive algorithm reduces computation time while maintaining quality of results. The approach is scalable to systems with multiple FPGAs. A data compression technique is used to mitigate the data transfer overhead between

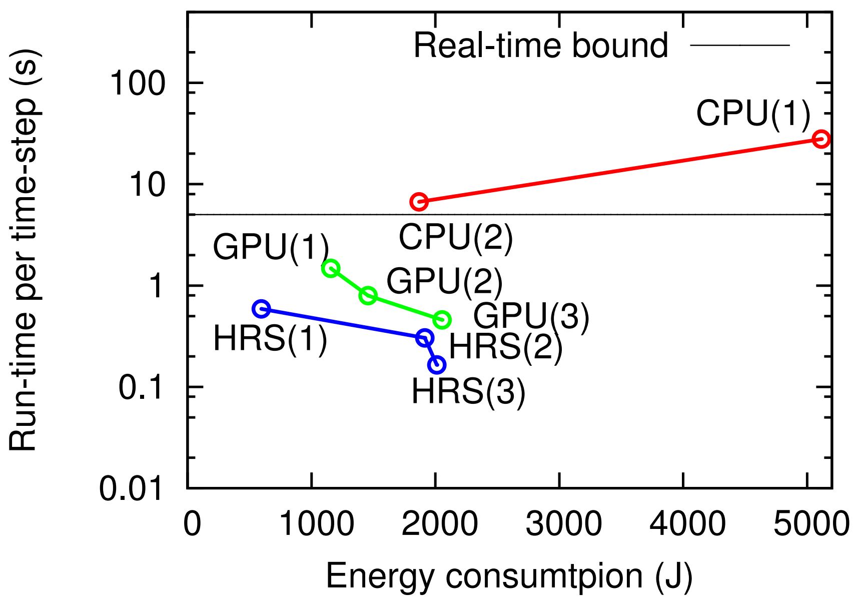


Figure 4.11: Run-time vs. energy consumption of HRS, CPUs and GPUs (5-second time-step, 67M particles; Refer to Table II for system settings)

the FPGAs and CPUs.

In the future, heterogeneous reconfigurable systems will be developed for other particle filters that are more compute-intensive and have more stringent real-time requirements. Air traffic management [3] and traffic estimation [37] are example applications that can substantially benefit from the proposed approach in meeting current and future requirements. Further work will also be required to automate the optimisation of designs targeting heterogeneous reconfigurable systems.

Chapter 5

Design Flow for Sequential Monte Carlo Applications

5.1 Introduction

Sequential Monte Carlo (SMC) methods are a set of on-line posterior density estimation algorithms that perform inference of unknown quantities from observations. The observations arrive sequentially in time and the inference is performed on-line. A common application is in the guidance, navigation and control of vehicles, particularly mobile robots [26] and aircraft [38]. For these applications, it is critical that high sampling rates can be handled in real-time. SMC methods also have applications in economics and finance [39] where minimising latency is crucial.

SMC methods are often preferable to Kalman filters and hidden Markov models, as they do not require exact analytical expressions to compute the evolving sequence of posterior distributions. Moreover, they can model high-dimensional data using non-linear dynamics and constraints, are parallelisable, and can greatly benefit from hardware acceleration. Acceleration of SMC methods has been studied in applications such as air traffic management [3, 40], robot localisation [2], object tracking [41] and signal processing [42].

While SMC has been implemented efficiently on FPGAs [2, 3, 41, 42], design productivity remains a challenge. Firstly, while different sets of SMC parameters produce the same accuracy, they have very different computational complexity. For example, the performance of SMC relies on a set of random samples, which are called particles in the following. The more complex the problem, the larger the number of particles needed. Using excessive numbers of particles unfortunately causes prohibitive run-time without increasing solution accuracy. The parameter space spans multiple dimensions and the objective function can be non-convex, making exhaustive optimisation impractical. Secondly, customising designs for different SMC applications requires tremendous effort.

In this chapter, we propose an SMC design flow for reconfigurable hardware. A computation engine captures the generic control structure shared among all SMC applications. A framework for mapping software to hardware is derived, so users can specify application-specific features which are automatically converted to efficient hardware. Timing model relates design parameters to performance constraints. To enable rapid learning of a large design space, a machine learning algorithm is used to automatically deduce characteristics of the design space.

The contributions are as follows:

- A design flow to reduce the development effort of SMC applications on reconfigurable systems (Section 5.3). Through templating the SMC structure, users can design efficient, multiple-FPGA SMC applications for arbitrary problems without any knowledge of reconfigurable computing, and the software template is open-source.[1]

- A machine learning approach that explores the SMC design space automatically and tunes design parameters to improve performance and accuracy (Section 5.4). The resulting parameters can be applied to the hardware design at run-time without the need for resynthesis. It is demonstrated that parameter optimisation enables the design space to be explored an order of magnitude faster without sacrificing quality. Compared with previous work [2, 3], we have achieved better quality of solutions and faster designs.

[1]Available online: `http://cc.doc.ic.ac.uk/projects/smcgen`

- The benefitbenefit of this approach in terms of design productivity and performance is quantified over a diverse set of SMC problems. Three applications are implemented on Altera and Xilinx-based reconfigurable platforms, with varying numbers of FPGAs. For these problems, the number of lines of code for the FPGA implementation is reduced by approximately 76%, and significant speedup and energy improvement over CPU and GPU implementations (Section 5.5) are demonstrated.

5.2 Overview

5.2.1 SMC Methods

SMC methods estimate the unobserved states of interest based on observations in controlling various agents [29]. The target posterior density $p(s_t|m_t)$ is represented by a set of particles, where s_t is the state and m_t is the observation at time step t. A sequential importance resampling (SIR) algorithm [43] is used to obtain a weighted set of N_P particles $\{s_t^{(i)}, w_t^{(i)}\}_{i=1}^{N_P}$. The importance weights $\{w_t^{(i)}\}_{i=1}^{N_P}$ are approximations to the relative posterior probabilities of the particles such that $\sum_{i=1}^{N_P} w_t^{(i)} = 1$. This process is described in Algorithm 3 and involves five stages of computation:

1. **Initialisation**: Weights $\{w_t^{(i)}\}_{i=1}^{N_P}$ are set to $\frac{1}{N_P}$.
2. **Sampling**: Next states $\{s_t^{'(i)}\}_{i=1}^{N_P}$ are computed based on the current state $\{s_{t-1}^{(i)}\}_{i=1}^{N_P}$.
3. **Importance weighting**: Weight $\{w_t^{(i)}\}_{i=1}^{N_P}$ is updated based on a score function which accounts for the likelihood of particles fitting the observation. Within each iteration $idx1$, the sampling and importance weighting stages are iterated *itl_inner* times so that those particles with sustained benefits are assigned higher weights. As $idx1$ increases, the set of particles reflects a more accurate approximation, so *itl_inner* is increased exponentially.
4. **Resampling**: By removing the particles with small weights and replicating those with large weights *itl_outer* times in a time step, the problem of degeneracy is addressed [44].

Algorithm 3 SMC methods

```
 1: for each time step t do
 2:     $idx1 \leftarrow 0$
 3:     Initialisation
 4:     while $idx1 \leq itl\_outer$ do
 5:         $idx2 \leftarrow 0$
 6:         $itl\_inner \leftarrow 3 + 5\exp(\frac{5*idx1}{itl\_outer})$
 7:         for each particle p do
 8:             while $idx2 \leq itl\_inner$ do
 9:                 Sampling
10:                 Importance weighting
11:                 $idx2 \leftarrow idx2 + 1$
12:             end while
13:         end for
14:         $idx1 \leftarrow idx1 + 1$
15:         if $idx1 \leq itl\_inner$ then
16:             Resampling
17:         end if
18:     end while
19:     Update
20: end for
```

Table 5.1: SMC design parameters. Dynamic: adjustable at run-time; static: fixed at compile-time.

