# SMCGen: Generating Reconfigurable Design for Sequential Monte Carlo Applications

Thomas C.P. Chau*, Maciej Kurek*,
James S. Targett*, Jake Humphrey†, George Skouroupathis*, Alison Eele‡, Jan Maciejowski‡,
Benjamin Cope¶, Kathryn Cobden¶, Philip Leong§, Peter Y.K. Cheung†, Wayne Luk*
*Department of Computing, †Department of Electrical and Electronic Engineering, Imperial College London, UK
‡Department of Engineering, University of Cambridge, UK
§School of Electrical and Information Engineering, University of Sydney, Australia
¶Altera Europe Limited
{c.chau10,mk306,james.targett10,jake.humphrey11,georgios.skouroupathis11,p.cheung,w.luk}@imperial.ac.uk
{aje46,jmm1}@cam.ac.uk philip.leong@sydney.edu.au {bcope,kcobden}@altera.com

*Abstract*—**The Sequential Monte Carlo (SMC) method is a simulation-based approach to compute posterior distributions. SMC methods often work well on applications considered intractable by other methods due to high dimensionality, but they are computationally demanding. While SMC has been implemented efficiently on FPGAs, design productivity remains a challenge. This paper introduces a design flow for generating efficient implementation of reconfigurable SMC designs. Through templating the SMC structure, the design flow enables efficient mapping of SMC applications to multiple FPGAs. The proposed design flow consists of a parametrisable SMC computation engine, and an open-source software template which enables efficient mapping of a variety of SMC designs to reconfigurable hardware. Design parameters that are critical to the performance and to the solution quality are tuned using a machine learning algorithm based on surrogate modelling. Experimental results for three case studies show that design performance is substantially improved after parameter optimisation. The proposed design flow demonstrates its capability of producing reconfigurable implementations for a range of SMC applications that have significant improvement in speed and in energy efficiency over optimised CPU and GPU implementations.**

***Keywords*-FPGA; Sequential Monte Carlo; Machine Learning**

## I. 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 [1] and aircraft [2]. 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 [3] 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 [4, 5], robot localisation [6], object tracking [7] and signal processing [8].

While SMC has been implemented efficiently on FPGAs [4, 6, 7, 8], 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 paper, 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 of this paper are as follows:

- A design flow to reduce the development effort of SMC applications on reconfigurable systems (Section III). Through templating the SMC structure, users can design efficient, multiple-FPGA SMC applications for arbitrary problems, and the software template is open-source.[1]

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

- A machine learning approach that explores the SMC design space automatically and tunes design parameters to improve performance and accuracy (Section IV). 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 [4, 6], we have achieved better quality of solutions and faster designs.
- The benefit 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 V) are demonstrated.

## II. Background and Related Work

### A. SMC Methods

SMC methods estimate the unobserved states of interest based on observations in controlling various agents [9]. 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 algorithm [10] 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 1 and involve five computation stages:

**Algorithm 1** SMC methods

```
for each time step t do
    $idx1 \leftarrow 0$
    Initialisation
    while $idx1 \leq itl\_outer$ do
        $idx2 \leftarrow 0$
        $itl\_inner \leftarrow 3 + 5\exp(\frac{5*idx1}{itl\_outer})$
        for each particle p do
            while $idx2 \leq itl\_inner$ do
                Sampling
                Importance weighting
                $idx2 \leftarrow idx2 + 1$
            end while
        end for
        $idx1 \leftarrow idx1 + 1$
        if $idx1 \leq itl\_inner$ then
            Resampling
        end if
    end while
    Update
end for
```

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 $\{s_{t-1}^{(i)}\}_{i=1}^{N_P}$.

Table I
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 | |

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 [11]. 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 I summarises the parameters of the SMC methods described in Section II-A.

### B. SMC Applications

*1) Stochastic Volatility:* These models are used extensively in mathematical finance [12, 13], 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 1 is employed, where $y_t$ is the observable time varying volatility and $s_t$ represents the stochastic log-volatility process. $\beta$ and $\phi$ are empirical constants.

