An Adaptive Sequential Monte Carlo Framework with Runtime HW/SW Repartitioning

Markus Happe [1], Enno Lübbers [2], Marco Platzner [1,2]

[1]*International Graduate School,* [2]*Computer Engineering Group*
University of Paderborn, Germany
{markus.happe, enno.luebbers, platzner}@uni-paderborn.de

***Abstract*—The considerable computational complexity of Sequential Monte Carlo (SMC) methods is a major obstacle when implementing them on CPU-based resource constrained embedded systems. Hybrid CPU/FPGA systems, on the other hand, are a more suitable target, as they can efficiently execute both the control-centric sequential as well as the data-parallel parts of an SMC application. Determining the optimal HW/SW partitioning is challenging in general, and since in most cases the optimal partitioning is data-dependent even impossible with a design time approach.**

In this paper, we present a framework for implementing SMC methods on CPU/FPGA based systems such as modern platform FPGAs. Based on a multithreaded programming model, our framework allows for an easy design space exploration with respect to the HW/SW partitioning. Additionally, an SMC application can adaptively switch between several partitionings during run-time to react to changing input data and performance requirements. To show its feasibility and evaluate its performance and area requirements, we demonstrate the framework on two real-world case studies and show that partial reconfiguration can be effectively and transparently used for realizing adaptive HW/SW systems.

I. Introduction

State estimation of non-linear dynamic systems is an important problem with applications in areas as diverse as object tracking, network packet processing, mobile communications and navigation systems. Of the available methods for solving this problem, Sequential Monte Carlo (SMC) Methods – also called particle filters – enjoy widespread popularity and are frequently applied when a system's state is known only by its statistical distribution and can only be partially observed through possibly noisy measurements. For example, Woelk et al. [1] perform outdoor tracking of athletes using a mobile helicopter and an SMC-based tracking system. Jaj et al. [2] employ particle filters in a wearable computing context for estimating the location and activities of a person. Ing et al. [3] present a distributed approach to particle filtering on wireless sensor nodes. Another parallel particle filter approach is used by Granmo [4] for real-time feature classification in data streams using a pipelining technique.

Despite their divergent application areas, all SMC methods follow the same fundamental algorithmic structure and thus share significant portions of common functionality. They track a number of possible state estimates, the *particles*, over time. These particles are continuously compared to measurements to determine the accuracy of the individual state estimates, and weighed accordingly. Usually, the quality of the state estimation can be improved by increasing the number of tracked particles.

The effective use of particle filters in resource-constrained embedded systems is hindered by the considerable computational complexity associated with tracking a large number of state estimates. Because the particles tracked by SMC methods are independent, many of the involved data-centric calculations can be parallelized and are, additionally, amenable to implementation in dedicated hardware. At the same time, the sequential algorithm controlling these computations is implemented more efficiently using a general-purpose CPU. This composite nature of particle filters makes them a perfect fit for hybrid CPU/FPGA architectures, such as modern platform FPGAs. At the same time, such hybrid hardware/software systems increase the complexity of performance estimation and require efficient and flexible mechanisms for design space exploration.

In this paper we present a framework for implementing particle filters on hybrid CPU/FPGA platforms which significantly simplifies the design of particle filters following the sampling importance resampling (SIR) algorithm. Utilizing our multithreaded reconfigurable operating system ReconOS [5], the framework handles the recurring tasks of particle data transfer and thread control, letting the designer focus on the application-specific details of an individual particle filter. Because the operating system transparently supports both software and hardware threads using one unifying programming model, the designer can quickly create different hardware/software partitionings to evaluate their performance and react to changing application and performance requirements.

The novel contributions over previous work [6] are the adaptive reconfiguration capabilities of the framework and significantly extended experimental results including two real-world application case studies. Adaptive reconfiguration allows us to change the HW/SW partitioning of the particle filters at run-time. This is made possible by the transparent hardware multitasking capabilities [7] of the underlying operating system.

The remainder of this paper is structured as follows: Section II discusses related approaches to accelerating particle filters with reconfigurable hardware. Section III introduces the SMC algorithm, while Section IV gives a short overview of the multithreaded HW/SW operating system used by our frame-

978-1-4244-4377-2/09/$25.00 © 2009 IEEE

work. In Section V, a detailed description of the components and composition of our framework is given, before Section VI discusses two SMC-based prototypes from different application domains, each with multiple HW/SW partitionings and corresponding performance figures. A discussion of the adaptive reconfiguration capabilities of the framework is given in Section VII, together with performance data of an adaptive prototype. Section VIII concludes the paper and gives an outlook on our ongoing and future work.