Parameters	Description	Type
itl_outer	Number of iterations of the outer loop	Dynamic
itl_inner	Number of iterations of the inner loop	
N_P	Number of particles	
S	Scaling factor for standard deviation of noise	
H	Prediction horizon	Static
N_A	Number of agents under control	

Without this step, only a small number of particles will have substantial weights for inference.

5. **Update**: State s_t is obtained from the resampled particle set $\{s_t^{(i)}\}_{i=1}^{N_P}$ via weighted average or more complicated functions that will be shown below.

Table 5.1 summarises the parameters of the SMC methods described in Section 5.2.1.

5.2.2 SMC Applications

Stochastic Volatility

These models are used extensively in mathematical finance [45, 46], and describe volatility as a stochastic process which better reflects the behaviour of many financial instruments but are computationally expensive. In this work, the sampling function shown in Equation 5.1 is employed, where y_t is the observable time varying volatility and s_t represents the stochastic log-volatility process. β and ϕ are empirical constants.

$$\begin{aligned} y_t &= \beta \exp(s_t/2)\epsilon_t,\ \epsilon_t \sim \mathcal{N}(0,1) \\ s_t &= \phi s_{t-1} + \mathcal{N}(0,1) \end{aligned} \tag{5.1}$$

The sampling function in Equation 5.2 is implied by Equation 5.1. The state transition from s_{t-1} to s_t is used to draw random samples s_t^i from the existing pool of particles.

$$s_t^i \sim \mathcal{N}(\phi s_{t-1}^i, 1) \tag{5.2}$$

Robot Localisation

SMC methods are applied to mobile robot localisation [26], and this application is used as an example throughout the paper. At regular time intervals, a robot obtains sensor values, identifies its location and commits a move. The robot needs to be aware of the locations of other moving objects in the environment.

The sampling stage is described by Equations 5.3 and 5.4. The robot estimates its updated state s_t based on the current known location (x, y) and heading h. State is affected by external reference status r_t which contains displacement δ and rotation γ. Importance weighting is used to calculate the likelihood of a location based on the observation, i.e. the sensor values.

$$\begin{pmatrix} s_t^i \end{pmatrix} = \begin{pmatrix} x_t^i \\ y_t^i \\ h_t^i \end{pmatrix} = \begin{pmatrix} x_{t-1}^i + \delta_t^{'i} \cos(h_{t-1}^i) \\ y_{t-1}^i + \delta_t^{'i} \sin(h_{t-1}^i) \\ h_{t-1}^i + \gamma_t^{'i} \end{pmatrix} \tag{5.3}$$

$$\begin{pmatrix} r_t^i \end{pmatrix} = \begin{pmatrix} \delta_t^{'i} \\ \gamma_t^{'i} \end{pmatrix} = \begin{pmatrix} \mathcal{N}(\delta_t, \sigma_a^2) \\ \mathcal{N}(\gamma_t, \sigma_b^2) \end{pmatrix} \tag{5.4}$$

Air Traffic Management

SMC methods are applied to model predictive control (MPC) optimisation where control actions at discrete time intervals are determined to minimise error criteria [38]. An example is air traffic management which avoids dangerous encounters by maintaining safe separation distances between aircraft.

At each sampling instant, the control sequence over a number of future time steps, called the prediction horizon H, is estimated. A state is a set of control sequences $\{s_t^{(i),0\ldots H-1}\}_{i=1}^{N_P}$ being picked within a permitted range and applied to the current reference status r_{t-1} to compute the future set of reference statuses $\{r_t^{'(i),0\ldots H-1}\}_{i=1}^{N_P}$. During importance weighting, a score function evaluates the quality of estimation for each particle, and weights the product of scores over the horizon. If any particle violates any constraint, its weight is set to zero. The first control s_t^0 in the sequence, is obtained by selecting the best one among $\{s_t^{(i),0\ldots H-1}\}_{i=1}^{N_P}$. Then the selected control is committed to form reference r_t.

Equation 5.5 illustrates a control tuple that consists of roll angle ϕ; pitch angle τ; and thrust T. Equation 5.6 shows a reference that consists of the current position in 3 dimensional space (x, y, a), heading angle χ, air speed V and mass M. For more details of the model, see [40].

$$\begin{pmatrix} s_t^i \end{pmatrix} = \begin{pmatrix} \phi_t^{'i} \\ \tau_t^{'i} \\ T_t^{'i} \end{pmatrix} = \begin{pmatrix} \mathcal{N}(\phi_t, \sigma_a^2) \\ \mathcal{N}(\tau_t, \sigma_b^2) \\ \mathcal{N}(T_t, \sigma_c^2) \end{pmatrix} \tag{5.5}$$

$$\begin{pmatrix} r_t^{'i} \end{pmatrix} = \begin{pmatrix} x_t^i \\ y_t^i \\ a_t^i \\ \chi_t^i \\ V_t^i \\ M_t^i \end{pmatrix} = \begin{pmatrix} x_{t-1} + V_{t-1}\cos(\chi_{t-1})\cos(\tau_t^{'i}) \\ y_{t-1} + V_{t-1}\sin(\chi_{t-1})\cos(\tau_t^{'i}) \\ a_{t-1} + V_{t-1}\sin(\tau_t^{'i}) \\ \chi_{t-1} + L\sin(\phi_t^{'i})/(M_{t-1}V_{t-1}) \\ V_{t-1} + (\frac{T_t^{'i} - D}{M_{t-1}} - g\sin(\tau_t^{'i})) \\ M_{t-1} - \eta T_t^{'i} \end{pmatrix} \tag{5.6}$$

5.3 SMC Design Flow

This section introduces a design flow for generating reconfigurable SMC designs. The design flow has two novel features to minimise hardware redesign efforts: (1) A generic high-level mapping where application-specific features are specified in a software template and automatically converted to hardware. The template supports the parameter optimisation described in Section 5.4. (2) A parametrisable SMC computation engine which is made up of customisable building blocks and generic control structure that maximises design reuse.

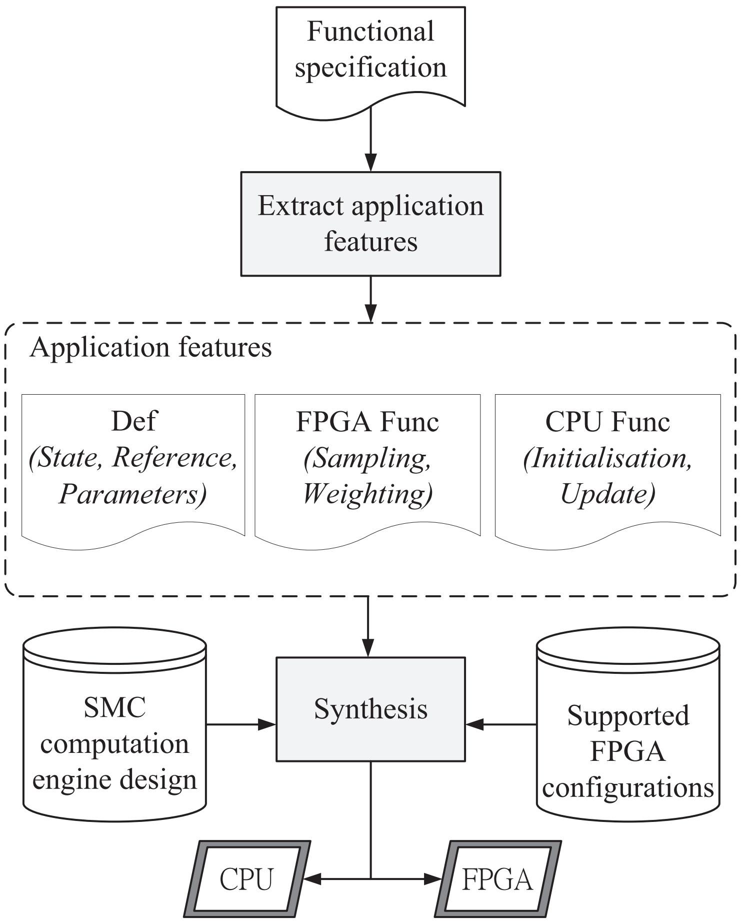


Figure 5.1: Design flow for SMC applications. Users only customise the application-specific descriptions inside the dotted box.

