J Real-Time Image Proc (2013) 8:95–110
DOI 10.1007/s11554-011-0212-y

SPECIAL ISSUE

A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking

Markus Happe · Enno Lübbers · Marco Platzner

Received: 30 November 2010 / Accepted: 29 June 2011 / Published online: 15 July 2011

Abstract Sequential Monte Carlo (SMC) represents a principal statistical method for tracking objects in video sequences by on-line estimation of the state of a non-linear dynamic system. The performance of individual stages of the SMC algorithm is usually data-dependent, making the prediction of the performance of a real-time capable system difficult and often leading to grossly overestimated and inefficient system designs. Also, the considerable computational complexity is a major obstacle when implementing SMC methods on purely CPU-based resource constrained embedded systems. In contrast, heterogeneous multi-cores present a more suitable implementation platform. We use hybrid CPU/FPGA systems, as they can efficiently execute both the control-centric sequential as well as the data-parallel parts of an SMC application. However, even with hybrid CPU/FPGA platforms, determining the optimal HW/SW partitioning is challenging in general, and even impossible with a design time approach. Thus, we need self-adaptive architectures and system software layers that are able to react autonomously to varying workloads and changing input data while preserving real-time constraints and area efficiency. In this article, we present a video tracking application modeled on top of 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, the application can adaptively switch between several partitionings during run-time to react to changing input data and performance requirements. Our system utilizes two variants of a add/remove self-adaptation technique for task partitioning inside this framework that achieve soft real-time behavior while trying to minimize the number of active cores. To evaluate its performance and area requirements, we demonstrate the application and the framework on a real-life video tracking case study and show that partial reconfiguration can be effectively and transparently used for realizing adaptive real-time HW/SW systems.

Keywords Real-time video object tracking · Heterogeneous multi-core architecture · Self-adaptation · CPU/FPGA systems · HW/SW co-design

M. Happe (✉) · M. Platzner
Computer Engineering Group, Warburger Str. 100,
33098 Paderborn, Germany
e-mail: markus.happe@uni-paderborn.de

M. Platzner
e-mail: platzner@uni-paderborn.de

E. Lübbers
EADS Innovation Works, Technical Capability Center 5,
81663 Munich, Germany
e-mail: Enno.Luebbers@eads.net

1 Introduction

Real-time video object tracking has important applications in areas as diverse as aircraft and missile tracking, pedestrian surveillance, athlete tracking and industrial vision systems. Some of these systems have to deal with the problem of determining the current position of an object with non-linear movement across sequential video frames. Therefore, they need to employ sophisticated methods to continuously extract a reasonable estimate of the object's location during run-time.

A popular class of state estimation algorithms is represented by Sequential Monte Carlo (SMC) methods. These iterative statistical methods, also denoted as particle filters, have already found widespread adoption in various computing contexts. As they are often employed to estimate physical systems, many embedded applications benefit from a particle filter's capability to track non-linear systems, as opposed to Kalman filters [12]. For example, Woelk et al. [27] perform outdoor tracking of athletes using a mobile helicopter and an SMC-based tracking system.

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 particles.

Real-time video object tracking is usually performed in resource-constrained embedded systems. In such systems, the considerable computational complexity associated with tracking a large number of estimates about the object's current position is challenging. Typically, the accuracy of object tracking increases dramatically with the number of particles and it is thus tightly connected to the computational capabilities of the targeted platform. However, being an on-line estimation scheme, the quality of an SMC system's tracking results depends not only on their accuracy, but also on their timeliness. Thus, meeting real-time constraints constitutes a high-design priority of embedded particle filters.

Since particles tracked by SMC methods are independent, many of the involved data-centric calculations can be parallelized and they 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. More specifically, a heterogeneous multi-core system that provides multiple execution environments including both dedicated hardware circuits and general purpose CPUs, which offers a great potential to meet the specific demands of particle filters while achieving high performance. At the same time, such heterogeneous multi-core systems increase the complexity of performance estimation and require efficient and flexible mechanisms for design space exploration.

Since the computational requirements for the individual parts of video object tracking often vary or are input data-dependent, the optimal distribution of tasks between custom hardware cores and software processors at different cost/performance/energy design points can change dramatically between implementations and also during run-time. For instance, in histogram-based video object tracking systems the object size strongly influences the computational complexity. Thus, many real-time video tracking systems track fixed sized objects. When considering self-adaptive heterogeneous multi-core systems, however, we can allow changing object sizes by activating or deactivating cores. Deactivating cores with low utilization helps to reduce the power consumption of the system. Hence, cost-efficient real-time video object tracking systems require mechanisms of adapting their architecture during run-time. This new requirement of *self-adaptability* asks for the development of flexible system architectures capable of dynamically reconfiguring their processing elements and distributing tasks among them, which again makes reconfigurable technology such as FPGAs a prime implementation target for both flexible design space exploration and run-time adaptability.

Thus, in summary, a design environment must fulfill the following requirements to effectively promote the efficient use of SMC-based algorithms in real-time video object tracking systems:

- provide a common SMC architecture for rapid development of particle filters from various application areas,
- combine efficient software and custom hardware execution cores for both sequential and data-parallel implementations of the individual parts of a particle filter,
- allow flexible HW/SW re-partitioning at design time to custom-fit an application to its targeted domain,
- facilitate run-time HW/SW re-partitioning to adapt to changing video data characteristics and meet real-time constraints while preserving implementation efficiency.

The main contribution of this article is a common and flexible design environment for real-time video object tracking applications targeted at hybrid CPU/FPGA platforms. We have developed a self-adaptive, multithreaded framework that directly exploits the dynamic reconfiguration capabilities of modern platform FPGAs and significantly simplifies the design of particle filters following the sampling importance resampling (SIR) algorithm. Utilizing our multithreaded reconfigurable operating system ReconOS, developed by Lübbers and Platzner [16], the framework handles all recurring tasks such as particle data transfer and thread control, letting the designer to focus on the application-specific details of an individual particle filter. Since the operating system transparently supports both software and hardware threads using one unifying programming model, the designer can quickly create different HW/ SW partitionings to evaluate their performance and react to changing application and performance requirements. During run-time, the framework also supports on-line task re-partitioning using partial reconfiguration.

This article summarizes and extends the work presented by Happe et al. [8, 9]. New aspects are (1) the integration of worker CPUs to form heterogeneous multi-core systems, (2) two self-adaptation techniques for real-time HW/SW repartitioning and (3) new experimental results. As a case study, we leverage the video object tracking system presented by Happe et al. [9] and advance it to utilize multiple processors and hardware cores. Our novel self-adaptation techniques successfully re-partition threads onto different cores and even across the HW/SW boundary to keep the performance of a video object tracking application above a user-defined bound or within a user-defined performance budget.

The article is structured as follows: Sect. 2 outlines the related work on real-time video object tracking and self-adaptation in heterogeneous multi-core platforms. Furthermore, it provides a background on SMC methods and introduces the employed multithreaded programming environment that was specifically designed for CPU/FPGA systems. Section 3 describes the components and design approach of our self-adaptive multithreaded framework which can be used to design any kind of particle filter applications. Thus, the framework is not restricted to be applied in video object tracking applications. The novel self-adaptive add/remove strategies are presented in Sect. 4. A case study application for video object tracking is introduced together with its system model, experimental results from prototype implementations with static and self-adaptive heterogeneous multi-core systems in Sect. 5. Finally, Sect. 6 concludes the article and gives an outlook on our ongoing and future work.

2 Background and related work

In this section, we present releated work on real-time video object tracking systems, in Sect. 2.1, and related work on self-adaptation in heterogeneous multi-core platforms, in Sect. 2.2. Section 2.3 introduces ReconOS, a common multithreaded programming environment of CPU/FPGA systems, which was used to implement our heterogeneous multi-core architecture. Finally, we present background and releated work on SMC methods, in Sect. 2.4.

2.1 Real-time video object tracking

Object tracking is among the computationally most intensive image processing tasks [18]. In various application areas, it has to adhere to real-time constraints.For instance, Gilbert et al. [7] designed a real-time video object tracking system that identifies and tracks missiles and aircraft using adaptive and statistical clustering and a project-based classification algorithm for military use. Other scenarios include the observation of pedestrians. Loza et al. [15] compared the combinational approaches to a particle filter for multiple target video object tracking in the scenario of tracking pedestrians in a public area. Their experiments show that particle filters are more accurate while combinational approaches achieve a higher performance, but require well-separated objects.