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

The sampling function in Equation 2 is implied by Equation 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.

$$s_t^i \sim \mathcal{N}(\phi s_{t-1}^i, 1) \tag{2}$$

*2) Robot Localisation:* SMC methods are applied to mobile robot localisation [1], 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 3 and 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 $\delta$ and rotation $\gamma$. Importance weighting is used to calculate the likelihood of a location based on the observation, i.e. the sensor values.

$$(s_t^i) = \begin{pmatrix} x_t^i \\ y_t^i \\ h_t^i \end{pmatrix} = \begin{pmatrix} x_{t-1}^i + \delta_t^{'i} \cos(h_{t-1}^i) \\ y_{t-1}^i + \delta_t^{'i} \sin(h_{t-1}^i) \\ h_{t-1}^i + \gamma_t^{'i} \end{pmatrix} \tag{3}$$

$$(r_t^i) = \begin{pmatrix} \delta_t^{'i} \\ \gamma_t^{'i} \end{pmatrix} = \begin{pmatrix} \mathcal{N}(\delta_t, \sigma_a^2) \\ \mathcal{N}(\gamma_t, \sigma_b^2) \end{pmatrix} \tag{4}$$

*3) 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 [2]. 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 $\{s_t^{(i),0...H-1}\}_{i=1}^{N_P}$ being picked within a permitted range and applied to the current reference status $r_{t-1}$ to compute the future set of reference statuses $\{r_t^{'(i),0...H-1}\}_{i=1}^{N_P}$. 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 $\{s_t^{(i),0...H-1}\}_{i=1}^{N_P}$. Then the selected control is committed to form reference $r_t$.

Equation 5 illustrates a control tuple that consists of roll angle $\phi$; pitch angle $\tau$; and thrust $T$. Equation 6 shows a reference that consists of the current position in three dimensional space $(x, y, a)$, heading angle $\chi$, air speed $V$ and mass $M$. For more details of the model, see [5].

$$(s_t^i) = \begin{pmatrix} \phi_t^{'i} \\ \tau_t^{'i} \\ T_t^{'i} \end{pmatrix} = \begin{pmatrix} \mathcal{N}(\phi_t, \sigma_a^2) \\ \mathcal{N}(\tau_t, \sigma_b^2) \\ \mathcal{N}(T_t, \sigma_c^2) \end{pmatrix} \tag{5}$$

$$\left(r_t^{'i}\right) = \begin{pmatrix} x_t^i \\ y_t^i \\ a_t^i \\ \chi_t^i \\ V_t^i \\ M_t^i \end{pmatrix} = \begin{pmatrix} x_{t-1} + V_{t-1} \cos(\chi_{t-1}) \cos(\tau_t^{'i}) \\ y_{t-1} + V_{t-1} \sin(\chi_{t-1}) \cos(\tau_t^{'i}) \\ a_{t-1} + V_{t-1} \sin(\tau_t^{'i}) \\ \chi_{t-1} + L \sin(\phi_t^{'i})/(M_{t-1} V_{t-1}) \\ V_{t-1} + (\frac{T_t^{'i} - D}{M_{t-1}} - g \sin(\tau_t^{'i})) \\ M_{t-1} - \eta T_t^{'i} \end{pmatrix} \tag{6}$$

## III. 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 IV. (2) A parametrisable SMC computation engine which is made up of customisable building blocks and generic control structure that maximises design reuse.

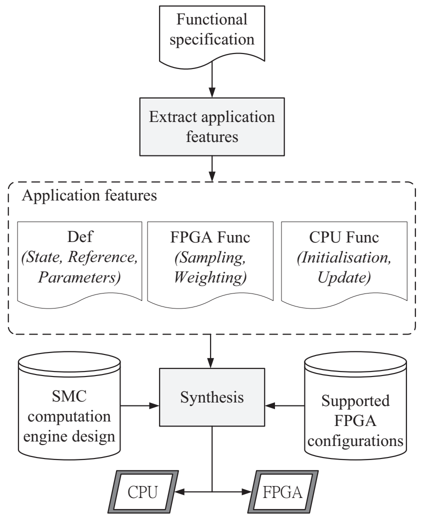


Figure 1. Design flow for SMC applications. Users only customise the application-specific descriptions inside the dotted box.