Figure 5.1 shows the proposed design flow. Starting with a functional specification such as software codes or mathematical descriptions, the users identify and code application-specific descriptions (Section 5.3.1). The design flow automatically weaves these descriptions with the computation engine (Section 5.3.2) to form a complete multiple-FPGA system. In this work the synthesis tool employed is Maxeler's MaxCompiler, which uses Java as the underlying language. MaxCompiler also supports FPGAs from multiple vendors, such that low level configurations, such as I/O binding, are performed automatically. Our approach can be extended to support other tools and devices, for example by having the appropriate templates in VHDL or Verilog.

5.3.1 Specifying Application Features

Users create a new SMC design by customising the application-specific Java descriptions inside the dotted box of Figure 5.1. These descriptions correspond to *Def* (Code 1), *FPGA Func* (Code 2) and *CPU Func*.

Def: Code 1 illustrates the class where number representation (floating-point, fixed-point with different bit-width), structs (state, reference), static parameters (Table 5.1) and system parameters are defined. Users are allowed to customise number representation to benefit from the flexibility of FPGA and make trade-off between accuracy and design complexity. State and reference structs determine the I/O interface. Static parameters are defined in this class, while dynamic parameters are provided at run-time. System parameters define device-specific properties such as clock speed and parallelism.

FPGA Func: *Sampling and importance weighting* (line 9 and 10 of Algorithm 3) are the most computation intensive functions, and accelerated by FPGAs. Code 2 illustrates how these two FPGA functions are defined. Given current state s_in, reference r_in and observation m_in (sensor values in this example), an estimation state s_out is computed. Weight w accounts for the probability of an observation from the estimated state. The weight is calculated from the product of scores over the horizon. In this example, the weight is equal to the score as the horizon length is only 1.

CPU Func: *Initialisation and update* are functions running on the CPU. They are responsible for obtaining and formatting data and displaying results. *resampling* is independent of applications so users need not to customise it.

```
public class Def {
  // Number Representation
  static final DFEType float_t =
    KernelLib.dfeFloat(8,24);
  static final DFEType fixed_t =
    KernelLib.dfeFixOffset(26,-20,SignMode.TWOSCOMPLEMENT);
  // State Struct
  public static final DFEStructType state_t = new
  DFEStructType(
    new StructFieldType(''x'', compType);
    new StructFieldType(''y'', compType);
    new StructFieldType(''h'', compType););
  // Reference Struct
  public static final DFEStructType ref_t = new
  DFEStructType(
    new StructFieldType(''d'', compType);
    new StructFieldType(''r'', compType););
  // Static Design parameters (Table I)
  public static int NPMin = 5000, NPMax = 25000;
  public static int H = 1, NA = 1;
  // System Parameters
  public static int NC_inner = 1, NC_P = 2;
  public static int Clk_core = 120, Clk_mem = 350;
  public static int FPGA_resampling = 0, Use_DRAM = 0;
  // Application parameters
  public static int NWall = 8, NSensor = 20;
}
```

Code 1: State, control and parameters for the robot localisation example.

5.3.2 Computation Engine Design

To allow customisation of the computation engine, the engine and data structure are designed as shown in Figure 5.2(a) and 5.2(b) respectively. The computation engine employs a heterogeneous structure that consists of multiple FPGAs and CPUs. FPGAs are responsible for sampling, importance weighting and optionally resampling index generation, and fully pipelined to maximise throughput. To exploit parallelism, particle simulations (sampling and importance weighting) are computed simultaneously by every processing core on each FPGA. Processing cores can be replicated as many times as FPGA resources allow. In situation where the computed results have to be grouped together, data are transferred among FPGAs via the inter-FPGA connection. To maximise the system throughput, remaining non-compute-intensive tasks that involve random and non-sequential data accesses are performed on the CPUs. FP-

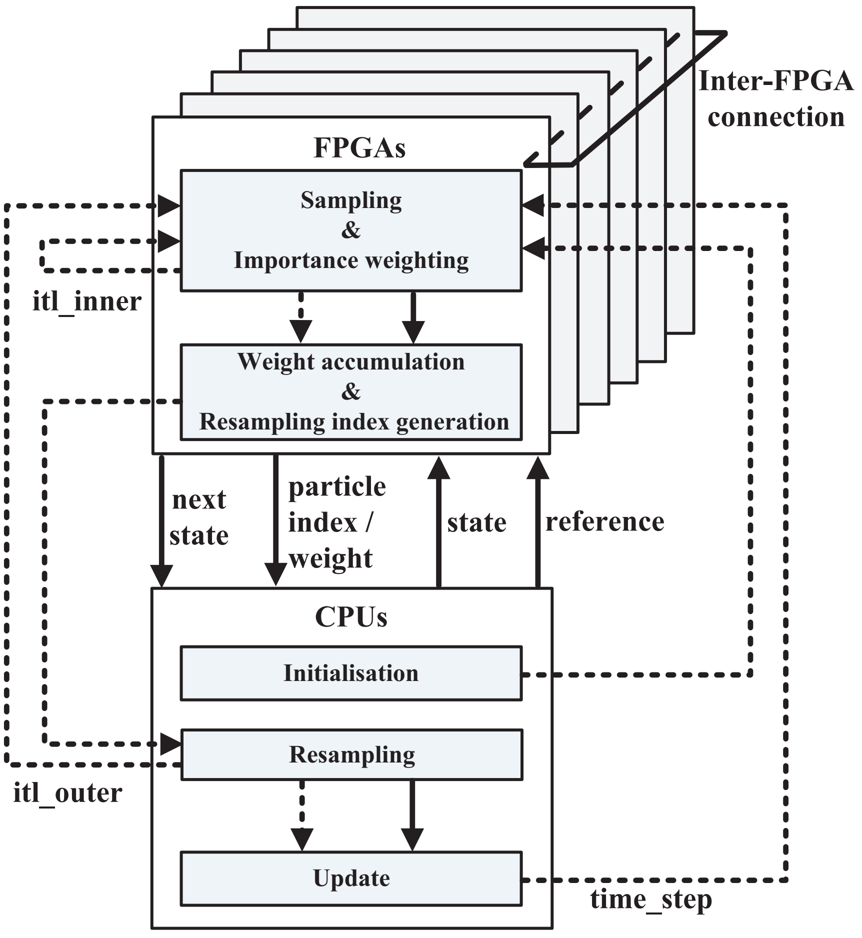
```
public class Func {
  public static DFEStruct sampling(
    DFEStruct s_in, DFEStruct c_in){
    DFEStruct s_out = state_t.newInstance(this);
    s_out.x = s_in.x + nrand(c_in.d,S*0.5) * cos(s_in.h);
    s_out.y = s_in.y + nrand(c_in.d,S*0.5) * sin(s_in.h);
    s_out.h = s_in.h + nrand(c_in.r,S*0.1);
    return s_out;
  }
  public static DFEVar weighting(
    DFEStruct s_in, DFEVar sensor){
    // Score calculation
    DFEVar score = exp(-1*pow(est(s_in)-sensor,2)/S/0.5);
    // Constraint handling
    bool succeed = est(s_in)>0 ? true : false;
    // Weight accumulation
    DFEVar w = succeed ? score : 0; //weight
    return w;
  }
}
```

Code 2: FPGA functions (Sampling and importance weighting) for the robot localisation example.