Kobayashi et al. [13] used particle swarm optimization to track a green ball in a video with the frame size of 640×480 and a target object size between 1 and 30 pixels at 25 frames per second (FPS) using 100 particles. Cho et al. [4] stated that pattern matching requires expensive computation power and it is a bottleneck to meet real-time constraints. Thus, they developed a hardware circuit that tracks an object by computing the color histogram for a 15×15 sample window for each possible position. The architecture was implemented on a Virtex-4 XC4VLX200 FPGA and achieves 81 FPS for frames at VGA resolution. However, the object size is limited and cannot be adapted at run-time. Ali et al. [1] implemented a kernel-based mean shift tracking algorithm on a MicroBlaze processor and connected it to hardware components for frame grabbing and display. The system tracks an object of size 32×32 at 25 FPS using color histograms for videos with the size 360×288, where the processor is clocked at 50 MHz and the histogram is constrained to two color components.

Visentini et al. [26] proposed a cascaded version of an online boosting algorithm that self-adapts the number of levels and, thus, the shape of the cascade at run-time to achieve real-time. As classifiers they used Haar features, local bin patterns and color histograms. They were able to achieve 25 FPS on an AMD Athlon 64 3500+ with 1 GB RAM, but they were restricted to a fixed sized object size. Here, the varying computational complexity results from the tracking confidence of the three cascade ensembles.

Similar to some related work, we use a particle filter in combination with color histograms for object tracking. However, in most real-time systems the target object size is fixed. Our real-time system, instead, uses a reconfigurable heterogeneous multi-core architecture, which can handle changing target object sizes and, thus, changing computational complexity by self-adaptation.

2.2 Self-adaptation in heterogeneous multi-core systems

In recent years, thread-level adaptability of embedded devices in both microprocessor and reconfigurable hardware contexts has been investigated from different angles. For example, Kumar et al. [14] argued that for many applications, core diversity is more important than uniformity to better adapt to the changing demands of applications. The authors designed and simulated a heterogeneous multiprocessor system where the processors show different power/performance characteristics. During the application's execution, the system software selects the most appropriate core that saves the most energy while still achieving a specified performance. For 14 SPEC benchmarks, they achieved an average energy saving of 39% with a performance loss of 3%.

Curtis-Maury et al. [5] recognized that threads targeted at complex multi-core systems interact in a complex and data-dependent manner and proposed to adapt the thread/core distribution based on run-time thread execution profiles. They developed a multithreading library for power/performance adaptation and evaluated their approach on OpenMP-based applications running on a symmetric multicore system without heterogeneous hardware cores.

In the field of dynamic task mapping in FPGA-based systems, Stitt et al. [25] proposed dynamically partitioning applications between software processors and hardware co-processors, where co-processors are generated at run-time using binary decompilation and synthesis. Huang and Vahid [11] showed that the dynamic co-processor management can be reduced to the metrical task system problem and presented a fading cumulative benefit (CBF) heuristic. The heuristic stores the theoretically achievable cumulative benefit for each application that could have been achieved by using a co-processor. The CBF values are fading in order to focus on temporal locality. Sigdel et al. [21] introduced a two-level design space exploration to solve the dynamic task mapping problem. At design time, the first-level DES explores the system under static conditions to find the optimal static task partitioning. At run-time, the second-level DES seeks to optimize the task mapping to adapt changes in the application, architecture or the environment. Similar to both approaches, we map tasks to processors and reconfigurable logic. In contrast to Vahid et al., who generates hardware co-processors from parallelizable code, we map entire threads (including OS calls) into hardware. While most approaches try to maximize the performance, we aim to achieve a desired performance while minimizing the number of active cores.

Considering more heterogeneous systems that also consider DSPs, ASICs, etc, Smit et al. [23] proposed two heuristics which minimize the energy consumption: the adapted minWeight heuristic and the more advanced hierarchical iterative heuristic [24]. Moreover, several heuristics for NoC-based systems were published, i.e., Nollet et al. [17]; Carvalho et al. [3]. Nollet et al. [17] improved the task assignment success rate by dynamic instantiation of softcore processors that allow tasks, which do not have a hardware implementation, to be mapped to the FPGA fabric. In contrast to our work, these approaches either focus on energy consumption or on the specifics of NoC-based architectures.

Finally, Sironi et al. [22] proposed a self-aware FPGA-based system where the applications performance can be constrained into an interval. The application's performance is monitored using the Heartbeat Application, where an application sends a heart beat to a framework at specific points in time. The Heartbeats framework observes if the measured heart rate is inside a user-defined performance budget and controls re-partitioning. In contrast to our work, the work of Sironi et al. does not provide any concrete re-partitioning heuristics or dynamic measurements.

2.3 Multithreaded programming of CPU/FPGA systems

Multithreaded programming is a popular 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. Given a suitable execution environment, multithreaded programming enables the system to effectively share computing resources, as well as transparently take advantage of the application's parallelism. It is supported by most contemporary operating systems and enjoys widespread use in the software domain, both for desktop and high-performance computing and within the embedded systems.

SMC methods following the SIR approach can be easily separated into individual threads representing the stages sampling, importance, and resampling. As the particles are independent of each other, all stages can be parallelized, such that multiple threads per stage process a separate group of particles each. With this decomposition, an SMC application can be represented by a collection of partly interdependent, communicating, and synchronizing threads, which (for the reasons introduced in Sect. 1) execute most efficiently on a hybrid HW/SW execution platform. Thus, a common multithreaded programming model for hardware and software threads offers an ideal basis for designing particle filters on such systems.

The operating system ReconOS [16] 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 in 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 the design space exploration regarding the hardware/software partitioning of an application a straightforward task and facilitates self-adaptation. Since all threads use identical 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 HW threads mapped to the reconfigurable fabric. Thus, the HW/SW partitioning of an application can be changed at design or run-time by simply instantiating the appropriate threads. ReconOS supports dynamic reconfiguration of hardware threads by taking advantage of the partial reconfiguration capabilities of Xilinx FPGAs.

ReconOS builds on top of and extends existing operating system kernels, such as eCos or Linux, and it is targeted at platform FPGAs integrating microprocessors and reconfigurable logic. Figure 1 shows the architecture of a typical ReconOS system. A master CPU executes the operating system kernel and manages all interactions between the individual threads of the application. In addition, it can run regular software threads using the kernel's software API. The reconfigurable area is divided into multiple *slots* of arbitrary size and shape. These slots and any additional microprocessor cores (called *worker* CPUs) are connected to the system through a dedicated hardware OS interface (OSIF). The OSIF also manages the low-level synchronization and includes the logic necessary for partial run-time reconfiguration. Hardware threads running in the reconfigurable slots (*HW Threads*), as well as software threads running on the worker processors (*CPU–HW Threads*), route all their operating system interaction requests through their OSIF, which forwards the requests to a corresponding delegate thread running on the CPU. This *delegate thread* performs all OS interaction on behalf of the HW and CPU–HW threads. Hence, the operating system can transparently handle both hardware and software threads running on a mix of heterogeneous processing elements. This approach provides maximum transparency and portability and allows CPUs with different instruction set architectures and binary formats as well as custom hardware accelerators with varying design, performance, and implementation costs, to be integrated into a single execution environment.

ReconOS presents a flexible foundation to design SMC applications for embedded heterogeneous multi-core systems. Through the common programming model and the offered transparency, the particle filter can be transparently distributed over the heterogeneous processing elements. Furthermore, thanks to the partial reconfiguration capacities of ReconOS, hardware and software threads can be inserted and removed at run-time. Hence, ReconOS provides a proficient basis to design self-adaptive systems that change their HW/SW partitioning at run-time to react to changing workloads.