Figure 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 III-A). The design flow automatically weaves these descriptions with the computation engine (Section III-B) 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.

### A. Specifying Application Features

Users create a new SMC design by customising the application-specific Java descriptions inside the dotted box of Figure 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 I) 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 1) 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.

```java
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;
}
```

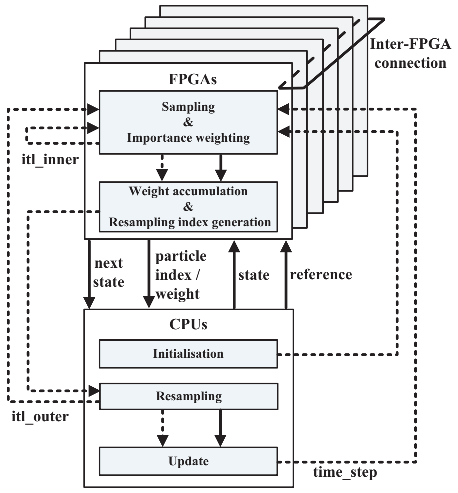
Code 1: Structs and parameters for the robot localisation example.

```java
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;
  }
}
```

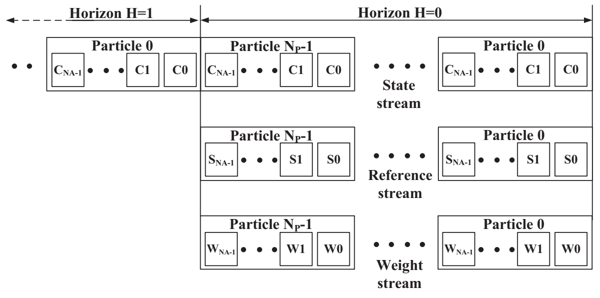
Code 2: FPGA functions (Sampling and importance weighting) for the robot localisation example.

## B. Computation Engine Design

To allow customisation of the computation engine, the engine and data structure are designed as shown in Figure 2(a) and 2(b) respectively. The computation engine



(a)



(b)

Figure 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 three data streams.

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

From the control paths (dotted lines) of Figure 2(a), we see that there are three loops matching Algorithm 1: (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.

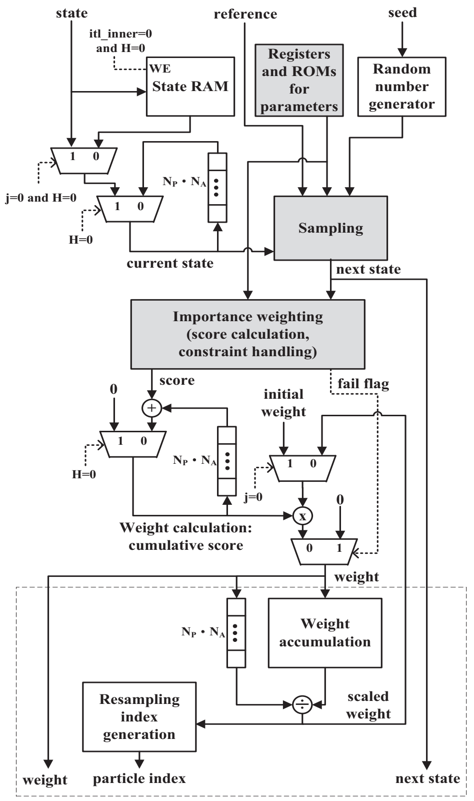


Figure 3. FPGA kernel design. The blocks that require users' customisation are darkened. The dotted box covers the blocks that are optional on FPGAs.

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 2(b) is derived. Each particle encapsulates three 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.

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 is fully pipelined and outputs one result per clock cycle.

Figure 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 [14, 15] 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 three 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 two 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.