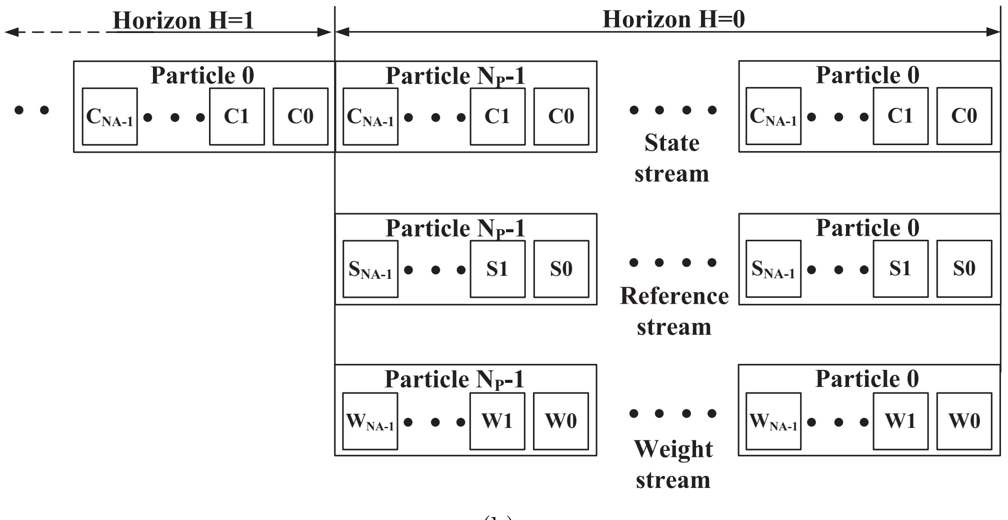
GAs and CPUs communicate through high bandwidth connections such as PCI Express or InfiniBand.

From the control paths (dotted lines) of Figure 5.2(a), we see that there are 3 loops matching Algorithm 3: (1) inner, (2) outer, and (3) time step. First, the inner loop iterates *itl_inner* number of times for *sampling* and *importance weighting*, *itl_inner* increases with the iteration count of the outer loop. Second, the outer loop iterates *itl_outer* times to do *resampling*. The resampling process is performed *itl_outer* times to refine the pool of particles. The particle indices are scrambled after this stage and the indices are transferred to the CPUs to update the particles. Third, the time loop iterates once per time step to obtain a new control strategy and update the current state.

Based on this fact, the data structure shown in Figure 5.2(b) is derived. Each particle encapsulates 3 pieces of information: (1) state, (2) reference, and (3) weight, each being stored as a stream as indicated in the figure. The length of the *state stream* is $N_P \cdot N_A \cdot H$ because each control strategy predicts H steps into the future. The *reference* and *weight streams* have information of N_A agents in N_P particles.



(a)



(b)

Figure 5.2: (a) Design of the SMC computation engine. Solid lines represent data paths while dotted lines represent control paths; (b) Data structure of particles represented by 3 data streams.

Changing the values of *itl_outer*, *itl_inner* and N_P at run-time is allowed since they only affect the length of the particle streams, and not the hardware data path. The computation engine

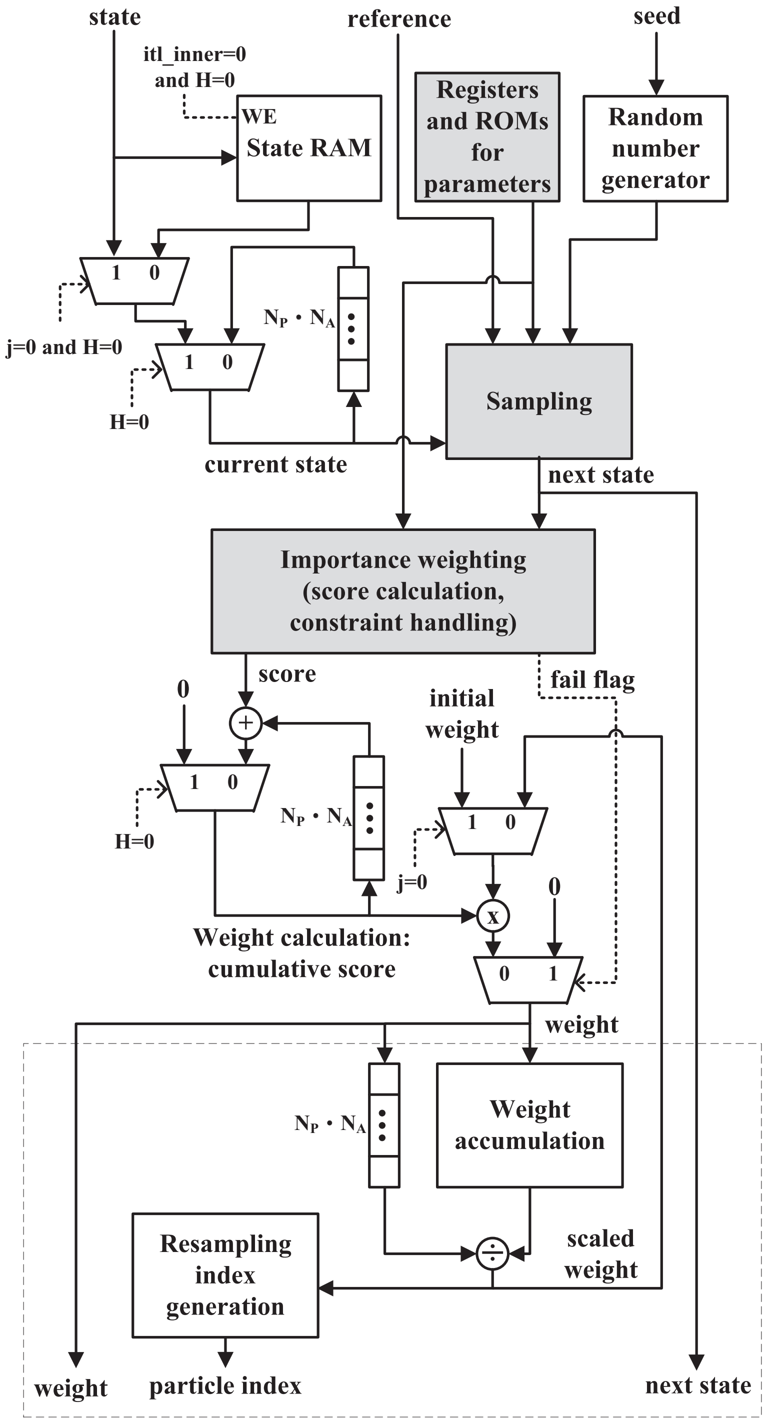


Figure 5.3: FPGA kernel design. The blocks that require users' customisation are darkened. The dotted box covers the blocks that are optional on FPGAs.

is fully pipelined and outputs one result per clock cycle.