2.4 Sequential Monte Carlo methods

SMC methods estimate a system state x_t at time stage t using probability distributions. We are considering that

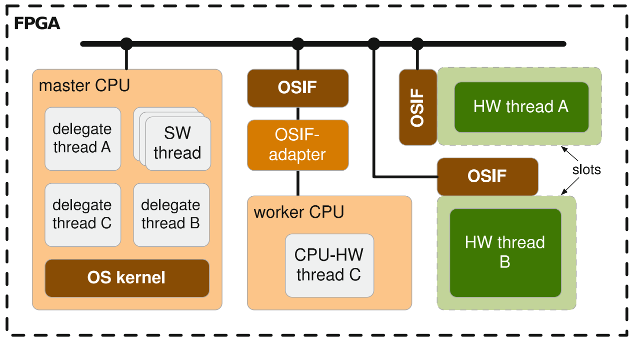

Fig. 1 ReconOS hardware architecture: the operating system and possibly multiple software threads run on a master processor. Threads that run on worker cores are represented as delegate threads on the master processor. In this example of multi-core architecture, one worker processor and two dynamically reconfigurable hardware slots are connected to the master processor using a shared bus

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 thus cannot 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.

Based on the system and measurement models, we are interested in *predicting* the current state based on past measurements, $p(X_t|y_1,\ldots,y_{t-1})$, or in *updating* the current state prediction based on the latest measurement, $p(X_t|y_1,\ldots,y_t)$. SMC methods approximate these distributions by drawing a large number of samples from them. However, as these distributions are typically unknown, we cannot directly draw samples from them. Several techniques have been proposed to circumvent this problem, among them is *importance sampling*. When the estimates for the system state have to be computed on-line while the state changes, as it is the case for tracking applications, the *sequential importance sampling* (SIS) and *sequential importance resampling* (SIR) methods can be applied, which present a recursive approach to approximate the desired distributions. Our framework closely follows the SIR algorithm (Algorithm 1) as it is one of the most widely used SMC methods.

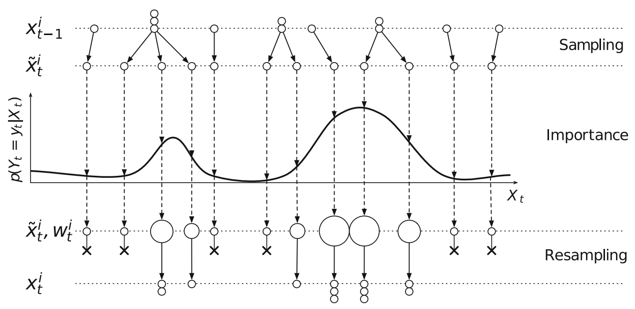

Fig. 2 Sampling–importance–resampling algorithm: the figure illustrates one iteration of the algorithm. Happe et al. [8]

Figure 2 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 $\tilde{x}^i_t$ is drawn from the distribution $p(X_t|X_{t-1} = x^i_{t-1})$. Now, the set of particles $\tilde{x}^i_t$ 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 $\tilde{x}^i_t$ of the particle. The resulting likelihood is assigned as a weight w^i_t to the particle. In Fig. 2, 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^i_t approximates the distribution of the weighted particles before resampling.

As can be derived from Fig. 2, the SIR algorithm replicates all particles that form a likely prediction of the state while eliminating particles that do not match well with the current measurement. This process can be viewed as a *filter* that tracks several state hypotheses at once, filtering out hypotheses that are not supported by the measurements. After each time stage, the most likely match can be extracted from the probability distribution approximated by the particles, e.g., by selecting the particle with maximum weight or computing the average over all particles.

Algorithm 1 SIR algorithm. Arulampalam et al (2002)

```
1: $[\{x^i_t, w^i_t\}^N_{i=1}]$ = SIR $[\{x^i_{t-1}, w^i_{t-1}\}^N_{i=1}, y_t]$
2: for $i = 1, \ldots, N$ do
3:     Draw $x^i_t \sim p(X_t|X_{t-1} = x^i_{t-1})$          ▷ Sampling
4:     $w^i_t = p(Y_t = y_t|X_t = x^i_t)$                    ▷ Importance
5: end for
6: $W = \sum^N_{i=1} w^i_t$                                  ▷ Calculate total weight
7: for $i = 1, \ldots, N$ do
8:     $w^i_t = w^i_t / W$                                   ▷ Normalize weights
9: end for
10: $[\{x^i_t, w^i_t, -\}^N_{i=1}]$ = RESAMPLE $[\{x^i_t, w^i_t\}^N_{i=1}]$   ▷ Resampling
```

Algorithm 1 presents the pseudo code of the SIR algorithm for N particles. In lines 2–5, the particles x^i_t are first drawn according to the prediction $p(X_t|X_{t-1} = x^i_{t-1})$ and then weighed according to the likelihood $p(Y_t = y_t|X_t = x^i_t)$. As the particles are independent from each other, the for-loop can be parallelized. Furthermore, the sampling and importance stages can be pipelined. For the resampling stage (line 10), the particle weights have to be normalized (lines 6–9). In most cases, the current state prediction is either the particle that has the highest weight or the (weighed) average of all particles.

For a more thorough discussion of the theoretical foundations of SMC methods we refer to Doucet et al. [6].

The necessity of accelerating particle filter algorithms with reconfigurable hardware has been addressed by several research groups in previous years. Sankaranarayanan et al. [20] 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. [19] presented a parameterizable 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 static hardware-only implementation of the filter and thus significantly differs from our flexible, multithreaded HW/SW approach. Furthermore, their static approach does not support any on-line self-adaptation techniques.

3 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 Sect. 2.3), which is used for its 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 study, as outlined in Sect. 5, 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 primarily 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 Fig. 3, a software thread named `preSampling` precedes the sampling stage. Using the application-specific *receive_new_measurements* function, this thread retrieves new measurements and prepares the sampling stage by reassigning the 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.

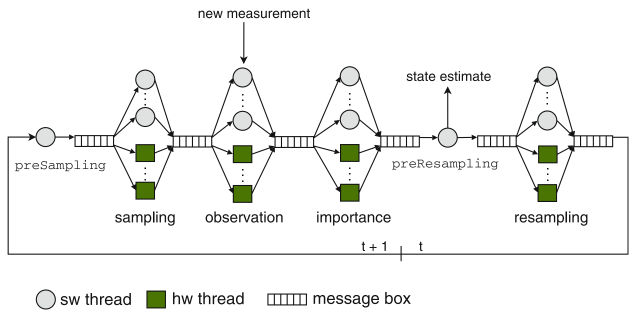

Fig. 3 Structure of a SIR filter implementation: the stages sampling, importance, observation, and resampling can be represented by multiple heterogeneous threads. The particles are partitioned into chunks which are sent as messages through the four stages. Happe et al. [9]

Before particles can be weighed at the importance stage, the observation stage extracts an application-specific feature (the *observation*) from the measurement for each particle. 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 (see Sect. 5.1).

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 they 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 are determined and initialized by a software thread, which also set the initial number of threads for each stage.

To take advantage of the services provided by the framework, an SMC application designer needs to structure the application-specific descriptions regarding the sampling, observation, importance, and resampling stages in a predefined way. For software implementations, the user simply fleshes out C function templates provided by the framework, which are linked into the corresponding software threads of the filter stages. On the other hand, hardware implementations are structured as shown in Fig. 4 which exemplary depicts the hardware description for the sampling stage. A hardware implementation of a filter stage is always split into a framework part and an application-specific part. On the one hand, the framework part captures the parts of a filter stage that are identical for different application areas which include the interaction with the operating system. On the other hand, the application-specific part defines the user logic which depends on the current application and thus has to be provided by the application designer.