II. Related Work

The need of accelerating particle filter algorithms with reconfigurable hardware has been addressed by several research groups in the previous years. Athalye et al. [8] developed methods and architectures for accelerating the resampling step of the SIR algorithm while at the same time reducing the memory requirements for hardware implementations. Our framework adapts their technique for parallelizing the resampling phase. Sankaranarayanan et al. [9] presented a flexible hardware architecture for SMC methods that uses density sampling techniques from the more general domain of Monte Carlo Markov chain algorithms. This allows them to drop the resampling step which poses scalability and efficiency issues when implemented in hardware. However, the approach does not show significant improvements in quality over traditional SIR filters. As we resolve some of the efficiency issues by implementing the resampling step in software, we chose not to adapt this method. Saha et al. [10] presented a parametrizable framework for the hardware implementation of particle filters, which bears some similarity to our approach in that it provides an interface for the model definition of a particle filter. However, their proposed framework targets a hardware-only implementation of the filter and thus significantly differs from our flexible, multithreaded hardware/software approach in [6].

III. Sequential Monte Carlo Methods

Sequential Monte Carlo Methods estimate a system state x_t at time step t using probability distributions. We are considering that the system state is to be tracked in a dynamic environment and the initial distribution $p(X_0)$ is given, where X_0 is a random variable describing the state at time $t = 0$. The *system model* is a Markov process of first order. Thus, $p(X_t|X_{t-1})$ denotes the probability distribution of the system's current state given the system's previous state. We assume that the system state is hidden and can thus not be observed directly, but can be tracked by measurements y_t, which may be influenced by noise. The relation between measurements and system states is described by the *measurement model*. The distribution $p(Y_t = y_t|X_t)$ describes the probability of the current measurement given the system's current state. In other words, we make a statement about the likelihood of observing a specific measurement, provided we are in a system state modeled by X_t. The probability distribution of X_t is approximated by a fixed number of samples x_t^i, also called particles.

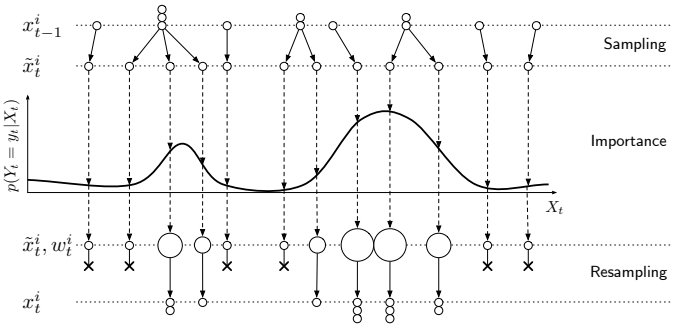

Fig. 1. Sampling importance resampling algorithm [6]

Figure 1 shows one iteration of the Sampling-Importance-Resampling (SIR) algorithm for SMC methods, where particles are depicted as circles.

1) **Sampling:** The new particle state $\tilde{x}_t^i$ is drawn or *sampled* from the distribution $p(X_t|X_{t-1} = x_{t-1}^i)$. Now, the set of particles $\tilde{x}_t^i$ forms a prediction of the distribution of X_t.
2) **Importance:** The measurement model is evaluated for every particle to determine the *likelihood* that the current measurement y_t matches the predicted state $\tilde{x}_t^i$ of the particle. The resulting likelihood is assigned as a weight w_t^i to the particle. In Figure 1, particles with higher weights are drawn as larger circles.
3) **Resampling:** Particles with comparatively high weights are duplicated and particles with low weights are eliminated. The distribution of the resulting particles x_t^i approximates the distribution of the weighted particles before resampling.

For a more thorough discussion of the theoretical foundations of SMC methods we refer to [11].

IV. Multithreaded Operating System

Most contemporary operating systems (OS) for both embedded and general-purpose computing support multithreaded programming as a means to decompose an application into individual threads of execution. In this way, the application designer can express an application's parallelism using a well-defined programming model consisting of threads as well as communication and synchronization primitives. Multithreaded programming allows to effectively share computing resources as well as transparently take advantage of the application's parallelism, given a suitable execution environment.

The operating system ReconOS [12][5] extends the multithreaded programming model to the domain of reconfigurable hardware. Instead of regarding hardware modules as passive coprocessors to the system CPU, they are treated as independent *hardware threads* on an equal footing with software threads running on the system. In particular, ReconOS allows hardware threads to use the same operating system services for communication and synchronization as software threads, providing a transparent programming model across the hardware/software boundary.

This transparency makes design space exploration regarding the hardware/software partitioning of an application a straightforward task. Since all threads use the same programming model primitives such as semaphores, mailboxes or shared memory, they do not need to know whether their communication peers are software threads executed on the CPU, or hardware threads mapped to the reconfigurable fabric. Thus, the hardware/software partitioning of an application can be changed by simply instantiating the appropriate threads.