Figure 5.3 shows the design of the FPGA kernel. Blocks that require customisation are darkened. The sampling function in Code 2 is mapped to the **Sampling** block which accepts a state and a reference on each clock cycle and calculates the next state on the prediction horizon. After getting a state from the CPU at the beginning ($itl\_inner = 0$ and $H = 0$), the data will be used by the kernel $itl\_inner \cdot N_P$ times. An optional *state RAM* enables reuse of state data and improve performance when the value of *itl_inner* is large. An array of LUT-based random number generators [47, 48] is seeded by CPU to provide random variables; application parameters are stored in registers; and a feedback path stores the state of the previous $N_P \cdot N_A$ cycles.

The **Importance weighting** block computes in 3 steps. Firstly, *Score calculation* uses the states from the *Next state* block to calculate scores of all the states over the horizon. A feedback loop of length $N_P \cdot N_A$ stores the cost of the previous horizon and accumulates the values. Secondly, *Constraint handling* uses the states from the *Next state* block to check the constraints. The block raises a fail flag if a constraint is violated. Lastly, *Weight calculation* combines the scores of the states over the horizon.

Part of the resampling process is handled by the **Resampling index generation** and **weight accumulation** blocks. Weights are accumulated to calculate the cumulative distribution function, then particles indices are reordered. These 2 blocks can either be computed on FPGAs or CPUs.

All the blocks allow precision customisation using fixed-point or floating-point number representation. Users have the flexibility to make trade-off between result accuracy and design complexity.

5.3.3 Performance Model

We derive a performance model to analyse the effect of parameters on the processing speed and resource utilisation of the computation engine. It will be used in Section 5.4 for parameter optimisation.

The processing time of a time step is shown in Equation 5.7. It has 4 components which are iterated *itl_outer* times.

$$T_{step} = itl\_outer \cdot (T_{s\&i} + T_{resample} + T_{cpu} + T_{transfer}) \tag{5.7}$$

$T_{s\&i}$ is the time spent on sampling and importance weighting in the FPGA kernels. Since the data is organised as a stream as described in Section 5.3.2, the time spent on sampling and importance weighting is linear with N_P, N_A and H. It is iterated *itl_inner* times in the inner loop. The sampling and importance weighting process can be accelerated using multiple cores, such that each of them is responsible for part of the inner loop iterations or particles. N_C represents the number of processing cores being used on one FPGA, and N_{Board} is the number of FPGA boards being used. $min(1, \frac{bandwidth}{sizeof(state) \cdot freq})$ accounts for the limitation of bandwidth between FPGAs and CPUs.

$$T_{s\&i} = \frac{itl\_inner \cdot N_P \cdot N_A \cdot H}{N_C \cdot N_{Board} \cdot freq} \cdot \min\left(1, \frac{bandwidth}{sizeof(state) \cdot freq}\right) \tag{5.8}$$

$T_{resample}$ is the time spent on generating the resampling indices. It takes $N_P \cdot PW + N_P \cdot N_A$ cycles to generate the cumulative probability distribution function, and a further $3 \cdot PL \cdot N_P$ cycles to generate particle indices. PW and PL are the length of the pipelines. $T_{resample}$ can be omitted if resampling is processed by the CPUs.

$$T_{resample} = \frac{N_P \cdot PW + N_P \cdot N_A + 3 \cdot PL \cdot N_P}{freq} \tag{5.9}$$

T_{cpu} is the time spent on resampling and updating the current state on the CPUs. The time is related to the amount of data and the speed of the CPU. α_1 is the scaling factor of the CPU speed.

$$T_{cpu} = \alpha_1 \cdot H \cdot N_P \cdot N_A \tag{5.10}$$

$T_{transfer}$ is the data transfer time that accounts for the time taken to transfer the state stream between CPUs and DRAM on an FPGA board. $T_{transfer}$ can be omitted if no DRAM is used.

$$T_{transfer} = \frac{N_P \cdot N_A \cdot (H \cdot sizeof(state))}{bandwidth} \tag{5.11}$$

5.4 Optimising SMC Computation Engine

The design parameters in Table 5.1 have great impact on the performance. 3 questions manifest when finding optimised customisation of the engine: **(1) Which sets of parameters have the best accuracy? (2) For the same accuracy, which sets of parameters meet the timing requirement? (3) How can we reduce the design parameter exploration time?**

5.4.1 Effect of the Design Parameters

Referring to Table 5.1, the SMC computation engine has up to 6 design parameters, each of which adds a dimension to the design space. It is ineffective to exhaustively search for the best set of parameters. Furthermore, the performance curve of each dimension can be non-linear and constrained by the real-time requirement and FPGA resources.

To answer **questions 1 and 2**, consider the robot localisation application. Its solution quality is measured by the root mean square error (RMSE) in localisation. We study the effect of changing design parameters using the functional specification in Figure 5.1, e.g. a C program. Its fast build time helps us to perform analysis effectively but its performance is too slow for real-time operation. The timing model described in Section 5.3.3 estimates the run-time of the FPGA implementation.

When N_P and *itl_outer* are explored together as shown in Figure 5.4, we see an uneven surface. Although non-linear, the trend of RMSE decreasing as N_P and *itl_outer* are increased is evident. The valid parameter space is constrained by the real-time requirement. The parameter space is darkened for those parameters leading to an RMSE greater than 1 m (Question 1). Moreover,

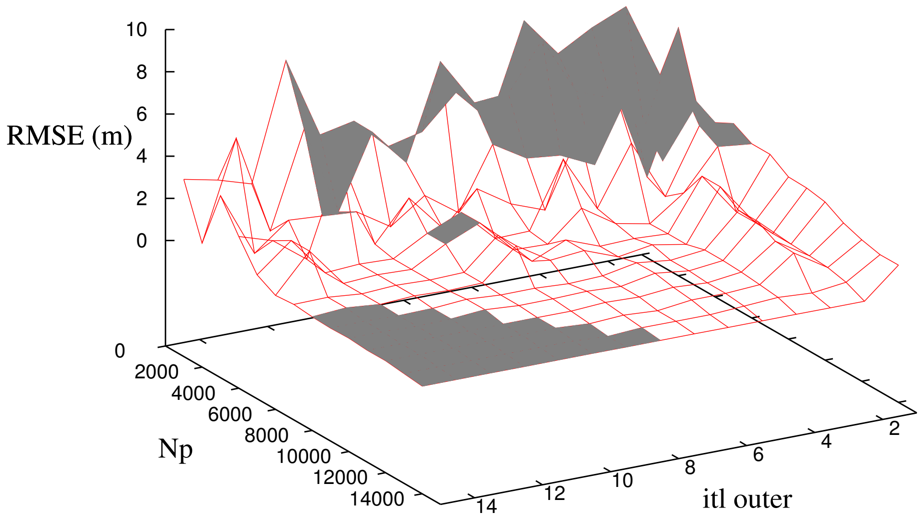


Figure 5.4: Parameter space of robot localisation system (N_A=8192, S=1). The dark region on the top-right indicates designs which fail localisation accuracy constraints, while those on the bottom-left indicates designs which fail real-time requirements.

the dark region with a run-time longer than the 5 seconds real-time requirement is marked as invalid (Question 2).