The framework part implements a synchronous state machine to interact with the operating system. First, the

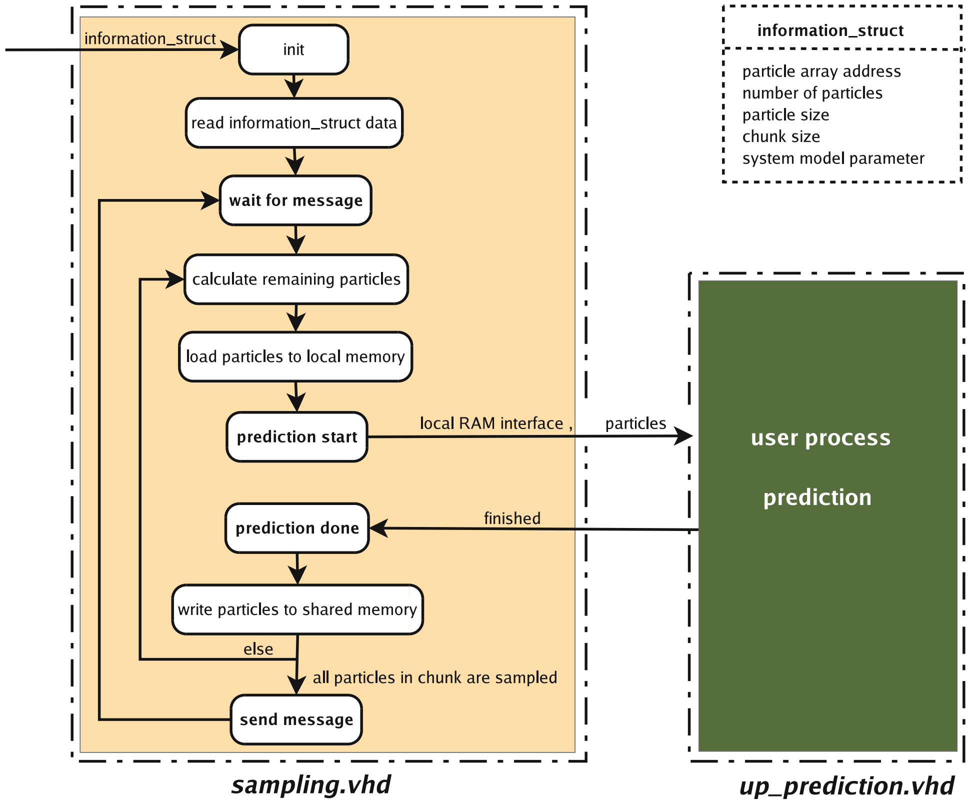

Fig. 4 Hardware thread for the sampling stage: the framework part contains a synchronous state machine that communicates with the operating system, while the application-specific part provides the user logic that samples particles according to the application's system model

hardware thread is initialized with information about the particles and the system model that is stored in `information_struct`. It then waits for a message containing the information which chunk of particles needs to be sampled. The framework stores as many particles as possible into the local RAM of the hardware thread before it informs the user process about the number of available particles. After the application-specific user process has predicted the next particle state, it notifies the framework that the sampling process is completed. The framework then writes the sampled particles into shared memory and either loads the next particles of the chunk or forwards the chunk message to the next stage. The hardware implementations for the other stages are designed in a similar way. The particle filter designer has the maximum flexibility to design the user logic for a filter stage after the framework has read all necessary data from the shared memory.

4 Real-time self-adaptation strategies

In this section, we present two self-adaptive add/remove strategies for multithreaded applications on heterogeneous multi-core systems, such as our video tracking case study. The two adaptive thread partitioning algorithms, $\mathrm{ATP}_{\mathrm{bound}}$ and $\mathrm{ATP}_{\mathrm{budget}}$, aim to satisfy soft real-time constraints while minimizing the required processing resources. While $\mathrm{ATP}_{\mathrm{bound}}$ always tries to keep the system performance above a lower bound, $\mathrm{ATP}_{\mathrm{budget}}$ aims to stay within a performance budget and attain a predefined average performance, thereby trading lower resource usage for slightly higher rates of missing the lower performance bound. Both approaches are well suited to all particle filter applications and, consequently, supported within our framework to enable on-line self-adaptation.

For both algorithms, we assume that an instance of each thread executes on the master processor at all times and that each worker core is assigned at most one thread instance in total. Initially, the application with all its threads executes entirely on the master processor. The master processor also measures the total application performance at user-defined time intervals.

Formally, we denote the set of threads as T, a distinct thread as T_i, the set of cores as C, an individual core as C_j and a specific thread instance executing on a specific core as $T_{i,j}$. Thread instances generally show differing performance on different heterogeneous cores, i.e., threads with data-parallel computations can be more efficiently mapped into reconfigurable logic. Furthermore, general speedup factors of a heterogeneous core cannot be given for every kind of task, whereas data-parallel computations can provide good speedups when mapped to reconfigurable logic, sequential algorithms usually show better performance on processors. Thus, $s_{i,j}$ models the specific speedup of a (worker) core C_j executing the thread instance of T_i compared with the thread instance that executes on the master core. Consequently, the speedup of the thread instance on the master core is 1. We assume that the workload of thread T_i distributes among the cores which execute instances of T_i proportionally to their speedup values $s_{i,j}$. Specifically, if the master core j executes a thread instance $T_{i,j}$ and a worker core j' executes another thread instance $T_{i,j'}$ with the speedup value $s_{i,j'} = 2$, we assume that the master core executes $\frac{1}{3}$ of the entire workload, see Fig. 5.

The main purpose of this assumption is to allow us to predict the application's execution time when adding or removing another thread instance. With $e_{i,\text{master}}$ as the *measured* fraction of T_i's execution time on the master processor under the current partitioning, we can model a *hypothetical* execution time (or workload) $W(T_i)$ for a single-core partitioning which would map T_i only on the master processor (and no worker cores):

$$W(T_i) = e_{i,\text{master}} \times \sum_{\forall j: T_i \in A(C_j)} s_{i,j} = e_{i,\text{master}} \times S_i$$

where, $A(C_j)$ represents the set of thread instances that execute on core C_j, and S_i denotes the estimated cumulated speedup of thread T_i. Note that since starting or stopping, a thread instance changes both $e_{i,\text{master}}$ and S_i, $W(T_i)$ remains constant regardless of the chosen partitioning. An additional thread instance $T_{i,j'}$ on a free core C_j' will lead to savings in the master processor's actual execution time (reward) for that thread of

$$reward_{i,j'} = e_{i,\text{master}} \frac{s_{i,j'}}{S_i + s_{i,j'}}$$

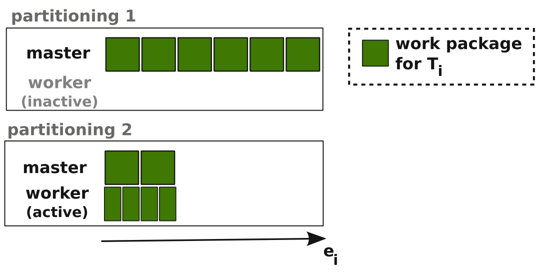

Fig. 5 The first partitioning shows the scenario where all work packages are computed by the master j. In contrast, the second partitioning shows the case where the worker j' with $s_{i,j'} = 2$ computes $\frac{2}{3}$ of the workload of T_1 and reduces the execution time of the master to $\frac{1}{3}$.

Similarly, terminating a thread instance $T_{i,j'}$ on a worker core C_j' will lead to a likewise increase in the master processor's actual execution time (penalty) for that thread of

$$penalty_{i,j'} = e_{i,\text{master}} \frac{s_{i,j'}}{S_i - s_{i,j'}}$$

The resulting system performances are denoted as p_{reward} and $p_{penalty}$, respectively.

4.1 ATP$_{\text{bound}}$ algorithm

Algorithm 2 presents the pseudo code for our bound-based add/remove self-adaptation technique ATP$_{\text{bound}}$. The algorithm executes in user-defined time intervals and re-partitions the system if the application's current performance p is below or above the user-defined performance bound $\mathscr{P}$ by either calling `add_core()` or `remove_core()`. The repartitioning is achieved by reconfiguration/release of a hardware slot (for hardware threads) or the activation/deactivation of a worker processor (for software threads). In case the performance is below the bound, the master creates an additional instance of the thread on the core that promises the largest increase in performance. If the performance exceeds the bound $\mathscr{P}$, the master terminates the instance of the thread that will free a core, but keep the application performance above $\mathscr{P}$, if such a thread exists.

In lines 9–17 of Algorithm 2, the function `add_core()` computes for all threads $T_i \in T$ the possible rewards in execution time $reward_{i,j'}$ in case an additional thread instance is started on an idle core. The function only considers idling worker cores (line 11). With $A(C_j)$ as the set of thread instances that execute on core C_j, an idling core is characterized by $A(C_j) = \emptyset$. For other cores, the reward is set to 0, in line 14. Then, in line 18, the function selects the thread instance $T_{i,j}$ that maximizes the estimated performance. Therefore, we estimate the performance $p_{reward_{i,j}}$ assuming that the execution time of the thread instances running on the master core can be reduced by $reward_{i,j}$. In case of several threads promising the same performance, we break ties by randomly choosing a thread. Finally, in lines 19–22, we create a new thread instance $T_{i',j'}$ and update the cumulated estimated speedup $S_{i'}$ and $A(C_{j'})$. When there is no idling worker core, the maximum achievable reward is 0. In this case, the current partitioning remains unchanged.