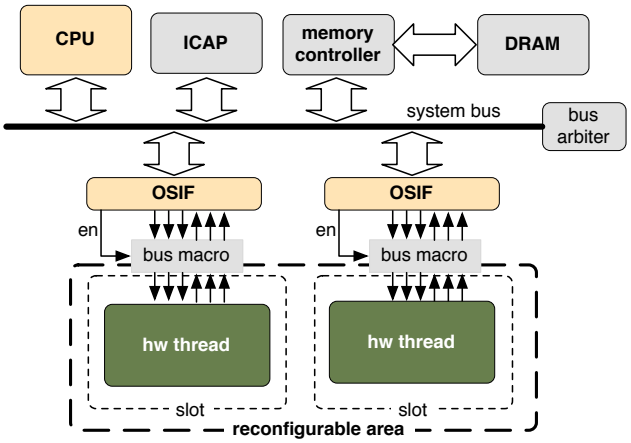

Fig. 2. ReconOS hardware architecture

ReconOS is implemented as an extension to existing operating system kernels, such as eCos or Linux, and targeted at platform FPGAs integrating microprocessors and reconfigurable logic. It takes advantage of the dynamic partial reconfiguration capabilities of Xilinx FPGAs to reconfigure hardware threads during run-time. This allows multiple hardware threads to transparently share the reconfigurable resources. Figure 2 shows the hardware architecture of a typical ReconOS system. The reconfigurable area is divided into multiple slots holding the individual hardware threads. A dedicated hardware OS interface (OSIF) handles the hardware thread's OS requests and forwards them to the operating system kernel running on the CPU. It also manages the low-level synchronization and includes the logic necessary for partial run-time reconfiguration. [7] provides more detail on the hardware multitasking mechanisms provided by ReconOS.

V. Particle Filter Framework

All particle filters using the SIR algorithm rely on the same underlying algorithmic structure. Hence, a substantial part of the functionality, code, and – in the case of hybrid CPU/FPGA systems – hardware circuitry can be re-used supported by a framework-based design approach. Our particle filter framework takes care of common tasks shared by all SIR implementations, such as data transfer and control flow, and lets the designer focus on the application-specific tasks, such as system and measurement modeling.

The characteristic feature of our particle filter framework is the use of multithreaded programming across the hardware/software boundary. The combination of control-centric and data-oriented processing inherent in particle filters closely matches the target platforms and capabilities of our multithreaded operating system (see Section IV), which is used for its implementation.

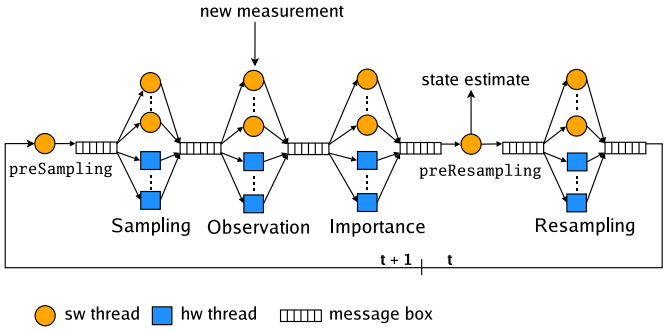

Fig. 3. Structure of a SIR filter implementation

Figure 3 shows the basic structure of an SIR implementation using our framework. The particles cycle through four stages, the three SIR filter stages sampling, importance, and resampling, and an additional observation stage. Each of these stages can have an arbitrary number of software and hardware threads. Based on our experiments and case studies, as outlined in Section VI, we expect that in many applications, determining the importance weight of a given measurement requires particle-specific preprocessing of the measurements and involves significant computational complexity. Since this preprocessing can be done independently for every particle, we have added an additional observation stage, again split into an arbitrary number of hardware and software threads. Our tests show that this technique significantly improves the performance of our case studies.

Communication and synchronization between the threads of different stages is managed using message box primitives of ReconOS. The execution times of threads within the stages may vary due to data-dependencies, memory latencies, CPU load and other factors. The total number of particles is thus split into chunks of user-defined size, which form the atomic entries stored in the message boxes. This enables the framework to balance the load between the threads of a stage and at the same time keeps the communication overhead small.

As shown in Figure 3, a software thread named `preSampling` precedes the sampling stage. This thread retrieves new measurements using the application-specific *receive_new_measurements* function and prepares the sampling stage by reassigning particles to chunks. This is necessary because the resampling stage of the previous iteration replicates some particles and deletes others, leaving the chunks non-uniformly populated with particles. Before particles can be weighed at the importance stage, an observation has to be extracted from the measurement for each particle in the observa-

tion stage. Finally, before resampling, the `preResampling` thread synchronizes the data flow of all particles to normalize their weights. Additionally, a user function *iteration_done* is executed to take care of application-specific tasks once per iteration, such as transferring of estimation results to a display, memory, or I/O, as well as hardware/software repartitioning of the threads in the filter stages. The latter is useful when the stage thread's performance is data-dependent and thus changes during runtime.

The application-specific system and measurement models are captured by the functions *prediction*, *extract_observation* and *likelihood*. For a software implementation, the user simply fleshes out C function templates provided by the framework, which are linked into the corresponding software threads of the sampling, observation and importance stages. When creating hardware versions of these functions, the user writes specialized entities in VHDL, which are then embedded into the hardware threads provided by the framework. Figure 4 illustrates the composition and interactions between functions and modules provided by the user and the framework to assemble a complete application. The complete set of application-specific functions is listed in Table I.