If the value of S is considered, the parameter optimisation problem expands to 3 dimensions as shown in Equation 5.12.

$$\begin{aligned} \text{minimise } RMSE &= f(N_P, itl\_outer, S) \\ \text{subject to } RMSE &\leq 1 \text{ m}, T_{step} \leq 5\text{s}, \end{aligned} \tag{5.12}$$

5.4.2 Parameter Optimisation

Now we come to **question 3**, the parameter optimisation problem, which is difficult as construction of an analytical model combining timing and quality of solution is either impossible or very time consuming. Furthermore the design space is constrained by multiple accuracy and real-time requirements. We cannot use a design unless the results are within certain error bound. The problem is further aggravated by the curse of dimensionality. We use an automated design exploration approach which is facilitated by a machine learning algorithm developed in [49]. The approach allows the performance impact of different parameters to be

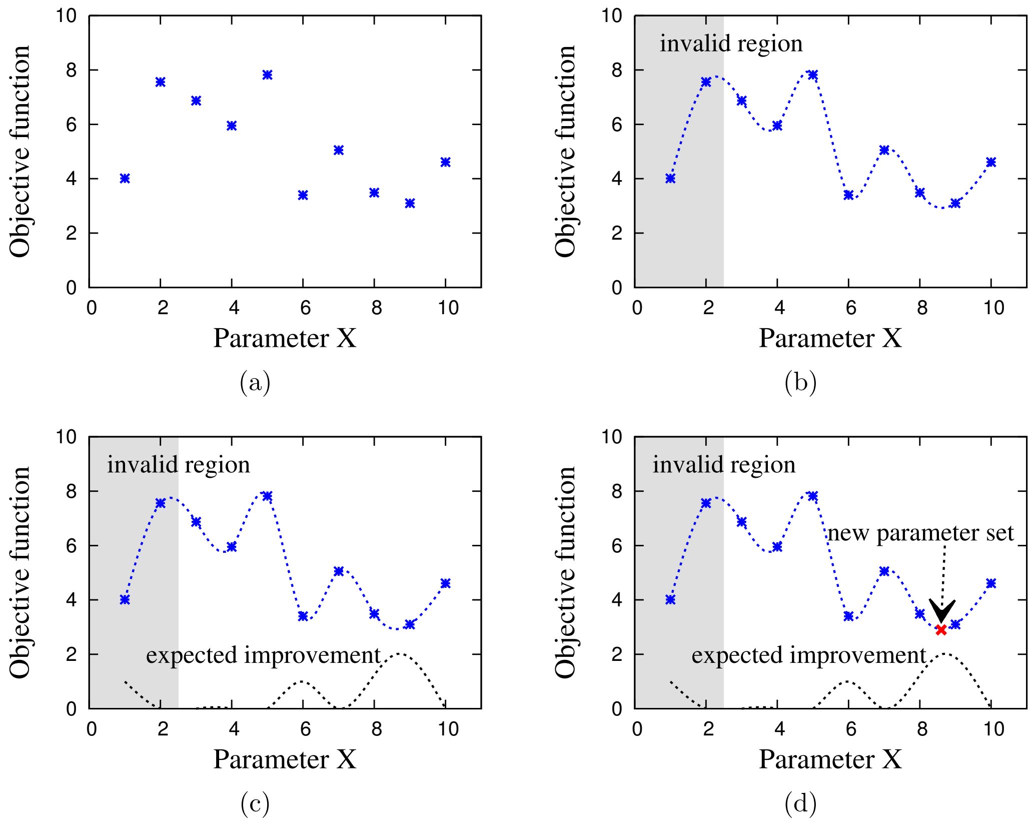


Figure 5.5: Illustration of automatic parameter optimisation: (a) Sampling parameter sets; (b) Building surrogate model; (c) Calculating expected improvement; (d) Moving to the point offering the highest improvement.

determined for any design based on our SMC computation engine.

A surrogate model is employed to enable rapid learning of the valid design space and deal with a large number of parameters. The idea is illustrated in Figure 5.5. Firstly, a number of randomly sampled designs is evaluated (Figure 5.5(a)). Secondly, the results obtained during evaluations are used to build a surrogate model. The model provides a regression of a fitness function and identifies regions of the parameter space which fail any of the constraints (Figure 5.5(b)). Thirdly, the surrogate model output is used to calculate the expected improvement (Figure 5.5(c)). Finally, the exploration converges to the parameter set that is expected to offer the highest improvement. Parameter sets in the invalid region are disqualified (Figure 5.5(d)).

Our SMC computation engine is made customisable to improve productivity of application builders who target FPGAs, based on an optimisation approach which is already applicable to CPUs and GPUs.

5.5 Evaluation

5.5.1 Design Productivity

We first analyse how the proposed design flow can reduce design effort. In Table 5.2, user-customisable code is classified into three parts: (a) *Def* is the definition of state, reference and parameters. (b) *FPGA Func* is the description of sampling and importance weighting functions. (c) *CPU Func* is the initiation, resampling and update part running on CPU. On average, users only need to customise 24% of the source code. Moreover, automatic design space optimisation greatly saves overall design time. As we will see in the applications below, we are able to choose the optimal set of parameters without conducting an exhaustive search.

Table 5.2: Lines of code for 3 SMC applications under the proposed design flow.

	Custom codes			All codes	Custom %
	Def	FPGA Func	CPU Func		
Sto. vol.	31	44	84	1,164	13.7
Robot loc.	54	143	56	1,113	22.7
Air traffic	45	360	70	1,360	35.0

5.5.2 Application 1: Stochastic Volatility

The stochastic volatility model was described using the flow Our design flow is used in targeting a stochastic volatility model to a XIlinx Virtex-6 XC6VSX475T FPGA at 150 MHz. Parallel single precision floating-point data paths are used to maximise resource utilisation and hence performance. Limited by I/O constraints, 16 processing cores are chosen. The resulting design uses 70,674 LUTs (24%), 448 DSPs (22%) and 394 block RAMs (19%). The CPU is an Intel Core i7 870 quad-core processor clocked at 2.93GHz.

The design space has 2 dimensions, N_P and S (Table 5.1). Out of 420 sets of design parameters, the machine learning approach evaluates 20 of the candidates, and obtains an optimal set of parameters $N_P = 768, S = 1.5$ which minimises the estimation error.

Table 5.3 summarises the performance of CPU and reconfigurable systems using the same set of tuned parameters. Both systems have the same microATX form factor for fair comparison. Since the data size being processed is very small, the processing time of reconfigurable system is dominated by the overhead of invoking the FPGA kernel.

Table 5.3: Performance comparison of stochastic volatility.

	CPU [a]	This work [b]
Clock frequency (MHz)	2,930	150
Number of cores	4	16
Run-time per step (ms)	0.05	0.5
Power (W)	120	140
Energy (mJ)	6	70

[a] Intel Core i7 870 CPU, optimised by Intel Compiler with SSE4.2 and flag *-fast* enabled.

[b] Maxeler MaxWorkstation with Xilinx Virtex-6 XC6VSX475T FPGA and Intel Core i7 870 CPU, developed using MaxCompiler.

5.5.3 Application 2: Mobile Robot Localisation

Now we look at an application with larger data set. For this example the same reconfigurable system as application 1 is used. Two processing cores are instantiated in an FPGA. Core computation in the sampling and importance weighting process is implemented using fixed-point arithmetic to optimise resource usage. The result utilises 148,431 LUTs (50%), 1,278 DSPs (63%) and 549 block RAMs (26%).

The design space has 3 dimensions: *itl_outer*, N_P and S. Out of 945 sets of parameters, 52 sets are evaluated to minimise the localisation error within the 5 seconds real-time constraint.

Table 5.4 compares the performance of our reconfigurable system with CPU, GPU and a previous system in [2] which has not been optimised by our proposed approach. The reconfigurable system is 8.9 times and 1.2 times faster than the CPU and GPU, respectively. With parameter optimisation that maximise accuracy, our work achieves a better RMSE than the previous work (0.15m vs. 0.52m).