The function `remove_core()`, in lines 24–39, operates in a similar way. We terminate the thread instance $T_{i,j}$ that promises the lowest increase in execution time (penalty). We do not deactivate a core, if the estimated performance drops below the user-defined bound $\mathscr{P}$ (line 35). In line 27 of Algorithm 2, we make sure that we only consider threads executing on worker cores ($T_i \in A(C_j) \wedge C_j \neq$ master).

Algorithm 2 Pseudo code for ATP$_{bound}$

```
Require: set $T$ of active threads and their instances $T_{i,j}$, set $C$ of heterogeneous cores,
    thread execution times $e_{i,master}$ measured on the master processor, current applica-
    tion performance $p$.
 1: procedure ATP$_{bound}$
 2:     if p $< \mathscr{P}$ then
 3:         add_core()
 4:     else if p $> \mathscr{P}$ then
 5:         remove_core()
 6:     end if
 7: end procedure
 8: procedure ADD_CORE
 9:     for $i = 1,\ldots,|T|$ do
10:         for $j = 1,\ldots,|C|$ do
11:             if $A(C_j) = \emptyset \wedge C_j \neq master$ then
12:                 $reward_{i,j} = e_{i,master}\frac{s_{i,j}}{S_i + s_{i,j}}$
13:             else
14:                 $reward_{i,j} = 0$
15:             end if
16:         end for
17:     end for
18:     $i', j' = \arg\max_{i,j}\{reward_{i,j}\}$
19:     if $reward_{i',j'} > 0$ then
20:         create $T_{i',j'}$
21:         $S_{i'} = S_{i'} + s_{i',j'}; A(C_{j'}) = \{T_{i'}\}$
22:     end if
23: end procedure
24: procedure REMOVE_CORE
25:     for $i = 1,\ldots,|T|$ do
26:         for $j = 1,\ldots,|C|$ do
27:             if $T_i \in A(C_j) \wedge C_j \neq master$ then
28:                 $penalty_{i,j} = e_{i,master}\frac{s_{i,j}}{S_i - s_{i,j}}$
29:             else
30:                 $penalty_{i,j} = \infty$
31:             end if
32:         end for
33:     end for
34:     $i', j' = \arg\min_{i,j}\{penalty_{i,j}\}$
35:     if $penalty_{i',j'} < \infty \wedge p_{penalty_{i,j}} > \mathscr{P}$ then
36:         terminate $T_{i',j'}$
37:         $S_{i'} = S_{i'} - s_{i',j'}; A(C_{j'}) = \emptyset$
38:     end if
39: end procedure
```

4.2 ATP$_{\text{budget}}$ algorithm

Algorithm 3 presents the pseudo code for our budget-based add/remove self-adaptation technique ATP$_{\text{budget}}$. The algorithm is similar to the ATP$_{\text{bound}}$ algorithm, but differs in the objective. While the ATP$_{\text{bound}}$ algorithm tries to keep the application's performance above a performance bound, the ATP$_{\text{budget}}$ algorithm seeks to stay inside a performance budget. The budget is spanned around a user-defined target performance $\mathscr{P}_{target}$, where an user-defined aberration is allowed. Occurring deviations from the performance budget are weighed equally in the sense that partitionings which are $\times$ times faster are equivalent to those which are $\times$ times slower.

Algorithm 3 Pseudo code for ATP$_{budget}$

```
Require: set $T$ of active threads and their instances $T_{i,j}$, set $C$ of heterogeneous cores,
    thread execution times $e_{i,master}$ measured on the master processor, current applica-
    tion performance $p$.
 1: procedure ATP$_{budget}$
 2:     if p $< \mathscr{P}_{lower}$ then
 3:         add_core()
 4:     else if p $> \mathscr{P}_{upper}$ then
 5:         remove_core()
 6:     end if
 7: end procedure
 8: procedure ADD_CORE
 9:     for $i = 1,\ldots,|T|$ do
10:         for $j = 1,\ldots,|C|$ do
11:             if $A(C_j) = \emptyset \wedge C_j \neq master$ then
12:                 $reward_{i,j} = e_{i,master}\frac{s_{i,j}}{S_i + s_{i,j}}$
13:             else
14:                 $reward_{i,j} = 0$
15:             end if
16:         end for
17:     end for
18:     $i', j' = \arg\min_{i,j} \frac{\max(\mathscr{P}_{target}, p_{reward_{i,j}})}{\min(\mathscr{P}_{target}, p_{reward_{i,j}})}$
19:     if $\frac{\max(\mathscr{P}_{target}, p_{reward_{i',j'}})}{\min(\mathscr{P}_{target}, p_{reward_{i',j'}})} < \frac{\max(\mathscr{P}_{target}, p)}{\min(\mathscr{P}_{target}, p)}$ then
20:         create $T_{i',j'}$
21:         $S_{i'} = S_{i'} + s_{i',j'}; A(C_{j'}) = \{T_{i'}\}$
22:     end if
23: end procedure
24: procedure REMOVE_CORE
25:     for $i = 1,\ldots,|T|$ do
26:         for $j = 1,\ldots,|C|$ do
27:             if $T_i \in A(C_j) \wedge C_j \neq master$ then
28:                 $penalty_{i,j} = e_{i,master}\frac{s_{i,j}}{S_i - s_{i,j}}$
29:             else
30:                 $penalty_{i,j} = \infty$
31:             end if
32:         end for
33:     end for
34:     $i', j' = \arg\min_{i,j} \frac{\max(\mathscr{P}_{target}, p_{penalty_{i,j}})}{\min(\mathscr{P}_{target}, p_{penalty_{i,j}})}$
35:     if $\frac{\max(\mathscr{P}_{target}, p_{penalty_{i',j'}})}{\min(\mathscr{P}_{target}, p_{penalty_{i',j'}})} < \frac{\max(\mathscr{P}_{target}, p)}{\min(\mathscr{P}_{target}, p)}$ then
36:         terminate $T_{i',j'}$
37:         $S_{i'} = S_{i'} - s_{i',j'}; A(C_{j'}) = \emptyset$
38:     end if
39: end procedure
```

The algorithm executes in user-defined time intervals and re-partitions the system only if the application's current performance p is outside the specified budget $[\mathscr{P}_{lower}, \mathscr{P}_{upper}]$ by either calling `add_core()` or `remove_core()` (lines 1–6). This reduces the number of function calls of `add_core()` or `remove_core()`. In case the performance drops below $\mathscr{P}_{lower}$, the master creates an additional thread instance on the core that promises either meeting the desired performance budget, if possible, or else the largest increase in performance. When the performance exceeds an upper threshold $\mathscr{P}_{upper}$, the master terminates the instance of the thread that will lead to the reduction which is as close as possible to the desired performance target. The `add_core()` function is modified in the lines 18–19, to select the partitioning that relatively approximates $\mathscr{P}_{target}$ best. Corresponding modifications can be found for the function `remove_core()`, in lines 34–35.

The `add_core()` and `remove_core()` functions of $\text{ATP}_{\text{budget}}$ and $\text{ATP}_{\text{bound}}$ have an execution time of $O(|C||T|)$.

5 Real-time video 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 [10] 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 (particle) in each subsequent frame (measurement) of the video sequence by comparing the histogram of each particle with the histogram of the initial selection (reference). Here, the observation stage creates the histograms for each particle and the importance stage performs the histogram comparison. An example of the desired tracking behavior can be seen in Fig. 6. For more details on the state and measurement models of this case study, we refer to Happe et al. [9].

5.1 Experimental setup

Our adaptive prototype is designed according to the reference architecture depicted in Fig. 1 and consists of a master processor, a worker processor, and two reconfigurable hardware slots. We have implemented the prototype on a Virtex-4 XC4VFX100 FPGA. 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 to enable transparent utilization of OS services by the hardware. A second embedded PowerPC 405 CPU, clocked at 300 MHz, is used as a worker processor. In all experiments, the particle filter tracks $N = 100$ particles divided into chunks of 10.