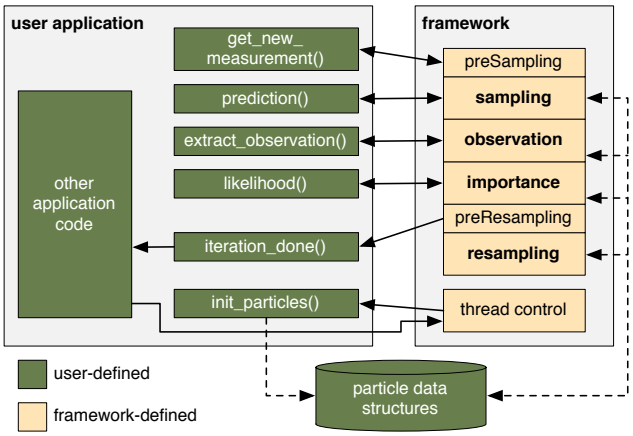

Fig. 4. System composition into application functions and framework functions

TABLE I
USER FUNCTIONS FOR CUSTOMIZING THE PARTICLE FILTER FRAMEWORK

function	description
init_particles	initializes the particles
prediction	moves a particle using the system model
get_new_measurement	retrieves a new measurement for every filter iteration
extract_observation	extracts an observation for a particle from the current measurement
likelihood	weighs a particle using the measurement model
iteration_done	called once per iteration; handles output of best estimate, HW/SW repartitioning or adaptation of reference data

The particle filter framework provides parallelization for the sampling, observation, importance, and resampling stages. The number of hardware and software threads for each stage can be freely chosen, except that there must be at least one thread in each stage. Generally, the number of threads will depend on the availability of computing resources, i.e. CPU utilization factors and logic area. The other threads of the framework are mainly control-dominated or show limited potential for parallelism and are therefore implemented in software. Access to the needed data, the control flow, as well as necessary operating system services for communication and synchronization are completely managed by the framework.

The relevant data structures and initial hardware partitioning is determined and initialized by a software thread, which also sets the initial number of threads for each stage. Table II shows an overview of the framework functions.

TABLE II
FRAMEWORK FUNCTIONS FOR INITIALIZATION AND EXECUTION CONTROL

function	description
create_particle_filter	creates and initializes the particle filter structure
init_reference_data	initializes reference data for *likelihood* function
set_sample_hw/sw	sets number of HW/SW threads for sampling
set_observe_hw/sw	sets number of HW/SW threads for observation
set_importance_hw/sw	sets number of HW/SW threads for importance
set_resample_hw/sw	sets number of HW/SW threads for resampling
start_particle_filter	starts the particle filter

VI. CASE STUDIES

In order to show the applicability of the framework to different problems, we have implemented case studies from two application domains. These two applications also serve to demonstrate how the multithreaded underpinnings of the framework facilitate design space exploration by providing detailed measurements on the performance of different partitionings.

We have implemented the prototypes on a Virtex-II Pro XC2VP30 FPGA and a Virtex-4 XC4VFX100 FPGA. On both systems, an embedded PowerPC 405 CPU, clocked at 300 MHz, runs the software threads and the OS kernel, while the remaining system including buses, peripherals and hardware threads is clocked at 100 MHz. Hardware threads connect to the system bus using OS interfaces provided by ReconOS, which enable transparent utilization of OS services by the hardware. Both applications track $N = 100$ particles divided into chunks of 10.

A. Object tracking

Tracking a moving object in a video sequence is a common task for state estimation methods. Using our framework, we have implemented a prototype system, using a software implementation by Hess [13] as a template and reference. Given an initial video frame, the user selects an object's initial position and its approximate outline in form of a bounding box. The particle filter then estimates the object's position and size in each subsequent frame of the video sequence. An example of the desired tracking behavior can be seen in Figure 5.

(a) Frame 5 (b) Frame 90

(c) Frame 150 (d) Frame 260

Fig. 5. Object tracking in a video sequence (soccer)

The state and measurement models and other application-specific parameters are defined as follows:

- **Particle:** A particle is composed of the horizontal and vertical coordinates x and y of the object's center pixel and the scaling factor s for the bounding box. The height and width of the original bounding box, as well as the first derivatives v_x, v_y, v_s are also stored in the particle to later compute $p(X_t|X_{t-1})$.
- **System model:** The values for x, y, s, v_x, v_y and v_s are predicted as follows, where u_{α_t} models the system noise:

$$\left.\begin{array}{lcl} \alpha_t & = & \alpha_{t-1} + v_{\alpha_{t-1}} + u_{\alpha_t} \\ v_{\alpha_t} & = & \alpha_t - \alpha_{t-1} \\ u_{\alpha_t} & \sim & \mathcal{N}(0, \sigma^2) \end{array}\right\} \alpha \in \{x, y, s\}$$