Table 5.4: Performance comparison of robot localisation.

	CPU opt. [a]	This work opt. [b]	Ref. sys. [2] w/o opt. [b]	GPU opt. [c]
Clk. freq. (MHz)	2,930	120	100	1,150
Number of cores	4	2	2	448
Run-time / step (s)	33.1	3.7	1.6	4.5
RMSE (m)	0.15	0.15	0.52	0.15
Power (W)	130	145	145	287
Energy (kJ)	4.3	0.54	0.23	1.29

[a,b] Refer to configurations in Table 5.3.
[c] NVIDIA Tesla C2070 GPU, developed using CUDA programming model.
[d] Parameters with optimisation: itl_outer=2, N_P=14000, S=1.2;
Parameters without optimisation: itl_outer=1, N_P=8192, S=1.

5.5.4 Application 3: Air Traffic Management

The air traffic management system is able to control 20 aircraft simultaneously. The FPGA part runs on a 1U machine hosting 6 Altera Stratix V GS 5SGSD8 FPGAs clocked at 220 MHz, each of which has a single precision floating-point data path that consumes 166,008 LUTs (63%), 337 multipliers (9%) and 1,528 block RAMs (60%). The CPU part runs on 2 Intel Xeon E5-2640 CPUs clocked at 2.53GHz. Both parts are connected via InfiniBand.

This application has 4 design parameters leading to a space with 4000 sets of parameters. The optimisation target is to minimise the time of aircraft spending in the air traffic control region. Machine learning reduces the number of evaluations to 1% as indicated in Table 5.5. Hence, the parameter optimisation time is reduced from days to hours.

Table 5.5: Parameter optimisation of air traffic management system using machine learning approach.

N_A	Parameter sets evaluated / total	Parameter set obtained			
		itl_outer	H	N_P	S
4	41 / 4000	20	5	500	0.1
20	31 / 4000	100	8	5000	0.05

Table 5.6 summarises the performance of the CPU, GPU and reconfigurable system. To ensure fair comparisons, we scale the CPU and GPU systems to similar form factors with the reconfigurable system. The scaling is based on the fact that the sampling and importance weighting process is evenly distributed to every GPU and computed independently, while the resampling

Table 5.6: Performance comparison of air traffic management.

		CPU opt. [a]	GPU opt. [b]	This work opt. [c]	Ref. FPGA [3] w/o opt. [d]
	Clk. freq. (MHz)	2,660	1,150	220	150
	Number of cores	24	1,792	6	5
	Power (W)	550	1100	600	N/A
4 aircraft	Run-time / step (s)	0.80	0.12	0.03	2.2
	Energy (kJ)	0.44	0.13	0.02	N/A
20 aircraft	Run-time / step (s)	198	28.25	11.6	N/A
	Energy (kJ)	108.90	29.95	7.0	N/A

[a] 4 Intel Xeon X5650 CPUs (scaled), optimised by Intel Compiler with SSE4.2 and flag *-fast* enabled.
[b] 4 NVIDIA Tesla C2070 GPUs (scaled), developed using CUDA programming model.
[c] Maxeler MPC-X2000, with 6 Altera Stratix V GS 5SGSD8 FPGAs and 2 Intel Xeon X5650 CPUs, developed using MaxCompiler.
[d] Altera Stratix IV EP4SGX530 FPGA.
[e] Parameters with optimisation: refer to Table 5.5;
Parameters without optimisation: itl_outer=100, N_P=1024, S=0.05, H=6.

process is computed on the CPU no matter how many GPUs are used. The reconfigurable platform is faster and more energy efficient than the other systems.

We also compare the performance of our work with a reference implementation that uses an Altera Stratix IV FPGA [3]. That implementation is only large enough to support 4 aircraft and it does not have the flexibility to tune parameters without re-compilation. Our design exploration approach is able to select the set of parameters that produces the same quality of results and is up to 73 times faster.

5.6 Summary

This paper demonstrates the feasibility of generating highly-optimised reconfigurable designs for SMC applications, while hiding detailed implementation aspects from the user. A software template makes the computation engine portable and facilitates code reuse, the number of lines of user-written code being decreased by approximately 76% for an application. We further establish that a surrogate software model combined with machine learning can be used to rapidly optimise designs, reducing optimisation time from days to hours; and that the resulting parameters can be utilised without resynthesis.

Ongoing and future work is focused on incorporating device-specific parameters, such as the level of parallelism and clock speed, into the machine learning approach [49]. We are currently investigating run-time optimisation of parameters based on our initial work [2]. We will also automate the design flow to allow translation of designs captured in software programming languages (e.g. R, MATLAB) to reconfigurable implementations, and extend the software template in VHDL/Verilog to support a wider range of systems.

Chapter 6

Conclusion

6.1 Summary of Thesis Achievements

Summary.

6.2 Applications

Applications.

6.3 Future Work

Future Work.

Bibliography

[1] T. C. P. Chau, W. Luk, P. Y. K. Cheung, A. Eele, and J. M. Maciejowski, “Adaptive sequential Monte Carlo approach for real-time applications,” in *Proceedings of International Conference Field Programmable Logic and Applications*, 2012, pp. 527–530.

[2] T. C. P. Chau, X. Niu, A. Eele, W. Luk, P. Y. K. Cheung, and J. M. Maciejowski, “Heterogeneous reconfigurable system for adaptive particle filters in real-time applications,” in *Proceedings of International Symposium Applied Reconfigurable Computing*, 2013, pp. 1–12.

[3] T. C. P. Chau, J. S. Targett, M. Wijeyasinghe, W. Luk, P. Y. K. Cheung, B. Cope, A. Eele, and J. M. Maciejowski, “Accelerating sequential Monte Carlo method for real-time air traffic management,” *SIGARCH Computer Architecture News*, vol. 41, no. 5, 2013.

[4] E. G. Gilbert and C. P. Foo, “Computing the distance between general convex objects in three-dimensional space,” *IEEE Transactions on Robotics and Automation*, vol. 6, no. 1, pp. 53–61, 1990.

[5] N. Chakraborty, J. Peng, S. Akella, and J. E. Mitchell, “Proximity queries between convex objects: An interior point approach for implicit surfaces,” *IEEE Transactions on Robotics*, vol. 24, no. 1, pp. 211–220, 2008.

[6] K.-W. Kwok, V. Vitiello, and G.-Z. Yang, “Control of articulated snake robot under dynamic active constraints,” in *Proceedings of International Conference Medical image computing and computer-assisted intervention*, 2010, pp. 229–236.

[7] M. Li, M. Ishii, and R. H. Taylor, “Spatial motion constraints using virtual fixtures generated by anatomy,” *IEEE Transactions on Robotics*, vol. 23, no. 1, pp. 4–19, 2007.

[8] D. Constantinescu, S. E. Salcudean, and E. A. Croft, “Haptic rendering of rigid contacts using impulsive and penalty forces,” *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 309–323, 2005.

[9] M. Jakopec, F. Rodriguez y Baena, S. Harris, P. Gomes, J. Cobb, and B. L. Davies, “The hands-on orthopaedic robot ”acrobot”: Early clinical trials of total knee replacement surgery,” *IEEE Transactions on Robotics and Automation*, vol. 19, no. 5, pp. 902–911, 2003.

[10] M. Benallegue, A. Escande, S. Miossec, and A. Kheddar, “Fast c1 proximity queries using support mapping of sphere-torus-patches bounding volumes,” in *Proceedings of International Conference Robotics and Automation*, 2009, pp. 483–488.