The main goal of our self-adaptive approach is to increase the efficiency with which an application designed using our framework utilizes the reconfigurable area and the worker processors. Regarding our object tracking case study and depending on the input data characteristics and the current predicted system state, different HW/SW partitionings with different area or core requirements may suffice to keep the tracking performance (measured in frames per second) inside a user-defined performance budget. Therefore, we apply the add/remove self-adaptation algorithm introduced in Sect. 4 to minimize the required processing resources. The video object tracker is a prime example to demonstrate the self-adaptive task partitioning for two reasons. First, the required processing power strongly depends on the current contents of the video frames and can vary significantly. Second, the data-parallelism of individual stages of the application favors multithreaded execution where adding more instances of the same thread helps to increase performance (see Fig. 3).

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. Thus, we focus on the HW/SW partitionings defined in Table 1.

5.2 Static partitionings

Figure 7 depicts the performance figures, measured in clock cycles per frame, of the individual partitionings when tracking the soccer player in the video sequence as shown in Fig. 6. For visual clarity, Fig. 7 is split into three diagrams, showing partitionings using one (top), two (middle), and three (bottom) worker cores.

It can be seen that the overall particle filter performance is data-dependent for all partitionings. This is mainly due to the histogram creation in the observation stage, as the computational complexity per particle depends on the size of the bounding box. In the first 100 frames, when the player occupies a larger region in the foreground, the filter performance is comparably low. Over the following 100 frames, the performance increases as the player retreats

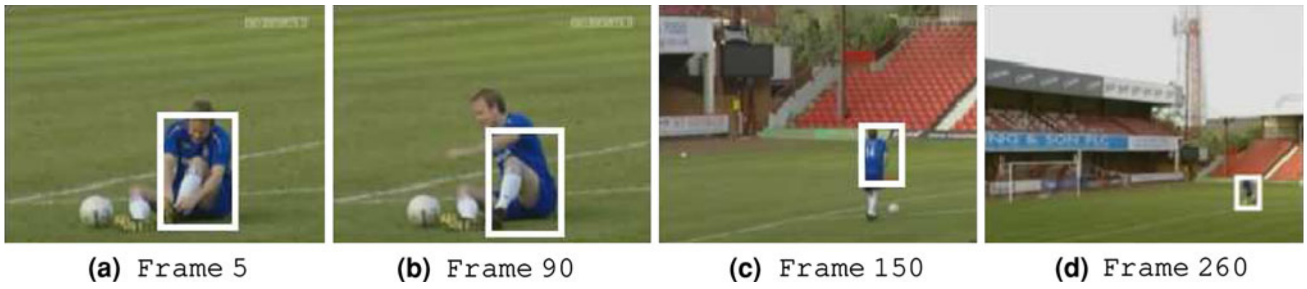

Fig. 6 Object tracking in a video sequence: the desired behavior of the object tracking is that the position and approximated outline of the player is tracked and visualized using a bounding box. Happe et al. [8]

Table 1 HW/SW partitionings

sw	All threads run in software
hw_i	One importance thread executes in HW
hw_o	One observation thread executes in HW
sw_i	One CPU–HW importance thread
sw_o	One CPU–HW observation thread
hw_{ii}	Two importance threads execute in HW
hw_{oo}	Two observation threads execute in HW
hw_{oi}	One observation thread, one importance thread in HW
sw_i, hw_i	One CPU–HW, one HW importance thread
sw_i, hw_o	One CPU–HW importance, one HW observation thread
sw_o, hw_i	One CPU–HW observation, one HW importance thread
sw_o, hw_o	One CPU–HW and one HW observation thread
sw_i, hw_{ii}	One CPU–HW, two HW importance threads
sw_i, hw_{oo}	One CPU–HW importance, two HW observation threads
sw_i, hw_{oi}	One CPU–HW, one hw importance, one HW observation thread
sw_o, hw_{ii}	One CPU–HW observation, two HW importance threads
sw_o, hw_{oo}	One CPU–HW and two HW observation threads
sw_o, hw_{oi}	One CPU–HW, one HW observation, one HW importance thread

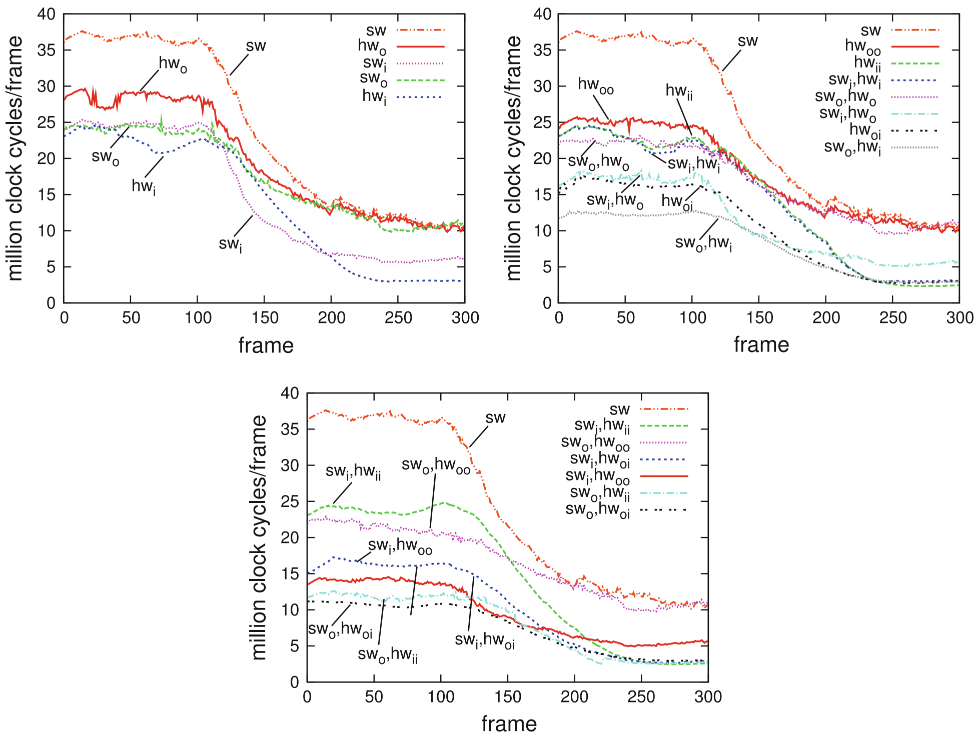

Fig. 7 Static measurements for object tracking in a video sequence (*soccer*) using one worker core (*top*), two worker cores (*middle*) and three worker cores (*bottom*)

into the background, since smaller bounding boxes and thus fewer pixels need to be observed per frame. In the soccer video, the number of pixels occupied by a particle's bounding box varies over time by a factor of 40, resulting in large variations of the observation stage's execution time.

In contrast, the performance of the importance stage remains constant over the course of the sequence because the filter compares fixed-size color histograms which do not depend on the particle's scaling factor. Thus, mapping a single importance thread to a hardware or software core benefits the performance across the entire video.

The difference in the input-data dependency of the observation and importance stages is illustrated in Fig. 8, which highlights the individual stages's execution times over the video sequence of a software-only partitioning (*sw*).

Additional observation threads that execute in parallel improve the performance significantly when the player is in

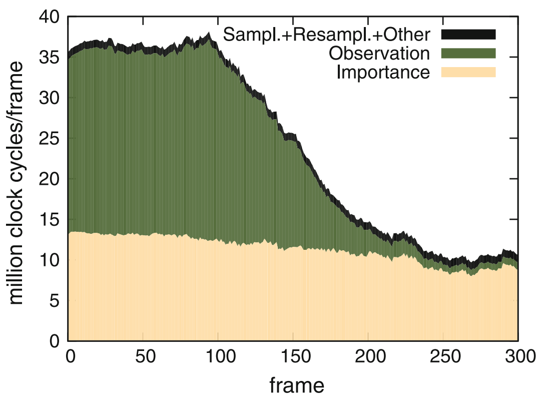

Fig. 8 Individual execution times of the observation and importance stages for each frame when the application runs entirely on the master

the foreground and the histogram calculation involves a higher number of pixels, i.e., in the first 100 frames of the video. Consequently, this interval shows the most interesting differences between different HW/SW partitionings.