- **Measurement:** In each iteration, the user function *get_new_measurement* reads a new frame from the video sequence.
- **Observation:** The user function *extract_observation* extracts HSV color histogram data from the measured video frames, distinguishing between colored and uncolored pixels. The histogram data for particle i is stored in an array $H_i(k)$, $k = 0, \ldots, l-1$.
- **Reference data:** The reference data is the histogram $H_R(k)$ of the object selected by the user in the first frame.
- **Likelihood function:** To calculate the *likelihood*, the histogram $H_i(k)$ of the estimated frame region is compared with the reference histogram $H_R(k)$ using the equation:

$$w_t^i = \exp^{-\left(1 - \sum_{0 \le k < l} \sqrt{(H_i(k) H_R(k))}\right)}$$

Thus, the likelihood value assigned to a particle depends exponentially on the similarity between the particle histogram and the histogram of the reference object.

The video data is stored on a PC, converted to HSV color space and transferred via Ethernet to the prototype boards. There, the data is cached in local DRAM, processed, and output to a VGA controller or sent back to the PC via Ethernet. The output includes a bounding box framing the most likely estimate, which is derived from the average of all particles after each importance step.

The HW/SW partitioning of the individual filter stages has a direct impact on the attainable performance. The performance prediction for a specific partitioning or the identification of an optimal partitioning is a rather involved problem, and mostly the performance is even data-dependent. Thus, a designer will implement the required functions in software and/or hardware and then experiment with different HW/SW partitionings, which can be very time-consuming. Here, our framework with its underlying multithreaded programming model assists the designer and allows him to quickly synthesize and evaluate different partitionings.

In this application, placing the sampling or resampling stages in hardware does not yield any performance improvements; for simplicity, these partitionings have not been included in the discussion. Instead, we have focused on the nine hardware/software partitionings shown in Table III.

TABLE III
HW/SW PARTITIONINGS FOR THE OBJECT TRACKING CASE STUDY

partitioning	
sw	All threads run in software.
hw_i	One importance thread executes in hardware.
hw_{ii}	Two importance threads execute in hardware.
hw_o	One observation thread executes in hardware.
hw_{oo}	Two observation threads execute in hardware.
hw_{oi}	One observation thread, one importance thread in hw
hw_{ooi}	Two observation threads, one importance thread in hw
hw_{oii}	One observation threads, two importance threads in hw
hw_{ooii}	Two observation threads, two importance threads in hw

Figure 6 shows the performance of the individual partitionings measured in clock cycles per frame. The measurements were performed on the soccer video sequence displayed in Figure 5.

During the first 100 frames of the sequence, the soccer player fills a large part of the video frame. Consequently, the bounding boxes maintained by the particles are rather large, making the histogram computation expensive. Over the course of the next 150 frames, the soccer player retreats into the background, causing the scaling factor of the particles to diminish. Consequently, the histogram calculations need to be done on smaller bounding boxes which explains the drop in necessary clock cycles per frame. Finally, as the player remains at about the same distance during the remainder of the video sequence, the number of clock cycles per frame levels out.

It can be seen that the performance gain obtained by placing one importance thread in hardware is data-independent across the entire video sequence. This is to be expected, as in this stage two fixed-sized histograms are compared, regardless of the scaling factor. Executing the observation stage in a

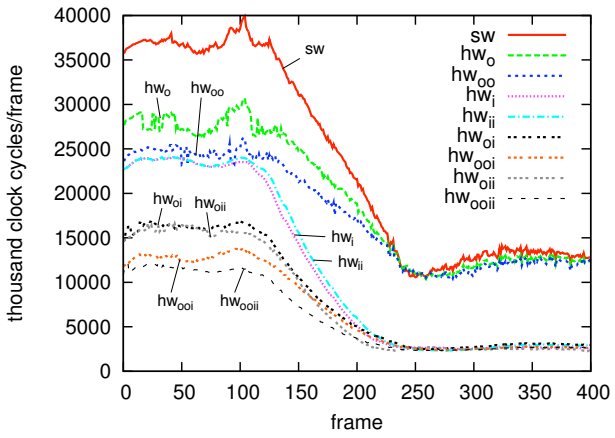

Fig. 6. Object tracking: runtime in $\frac{cycles}{frame}$ of different static partitionings

single hardware thread increases the performance when large amounts of pixel data have to be processed into histograms for a given frame, i.e. when a particle's scaling factor is high. Thus, a performance gain can only be achieved for the first 250 frames before the soccer player retreats into the background.

Because a stage's computations are independent for different particles, we can exploit thread-level parallelism by mapping multiple threads in hardware, provided that the preceding stages can sustain the required data bandwidth. Thus, multiple importance threads only improve performance if the observation stage is also parallelized at the same level, as shown in the measurements of the partitionings hw_{ooi}, hw_{oi} and hw_{ooii}.; otherwise, no additional performance gain is achieved (hw_{ii}, hw_{oii}).