## C. 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 IV for parameter optimisation.

The processing time of a time step is shown in Equation 7. It has four components which are iterated $itl\_outer$ times.

$$T_{step} = itl\_outer \cdot \left(T_{s\&i} + T_{resample} + T_{cpu} + T_{transfer}\right) \quad (7)$$

$T_{s\&i}$ 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 III-B, 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.

$$T_{s\&i} = \frac{itl\_inner \cdot N_P \cdot N_A \cdot H}{N_C \cdot N_{Board} \cdot freq} \cdot \min\left(1, \frac{bandwidth}{sizeof(state) \cdot freq}\right) \quad (8)$$

$T_{resample}$ 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.

$$T_{resample} = \frac{N_P \cdot PW + N_P \cdot N_A + 3 \cdot PL \cdot N_P}{freq} \quad (9)$$

$T_{cpu}$ 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. $\alpha_1$ is the scaling factor of the CPU speed.

$$T_{cpu} = \alpha_1 \cdot H \cdot N_P \cdot N_A \tag{10}$$

$T_{transfer}$ 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.

$$T_{transfer} = \frac{N_P \cdot N_A \cdot (H \cdot sizeof(state))}{bandwidth} \tag{11}$$

## IV. Optimising SMC Computation Engine

The design parameters in Table I 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?**

### A. Effect of the Design Parameters

Referring to Table I, the SMC computation engine has up to six 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 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 III-C estimates the run-time of the FPGA implementation.

When $N_P$ and *itl_outer* are explored together as shown in Figure 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, 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 three dimensions as shown in Equation 12.

$$\begin{aligned} &\text{minimise } RMSE = f(N_P, itl\_outer, S) \\ &\text{subject to } RMSE \leq 1 \text{ m}, T_{step} \leq 5\text{s}, \end{aligned} \tag{12}$$

### B. Parameter Optimisation

Now we come to **question 3**, the parameter optimisation problem, which is difficult as construction of an analytical

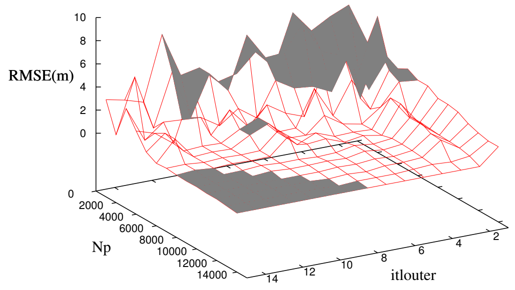


Figure 4. Parameter space of robot localisation system ($N_A$=8192, $S$=1). The dark region on the top-right indicates designs which fail localisation accuracy constraints, while those on the bottom-left indicates designs which fail real-time requirements.

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. 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 [16]. The approach allows the performance impact of different parameters to be 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. Firstly, a number of randomly sampled designs is evaluated (Figure 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(b)). Thirdly, the surrogate model output is used to calculate the expected improvement (Figure 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(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.

## V. Evaluation

### A. Design Productivity

We first analyse how the proposed design flow can reduce design effort. In Table II, 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.

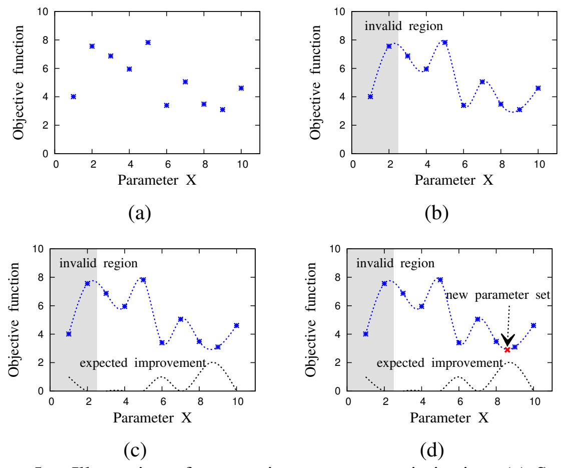


Figure 5. Illustration of automatic parameter optimisation: (a) Sampling parameter sets; (b) Building surrogate model; (c) Calculating expected improvement; (d) Moving to the point offering the highest improvement.

Table II
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 |

### B. Application 1: Stochastic Volatility

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 two dimensions, $N_P$ and $S$ (Table I). 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 III 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.

### C. 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%).

Table III
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.