When looking at the differences in performance of individual worker cores, the CPU–HW thread implementation for the observation stage generally shows better performance than the hardware thread implementation. However, this difference becomes insignificant when the tracked object's size decreases, and the worker cores cannot markedly reduce the observation stage's workload for the master core. In general, a combination of both hardware and CPU–HW threads shows the best performance, as it utilizes more computing resources and enables the highest amount of parallel execution.

5.3 Self-adaptation strategy with bound

The time interval for running the self-adaptation algorithm controls a trade-off between the overhead incurred by partial reconfiguration and the latency with which the algorithm reacts to changing data-dependent thread performance. The overhead influences overall application efficiency, while the reaction latency is relevant for the real-time characterization of the system. In this example, we execute the self-adaptation algorithm every 20 frames with an initial offset of 8 frames. The speedup values $s_{i,j}$ are obtained in static measurements and given in Table 2.

Figure 9 shows an exemplary run of the system using the ATP$_{\text{bound}}$ algorithm (Algorithm 2) that uses a single lower performance constraint and removes thread instances on worker cores if the estimated resulting performance remains above the user-defined soft real-time constraint. As input, the soccer video (see Fig. 6) was altered, such that the first 300 frames are played four times (forward, backward, forward, backward). This leads to a video where the player runs into the background and then returns to the foreground, twice, implied at the top of Fig. 9. This exemplary run meets the real-time constraint 86% of the time, excluding the initialization phase within the first 30 frames where first thread instances are created on the worker cores. Although all cores run thread instances of the framework, the constraint is missed in an interval of about 150 frames (frames 570–720). Here, the system is partitioned in an optimal way for the given layout of worker CPUs and HW slots. In other words, no other partitioning could achieve the user-defined soft-real time constraint for this input data.

Table 2 Speedup values $s_{i,j}$ for thread instances $T_{i,j}$

Stage	sw	hw	CPU–HW
Observation	1.0	0.6	1.3
Importance	1.0	14.4	1.4

5.4 Self-adaptation strategy with budget

Figure 10 shows an exemplary run of our self-adaptive task partitioning system for the altered soccer video using the ATP$_{\text{budget}}$ algorithm (Algorithm 3). Again, with an offset of 8 frames, the self-adaptation technique verifies every 20 frames whether an adaptation is needed based on performance measurements on the master core. The application's performance is measured in frames per second and the desired average performance range is set to 8 FPS, where the budget is set to be 33% faster or slower than the defined average performance.

The system starts with all threads running on the master core. Since the resulting performance is lower than the threshold, the first self-adaptation occurs at the 8th frame. At this point, the observation stage takes about 60% of the system's execution time. The CPU–HW thread offers the highest speedup and it is selected by the algorithm to boost the performance from 2.67 to 4.15 FPS. At the next self-adaptation point (frame 28), the performance still does not meet the performance requirements. Now, the importance stage consumes more than 54% of the system's execution time on the master core. The instantiation of an additional hardware thread promises a total performance of 8.33 FPS, which is close to the desired average performance. Thus, an importance stage thread instance is reconfigured into one of the hardware slots. Due to the reconfiguration overhead, the importance thread becomes available with an offset of 8 frames.

After this, the system's performance is inside the requested performance budget for about 120 frames before it exceeds the upper threshold. Now, the system consumes more processing resources than necessary; hence, both worker cores are released again and the application

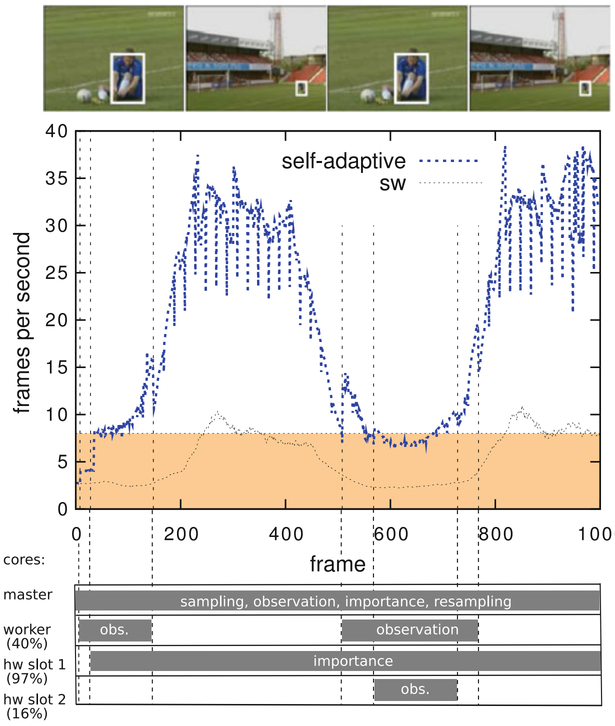

Fig. 9 Self-adaptation exemplary run considering a soft real-time constraint of 8 FPS: resulting performance in frames/second (*upper part*) and thread assignment (*lower part*). Re-partitioning points are represented by *vertical dashed lines*. The performance target is avoiding the *highlighted horizontal bar*

continues with the initial partitioning where the entire application runs on the master processor (frame 188). Then, the measured performance stays inside the budget for 280 frames, while entirely executing on the master core. At frames 468, 508, and 528, new cores are added to face the dropping performance, which cannot be completely compensated because the input data complexity overburdens the system for the desired performance constraint. All three worker cores are later deactivated at frames 768, 868, and 948.

This exemplary run stays inside the performance budget for 65.75% of the time, the performance exceeds the budget for 10.1% of the time, and it falls below the budget for 24.1% of the time. The average performance is 7.18 FPS which is inside the desired performance budget but below the desired average performance. Compared to the exemplary run in Fig. 9, the overall worker core utilization reduces from 51 to 46%, where we assume that a worker core is fully utilized in an active state and unutilized in a deactivated state.

5.5 Resource requirements

The resource requirements for the reference design (Fig. 1) on a Virtex-4 XC4VFX100 FPGA are listed in Table 3.

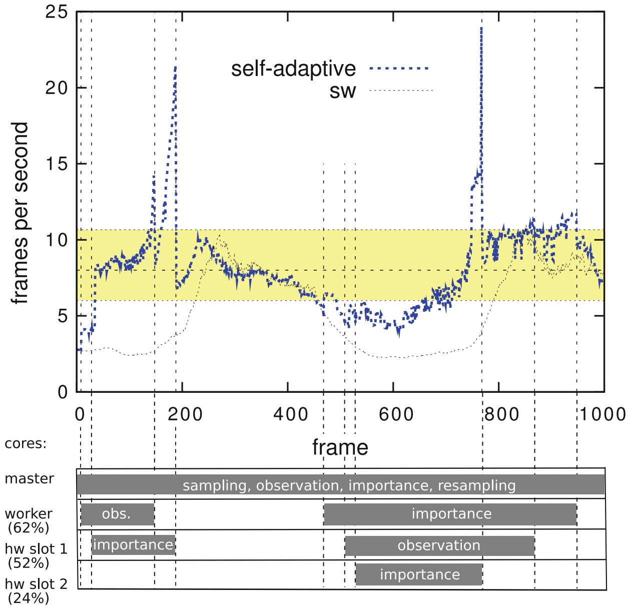

Fig. 10 Self-adaptation exemplary run: resulting performance in frames/second (*upper part*) and thread assignment (*lower part*). Re-partitioning points are represented by *vertical dashed lines*. The performance target is highlighted by a *horizontal bar*

Table 3 Resource requirements in slices of the self-adaptive system (see Fig. 1) implemented on a Virtex-4 XC4VFX100 FPGA

Static design with one worker processor and empty slots (%)	10,895 (25.8)
Size of 1st slot (%)	7,473 (17.7)
Size of 2nd slot (%)	7,923 (18.8)
Importance hardware thread (%)	2,792 (6.6)
Observation hardware thread (%)	5,020 (11.9)

The reference design consists of a static design including the OS infrastructure, the master CPU, one CPU–HW thread and two hardware slots. Hardware threads available for the importance and the observation stage, which occupy roughly the same amount of slices for both slots.

6 Conclusion

SMC methods are a natural choice for on-line estimation of nonlinear systems, such as the tracking of an object in a video sequence. However, the widespread adoption of SMC methods within traditional real-time constrained embedded systems is hampered by three major obstacles: the application-specific input data dependencies make the design of efficient, but real-time capable systems—with task allocation at design time—infeasible. Attaining a significant accuracy in the state estimation calls for high-computational performance and dedicated hardware support; and the heterogeneous nature of SMC methods with both control- and data-intensive components does not map well to pure software- or hardware-based systems. In general, the specific requirements of different application domains in which individual particle filters are applied, as well as significant input data dependencies increase the need for flexible and even self-adaptive run-time environments.