[11] K.-W. Kwok, K. H. Tsoi, V. Vitiello, J. Clark, G. C. T. Chow, W. Luk, and G.-Z. Yang, “Dimensionality reduction in controlling articulated snake robot for endoscopy under dynamic active constraints,” *IEEE Transactions on Robotics*, vol. 29, no. 1, pp. 15–31, 2013.

[12] X. Zhang and Y. J. Kim, “Interactive collision detection for deformable models using streaming aabbs,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 2, pp. 318–329, 2007.

[13] G. C. T. Chow, K. W. Kwok, W. Luk, and P. H. W. Leong, “Mixed precision processing in reconfigurable systems,” in *Proceedings of International Symposium Field-Programmable Custom Computing Machines*, 2011, pp. 17–24.

[14] E. Gilbert, D. W. Johnson, and S. S. Keerthi, “A fast procedure for computing the distance between complex objects in three-dimensional space,” *IEEE Journal of Robotics and Automation*, vol. 4, no. 2, pp. 193–203, 1988.

[15] B. Mirtich and B. Mirtich, “V-clip: Fast and robust polyhedral collision detection,” *ACM Transactions on Graphics*, vol. 17, pp. 177–208, 1998.

[16] M. C. Lin and J. F. Canny, “A fast algorithm for incremental distance calculation,” in *Proceedings of International Conference Robotics and Automation*, 1991, pp. 1008–1014.

[17] C. F. Fang, R. A. Rutenbar, and T. Chen, “Fast, accurate static analysis for fixed-point finite-precision effects in dsp designs,” in *Proceedings of International Conference Computer-aided Design*, 2003, pp. 275–282.

[18] D.-U. Lee, A. Abdul Gaffar, W. Luk, and O. Mencer, “MiniBit: Bit-width optimization via affine arithmetic,” in *Proceedings of Design Automation Conference*, 2005, pp. 837–840.

[19] W. G. Osborne, J. Coutinho, R. C. C. Cheung, W. Luk, and O. Mencer, “Instrumented multi-stage word-length optimization,” in *Proceedings of International Conference Field-Programmable Technology*, 2007, pp. 89–96.

[20] D. Boland and G. A. Constantinides, “Automated precision analysis: A polynomial algebraic approach,” in *Proceedings of International Symposium Field-Programmable Custom Computing Machines*, 2010, pp. 157–164.

[21] J. Ponce, D. Chelberg, and W. B. Mann, “Invariant properties of straight homogeneous generalized cylinders and their contours,” *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 11, no. 9, pp. 951–966, 1989.

[22] F. P. Preparate and M. I. Shamos, *Computational Geometry.* Springer, 1985.

[23] E. Weisstein, “Point-line distance–3-dimensional,” http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html.

[24] L. Fousse, G. Hanrot, V. Lefère, P. Péissier, and P. Zimmermann, “MPFR: A multiple-precision binary floating-point library with correct rounding,” *ACM Transactions on Mathematical Software*, vol. 33, no. 2, pp. 13:1–13:15, 2007.

[25] M. Happe, E. Lübbers, and M. Platzner, “A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking,” *Journal of Real-Time Image Processing*, pp. 1–16, 2011.

[26] M. Montemerlo, S. Thrun, and W. Whittaker, "Conditional particle filters for simultaneous mobile robot localization and people-tracking," in *Proceedings of International Conference Robotics and Automation*, 2002, pp. 695–701.

[27] J. Vermaak, C. Andrieu, A. Doucet, and S. J. Godsill, "Particle methods for bayesian modeling and enhancement of speech signals," *IEEE Transactions on Speech and Audio Processing*, vol. 10, no. 3, pp. 173–185, 2002.

[28] A. Eele and J. M. Maciejowski, "Comparison of stochastic optimisation methods for control in air traffic management," in *Proceedings of IFAC World Congress*, 2011.

[29] A. Doucet, N. de Freitas, and N. Gordon, *Sequential Monte Carlo methods in practice.* Springer, 2001.

[30] M. Bolic, P. M. 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.

[31] D. Koller and R. Fratkina, "Using learning for approximation in stochastic processes," in *Proceedings of International Conference Machine Learning*, 1998, pp. 287–295.

[32] D. Fox, "Adapting the sample size in particle filters through KLD-sampling," *International Transactions on Robotics*, vol. 22, no. 12, pp. 985–1003, 2003.

[33] S.-H. Park, Y.-J. Kim, and M.-T. Lim, "Novel adaptive particle filter using adjusted variance and its application," *International Journal of Control, Automation and Systems*, vol. 8, no. 4, pp. 801–807, 2010.

[34] M. Bolic, S. Hong, and P. M. Djuric, "Performance and complexity analysis of adaptive particle filtering for tracking applications," in *Proceedings of Asilomar Conference Signals, Systems, and Computers*, vol. 1, 2002, pp. 853–857.

[35] Z. Liu, Z. Shi, M. Zhao, and W. Xu, "Mobile robots global localization using adaptive dynamic clustered particle filters," in *Proceedings of International Conference Intelligent Robots and Systems*, 2007, pp. 1059–1064.

[36] 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 Processing Systems*, vol. 65, no. 2, pp. 211–227, 2011.

[37] L. Mihaylova, R. Boel, and A. Hegyi, "Freeway traffic estimation within particle filtering framework," *Automatica*, vol. 43, no. 2, pp. 290–300, 2007.

[38] N. Kantas, J. M. Maciejowski, and A. Lecchini-Visintini, "Sequential Monte Carlo for model predictive control," in *Nonlinear Model Predictive Control*, ser. Lecture Notes in Control and Information Sciences, 2009, pp. 263–273.

[39] D. Creal, "A survey of sequential Monte Carlo methods for economics and finance," *Econometric Reviews*, vol. 31, no. 3, pp. 245–296, 2012.

[40] A. Eele, J. M. Maciejowski, T. C. P. Chau, and W. Luk, "Parallelisation of sequential Monte Carlo for real-time control in air traffic management," in *Proceedings of International Conference Decision and Control*, 2013.

[41] M. Happe, E. Lübers, and M. Platzner, "A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking," *Journal of Real-Time Image Processing*, vol. 8, no. 1, pp. 95–110, 2013.

[42] G. Hendeby, J. Hol, R. Karlsson, and F. Gustafsson, "A graphics processing unit implementation of the particle filter," in *Proceedings of European Signal Processing Conference*, 2007, pp. 1639–1643.

[43] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *Proceedings of Radar and Signal Processing*, vol. 140, no. 2, pp. 107–113, 1993.

[44] G. Kitagawa, "Monte Carlo filter and smoother for non-gaussian nonlinear state space models," *Journal of Computational and Graphical Statistics*, vol. 5, no. 1, pp. 1–25, 1996.

[45] R. Casarin, "Bayesian monte carlo filtering for stochastic volatility models," Universitè Paris-Dauphine, Tech. Rep., 2004.

[46] Z. Weng, “Particle++,” http://www.ece.sunysb.edu/ zyweng/particle.html, 2012.

[47] D. B. Thomas and W. Luk, “High quality uniform random number generation using LUT optimised state-transition matrices,” *Journal of VLSI Signal Processing Systems*, vol. 47, no. 1, pp. 77–92, 2007.

[48] ——, “An FPGA-specific algorithm for direct generation of multi-variate gaussian random numbers,” in *Proc. Int. Conf. Application-specific Systems Architectures and Processors*, 2010, pp. 208–215.

[49] M. Kurek, T. Becker, T. C. P. Chau, and W. Luk, “Automating optimization of reconfigurable designs,” in *Proceedings of International Symposium Field-Programmable Custom Computing Machines*, 2014.