In summary, we see that a single importance thread in hardware does increase the performance of the object tracker in general while the benefit of using one or more observation threads is data-dependent. This fact can be used for efficient utilization of the reconfigurable area, as discussed in Section VII.

B. Beat tracking

As a second case study from a different application domain, we have developed a beat tracker for following the beat frequency in audio files. Beat tracking - like foot-tapping or finger-flipping to the music - can be done by most humans intuitively. However, this task is challenging for a digital system. Our implementation is based on the technique presented by Dixon [14], where a multi-agent system tracks tempo hypotheses over the course of a piece of music.

To obtain an initial distribution of particles, the sound file is split into short intervals (frames), which are analyzed for sound events with high amplitude and low frequency. Starting from the beginning, a tempo estimate (i.e. particle) is assigned to the maximum event exceeding a threshold in every interval until k initial beats are found. A particle is composed of the last beat position and the tempo. The framework stages only execute for a particle, if the estimated beat position is inside the current sound interval. The sampling stage predicts the next beat position according to the current tempo and additive Gaussian noise. When a particle predicts a beat in the current audio frame, the audio data near the prediction is transformed into the frequency domain using a fast Fourier transformation (FFT) in the observation stage. In the importance stage, this data is analyzed. If low frequencies dominate and the audio data exceeds a minimal amplitude A_{min} at the predicted beat position, then the likelihood value is set to the absolute value of the amplitude. The particle with the highest likelihood is selected for beat tracking.

In our case study, we use raw audio data with one audio channel (mono), a sample rate of 44 kHz and a sample depth of two bytes. The audio frames have a fixed size (8kb) and are transfered as TCP/IP-packages via ethernet from a desktop PC to the FPGA. The framework superimposes the beat estimates on the audio frames at the predicted positions and sends them back to the desktop PC for display and playback. To evaluate our beat tracker, we have chosen music pieces with clearly discernible beats in the beginning.

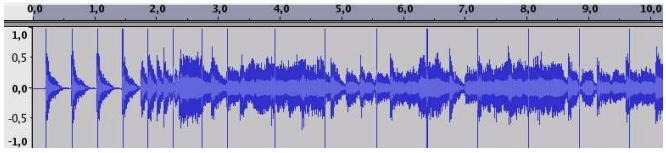

Fig. 7. Beat tracking in a music sequence

Figure 7 shows a processed audio file. Vertical lines represent a beat estimate. It can be seen that the beat tracking starts with an accurate tempo hypothesis of the first four initial beats. After three seconds the estimated beat frequency is halved, but otherwise stays consistent. While more sophisticated methods (e.g. [15]) may improve the quality of beat extraction, our implementation aptly serves to demonstrate the applicability of our SMC framework to different application domains.

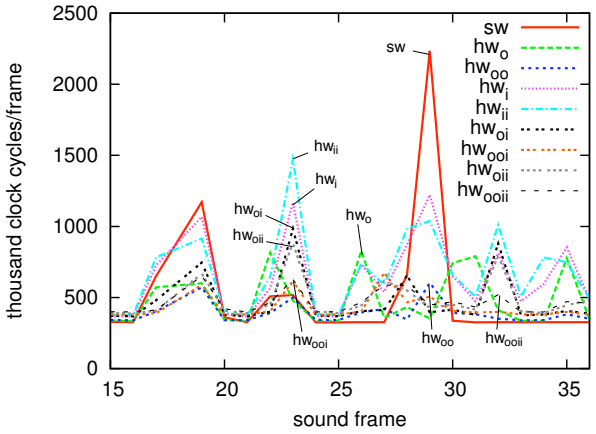

Fig. 8. Beat tracking: runtime in $\frac{cycles}{frame}$ of different partitionings

Figure 8 shows performance measurements obtained from our beat tracking prototype. Depending on the actual predictions, processing occurs only during frames in which the application assumes a beat. For example, in Figure 8 many particles project a beat in frame 19, increasing the computational demand in the corresponding time interval. We have used the same partitionings as defined for our first case study. Again, the observation and the importance stage are the computationally most intensive stages, and thus our primary focus for performance improvements. A dedicated FFT core is provided to accelerate software threads of the observation stage, effectively elevating the software performance of a single thread to the level of a hardware observation thread. In other words, adding additional hardware observation threads only improves performance by exploiting thread-level parallelism. As in the first case study, providing more than one importance thread in hardware has no positive effect.