The design space has three 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 IV compares the performance of our reconfigurable system with CPU, GPU and a previous system in [6] 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 IV
PERFORMANCE COMPARISON OF ROBOT LOCALISATION.

| | CPU opt. [a] | This work opt. [b] | Ref. sys. [6] 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 III.
[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.

### D. 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 six 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 two Intel Xeon E5-2640 CPUs clocked at 2.53GHz. Both parts are connected via InfiniBand.

This application has four 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 V. Hence, the parameter optimisation time is reduced from days to hours.

Table VI 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

Table V
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 VI
PERFORMANCE COMPARISON OF AIR TRAFFIC MANAGEMENT.

| | | CPU opt. [a] | GPU opt. [b] | This work opt. [c] | Ref. FPGA [4] 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 V;
Parameters without optimisation: $itl\_outer$=100, $N_P$=1024, $S$=0.05, $H$=6.

the fact that the sampling and importance weighting process is evenly distributed to every GPU and computed independently, while the resampling 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 [4]. That implementation is only large enough to support four 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.

## VI. CONCLUSION

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 resource requirements with device-specific parameters, such as the level of parallelism and clock speed, into the machine learning approach [16]. We are currently investigating run-time optimisation of parameters based on our initial work [6]. 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.

## ACKNOWLEDGMENTS

This work is supported in part by the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, by UK EPSRC, by the Australian Research Council under Linkage Project LP110200413, by Maxeler University Programme, by HiPEAC NoE, by Altera, by Xilinx, and by the Croucher Foundation.

## REFERENCES

[1] M. Montemerlo, S. Thrun, and W. Whittaker, “Conditional particle filters for simultaneous mobile robot localization and people-tracking,” in *Proc. Int. Conf. Robotics and Automation*, 2002, pp. 695–701.
[2] 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.
[3] D. Creal, “A survey of sequential Monte Carlo methods for economics and finance,” *Econometric Reviews*, vol. 31, no. 3, pp. 245–296, 2012.
[4] 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.
[5] 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 *Proc. Int. Conf. Decision and Control*, 2013.
[6] 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 *Proc. Int. Symp. Applied Reconfigurable Computing*, 2013, pp. 1–12.
[7] M. Happe, E. Lübers, and M. Platzner, “A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking,” *J. Real-Time Image Processing*, vol. 8, no. 1, pp. 95–110, 2013.
[8] G. Hendeby, J. Hol, R. Karlsson, and F. Gustafsson, “A graphics processing unit implementation of the particle filter,” in *Proc. European Signal Processing Conf.*, 2007, pp. 1639–1643.
[9] A. Doucet, N. d. Freitas, and N. Gordon, *Sequential Monte Carlo methods in practice*. Springer, 2001.
[10] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, “Novel approach to nonlinear/non-Gaussian Bayesian state estimation,” *Proc. Radar and Signal Processing*, vol. 140, no. 2, pp. 107–113, 1993.
[11] G. Kitagawa, “Monte Carlo filter and smoother for non-gaussian nonlinear state space models,” *J. Computational and Graphical Statistics*, vol. 5, no. 1, pp. 1–25, 1996.
[12] R. Casarin, “Bayesian Monte Carlo filtering for stochastic volatility models,” Universitè Paris-Dauphine, Tech. Rep., 2004.
[13] Z. Weng, “Particle++,” 2012. [Online]. Available: http://www.ece.sunysb.edu/~zyweng/particle.html
[14] D. B. Thomas and W. Luk, “High quality uniform random number generation using LUT optimised state-transition matrices,” *J. VLSI Signal Process. Syst.*, vol. 47, no. 1, pp. 77–92, 2007.
[15] ——, “An FPGA-specific algorithm for direct generation of multivariate gaussian random numbers,” in *Proc. Int. Conf. Application-specific Systems Architectures and Processors*, 2010, pp. 208–215.
[16] M. Kurek, T. Becker, T. C. P. Chau, and W. Luk, “Automating optimization of reconfigurable designs,” in *Proc. Int. Symp. Field-Programmable Custom Computing Machines*, 2014.