C. Resource requirements

TABLE IV
RESOURCE REQUIREMENTS IN SLICES (XC4VFX100)

partitioning	object tracking	beat tracking	OS overhead
sw	5372 (12.7%)	7798 (18.5%)	0 (0%)
hw_i	8737 (20.7%)	10304 (24.4%)	1277 (3%)
hw_{ii}	12015 (28.5%)	13013 (30.9%)	2554 (6%)
hw_o	11611 (27.5%)	11611 (27.5%)	1277 (3%)
hw_{oo}	17796 (42.2%)	15647 (37.1%)	2554 (6%)
hw_{oi}	14901 (35.3%)	14317 (33.9%)	2554 (6%)
hw_{ooi}	21298 (50.5%)	18170 (43.1%)	3831 (9%)
hw_{oii}	18384 (43.6%)	16861 (40.0%)	3831 (9%)
hw_{ooii}	24683 (58.5%)	20726 (49.1%)	5108 (12%)

The resource requirements of the individual partitionings of the object tracking prototype and the beat tracking prototype on a Virtex-4 XC4VFX100 FPGA are given in slices in Table IV. The numbers include the OS area overheads consisting of the OSIF area requirements, which are also explicitly listed in the rightmost column. For all hw designs, the area occupied by the CPU, bus infrastructure and other peripherals is roughly equivalent to the area requirements of the sw partitioning. The increased size of the non-thread area is due to the additional FFT core used for SW thread acceleration.

VII. ADAPTIVE RECONFIGURATION

Depending on the system and measurement models of a given application, individual stages of the SIR algorithm may benefit from a hardware implementation to a varying degree. Our framework simplifies the exploration of different HW/SW partitionings, the performance of which are often not easily predictable, through the transparent multithreaded approach explained in Sections IV and V. However, in many applications, such as the case study presented in Section VI-A, the performance of individual stage threads is not only difficult to predict, but data-dependent. Thus, a partitioning that is optimal at one point during the video sequence may at a different point be inferior to another partitioning. In this case, changing the HW/SW partitioning during runtime would increase the overall performance of the application. Additionally, if the application has to satisfy certain performance constraints, such as a predefined iteration frequency of the SIR algorithm, we can change the HW/SW partitioning during runtime to fulfill these constraints while optimizing the application's area usage.

ReconOS transparently supports hardware multitasking through the dynamic reconfiguration of hardware threads. It offers two multitasking models: *non-preemptive* multitasking, where hardware threads run uninterrupted from creation to completion, and *cooperative* multitasking, which allows the preemption of hardware threads at specified preemption points, usually during blocking operating system calls.

In ReconOS, a high-priority software thread, the *hardware scheduler* maintains state data (e.g. *inactive*, *running*, *blocking*, *terminated*) about all hardware threads in the system and controls the reconfiguration process. On a hardware scheduling event, such as thread creation, termination or, in the case of cooperative multitasking, when a thread has reached a preemption point, the scheduler determines the appropriate set of hardware threads and reconfigures them into the slots of the reconfigurable fabric. More detail about the hardware multitasking techniques in ReconOS can be found in [7].

We have evaluated both multitasking techniques for the hardware threads of our framework. Compared to non-preemptive multitasking, the cooperative approach has the potential of better area utilization, as inactive hardware threads can be temporarily replaced by other runnable hardware threads. At the same time, this increases the number of reconfigurations, causing the total reconfiguration overhead to rise. As a result, the two techniques are applicable to different scenarios. Applications which are able to tolerate reconfiguration delays or require only infrequent repartitioning can improve their area utilization by cooperatively sharing the reconfigurable resources. Other applications can still benefit from run-time reconfiguration by utilizing the non-preemptive approach. Reducing reconfiguration overheads is a major focus of our ongoing efforts to improve the reconfiguration infrastructure of our prototypes, and will increase the number of applicable scenarios for cooperative hardware multitasking.

To evaluate adaptive reconfiguration within our framework, we have enabled it for the object tracking case study presented in Section VI-A using the non-preemptive multitasking model. Here, the HW/SW partitioning is changed by terminating unneeded hardware threads or creating new hardware threads, using the standard multithreading API. The dynamic reconfiguration process itself is transparently managed by the hardware scheduler. By estimating the performance of the observation stage based on the scaling factor of the particles, the application can decide whether the observation thread should be executed in hardware or software to satisfy a previously set performance requirement, while at the same time using as little of the reconfigurable area as possible, thus yielding more processing resources to other executing threads or applications running at a lower hardware scheduling priority.

The performance of an *adaptive* partitioning of the object tracking application optimizing area usage can be seen in

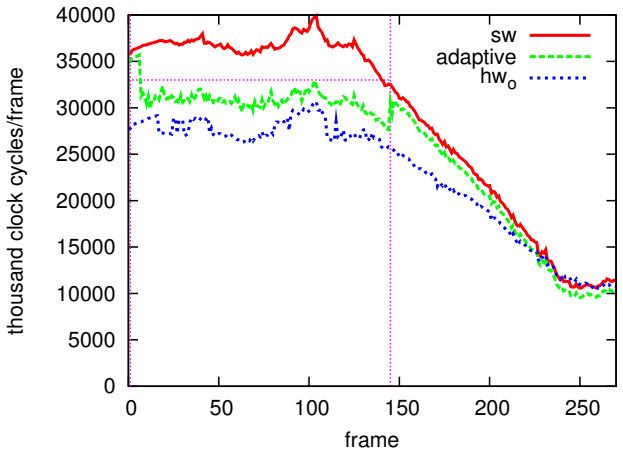

Fig. 9. Object tracking: runtime in $\frac{cycles}{frame}$ of adaptive partitioning, compared to static sw and hw_o partitionings. Reconfiguration occurs at frames 0 and 144, the horizontal line represents the performance requirement.

Figure 9. At the beginning of the video, the comparably large scaling factor prompts the application to instantiate a hardware observation thread, similar to the hw_o partitioning. This causes the frame processing time to drop below the performance requirement of 33×10^6 cycles per frame. Because the additional bus macro logic required for partial reconfiguration adds additional latency to OS calls and memory accesses, the adaptive partitioning does not exactly reach the performance of the static hw_o partitioning. As soon as the soccer player retreats into the background, the scaling factor recedes below a threshold of 0.9. This allows the framework to switch back to a software-only partitioning, comparable to the sw partitioning, which performs good enough for the performance requirement but frees one slot in the reconfigurable logic to be used by other threads.

VIII. Conclusion

In this paper we have presented an adaptive framework for implementing Sequential Monte Carlo methods on hybrid CPU/FPGA platforms. We have demonstrated how using a multithreaded hardware/software programming model can simplify the creation of multiple prototypes to explore the design space of possible hardware/software partitionings. We have presented two case studies from different application domains based on this framework and evaluated the performance of different partitionings. By extending the multithreaded paradigm to reconfigurable hardware, we have shown how transparent hardware multitasking using dynamic partial reconfiguration can be used for exploiting data-dependencies for performance and area optimization of our proposed framework.

Our ongoing and future work is focused on enabling a greater degree of run-time reconfigurability for multithreaded hardware. This entails infrastructural improvements in our hardware architecture, as well as further research into effective scheduling of hardware threads in ReconOS. As discussed in Section VII, the considerable reconfiguration overhead has a direct impact on the applicability and feasibility of the proposed hardware multitasking techniques. We expect these refinements to directly benefit applications based on our framework.

Acknowledgment

This work was supported by the German Research Foundation under project number PL471/2-1 and by the International Graduate School of Dynamic Intelligent Systems.

References

[1] F. Woelk, I. Schiller, and R. Koch, "An Airborne Bayesian Color Tracking System," in *IEEE Intelligent Vehicles Symposium*, June 2005.

[2] A. Jaj, A. Subramanya, D. Fox, and J. Bilmes, "Rao-Blackwellized Particle Filters for Recognizing Activities and Spatial Context from Wearable Sensors," in *9th International Symposium on Experimental Robotics*, 2006.

[3] G. Ing and M. J. Coates, "Parallel Particle Filters for Tracking in Wireless Sensor Networks," in *6th IEEE Workshop on Signal Proc. Advances in Wireless Communications*, 2005.

[4] O.-C. Granmo, "Pipelined Execution of a Parallel Particle Filter for Real-time Feature Selection and Classification in Data Streams," *WSEAS Transactions on Information Science and Applications*, 2004.

[5] E. Lübbers and M. Platzner, "ReconOS: An RTOS supporting Hard- and Software Threads," *IEEE Int. Conf. on Field Programmable Logic and Applications*, 2007.

[6] M. Happe, E. Lübbers, and M. Platzner, "A Multithreaded Framework for Sequential Monte Carlo Methods on CPU/FPGA Platforms," *International Workshop on Applied Reconfigurable Computing*, 2009.

[7] E. Lübbers and M. Platzner, "Cooperative Multithreading in Dynamically Reconfigurable Systems," *19th International Conference on Field Programmable Logic and Applications*, 2009.

[8] A. Athalye, M. Bolić, S. Hong, and P. M. Djuric, "Generic Hardware Architectures for Sampling and Resampling in Particle Filters," *EURASIP Journal on Applied Signal Processing*, 2005.

[9] A. C. Sankaranarayanan, R. Chellappa, and A. Srivastava, "Algorithmic and Architectural Design Methodology for Particle Filters in Hardware," *Int. Conf. on Computer Design*, 2005.

[10] S. Saha, N. K. Bambha, and S. S. Bhattacharyya, "A Parameterized Design Framework for Hardware Implementation of Particle Filters," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, 2008.

[11] A. Doucet, N. de Freitas, and N. Gordon, *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[12] E. Lübbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM TECS Special Issue (CAPA)*, 2009, to appear.

[13] R. Hess, "Particle Filter Object Tracking," 2006, http://web.engr.oregonstate.edu/~hess.

[14] S. Dixon, "Automatic extraction of tempo and beat from expressive performances," *Journal of New Music Research*, vol. 30, pp. 39–58, 2001.

[15] W. A. Sethares, R. D. Morris, and J. C. Sethares, "Beat tracking of musical performances using low-level audio features," *IEEE Transactions on Speech and Audio Processing*, 2005.