In this article, we have presented a real-time capable video object tracking application based on a multi-threaded framework for implementing SMC methods on hybrid CPU/FPGA platforms. We have demonstrated how using a multithreaded HW/SW programming model can enable the system to react to changing requirements during run-time by employing a thread-centric add/remove adaptation strategy. The programming model also significantly simplifies the exploration of possible hardware/software partitionings at design time. Besides meeting the demanding requirements of real-time affinity, heterogeneity in hardware and software, flexible repartitioning and dynamic self-adaptivity, our particle filter framework provides a powerful SMC design methodology which autonomously handles the recurring low-level tasks common to all SIR-based methods and presents a flexible implementation and execution environment for particle filters targeted at heterogeneous and embedded multi-core systems. We have presented a video object tracking case study based on this framework and have quantitatively evaluated the performance of different static and dynamic partitionings on a prototype employing a mix of custom hardware cores and dedicated microprocessors. The experiments show that our self-adaptation technique is able to effectively re-partition the HW/SW composition of a real-world tracking application to meet predefined soft real-time constraints while minimizing the number of active cores.

Future work involving our framework is mainly targeted at the self-adaptation capabilities of the system. More specifically, we plan to improve the framework's handling of software threads on worker processors by following two distinct approaches. First, we intend to increase the CPU cores' utilization and allow them to handle multiple stages at the same time by enabling the operating system to transparently schedule multiple software threads onto a single worker CPU. As a second step, we will try to combine the reconfigurability of the hardware slots with software-based processing to increase the flexibility of the on-line self-adaptation technique by dynamically configuring soft-core processors, such as the Xilinx MicroBlaze, to the reconfigurable hardware slots. Also, we will integrate existing research on reduction of reconfiguration overheads to further increase the efficiency of the framework.

Acknowledgments The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 257906. This work was supported by the International Graduate School of Dynamic Intelligent Systems and by the German Research Foundation under project number PL471/2-1.

References

1. Ali, U., Malik, M., Munawar, K.: FPGA/soft-processor based real-time object tracking system. 5th Southern Conference on Programmable Logic (2009)
2. Arulampalam, M.S., Maskell, S., Gordon, N., Clapp, T.: A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. IEEE Trans. Signal Process. **50**(2), 174–188 (2002)
3. Carvalho, E., Calazans, N., Moraes, F.: Heuristics for dynamic task mapping in NoC-based heterogeneous MPSoCs. International Workshop on Rapid System Prototyping (RSP) (2007)
4. Cho, J.U., Jin, S.H., Pham, X.D., Kim, D., Jeon, J.W.: A real-time color feature tracking system using color histograms. International Conference on Control, Automation and Systems (2007)
5. Curtis-Maury, M., Dzierwa, J., Antonopoulos, C.D., Nikolopoulos, D.S.: Online power-performance adaptation of multithreaded programs using hardware event-based prediction. International Conference on Supercomputing (2006)
6. Doucet, A., de Freitas, N., Gordon, N.: Sequential Monte Carlo Methods in Practice. Springer, Berlin (2001)
7. Gilbert, A.L., Giles, M.K., Flachs, G.M., Rogers, R.B., Hsun, U.Y.: A real-time video tracking system. IEEE Trans. Pattern. Anal. Mach. Intell. **2**, 47–56 (1980)
8. Happe, M., Lübbers, E., Platzner, M.: A multithreaded framework for sequential Monte Carlo methods on CPU/FPGA platforms. International Workshop on Applied Reconfigurable Computing (ARC) (2009a)
9. Happe, M., Lübbers, E., Platzner, M.: An adaptive sequential Monte Carlo framework with runtime HW/SW partitioning. IEEE Int. Conf. Field Program. Technol. (FPT) (2009b)
10. Hess, R.: Particle filter object tracking. http://web.engr.oregonstate.edu/hess (2006)
11. Huang, C., Vahid, F.: Dynamic coprocessor management for FPGA-enhanced compute platforms. International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) (2008)
12. Kalman, E.R.: A new approach to linear filtering and prediction problems. Trans. ASME—J. Basic Eng. **82**(Series D), 35–45 (1960)
13. Kobayashi, T., Nakagawa, K., Imae, J., Zhai, G.: Real Time object tracking on video image sequence using particle swarm optimization. International Conference on Control, Automation and Systems (ICCAS) (2007)
14. Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M.: Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In: International Symposium on Microarchitecture (2003)
15. Loza, A., Patricio, M., Garcia, J., Molina, J.: Advanced algorithms for real-time video tracking with multiple targets. 10th International Conference on Control, Automation, Robotics and Vision (ICARCV) (2008)
16. Lübbers, E., Platzner, M.: ReconOS: multithreaded programming for reconfigurable computers. ACM Trans. Embed. Comput. Syst. **9**(1), 1–33 (2009)
17. Nollet, V., Avasare, P., Eeckhaut, H., Verkest, D., Corporaal, H.: Run-time management of a MPSoC containing FPGA fabric tiles. Trans. Very Large Scale Integr. Syst. (2008)

18. Porikli, F.: Achieving real-time object detection and tracking under extreme conditions. J. Real-time Image Proc. **1**(1), 33–40 (2006)
19. Saha, S., Bambha, N.K., Bhattacharyya, S.S.: A parameterized design framework for hardware implementation of particle filters. IEEE International Conference on Acoustics, Speech and Signal Processing (2008)
20. Sankaranarayanan, A.C., Chellappa, R., Srivastava, A.: Algorithmic and architectural design methodology for particle filters in hardware. International Conference on Computer Design (2005)
21. Sigdel, K., Thompson, M., Pimentel, A.D., Galuzzi, C., Bertels, K.: System-level runtime mapping exploration of reconfigurable architectures. Proceedings of the International Symposium on Parallel & Distributed Processing (IPDPS) (2009)
22. Sironi, F., Triverio, M., Hoffmann, H., Maggio, M., Santambrogio, M.: Self-aware adaptation in FPGA-based systems. International Conference on Field Programmable Logic and Applications (FPL) (2010)
23. Smit, L.T., Smit, G.J.M., Hurink, J.L., Broersma, H., Paulusma, D., Wolkotte, P.T.: Run-time assignment of tasks to a heterogeneous processors. In: Embedded Systems Symposium (2004)
24. Smit, L.T., Hurink, J.L., Smit, G.J.M.: Run-time mapping of applications to a heterogeneous SoC. Proceedings of the International Symposium on System-on-Chip (2005)
25. Stitt, G., Lysecky, R., Vahid, F.: Dynamic hardware/software partitioning: a first approach. Proceedings of the Design Automation Conference (DAC) (2003)
26. Visentini, I., Snidaro, L., Foresti, G.L.: Cascaded online boosting. J. Real-time Image Proc. **5**(4), 245–257 (2010)
27. Woelk, F., Schiller, I., Koch, R.: An airborne bayesian color tracking system. In: IEEE Intelligent Vehicles Symposium (2005)

Author Biographies

Markus Happe is PhD student in the field of Computer Engineering at the International Graduate School Dynamic Intelligent Systems in Paderborn, Germany. He holds a master degree in computer science (University of Paderborn, 2008). His research interests include heterogeneous multi-core architectures, embedded operating systems, and self-adaptation strategies.

Enno Lübbers is a researcher for reconfigurable systems and embedded architectures for the EADS corporate research facility, Innovation Works, in Munich, where he works on design methods for reconfigurable hardware and multi-core avionics systems. Previously, he held a research position in Computer Engineering at University of Paderborn. He received a Diploma degree in Computer and Communication Systems Engineering from the Technical University of Braunschweig (2005) and a PhD degree in Computer Engineering from the University of Paderborn (2010). His research interests include programming models and system software for embedded and reconfigurable systems, hardware/software co-design and parallel computer architectures.

Marco Platzner is professor for Computer Engineering at the University of Paderborn. Previously, he held research positions at the Computer Engineering and Networks Lab at ETH Zurich, Switzerland, the Computer Systems Lab at Stanford University, USA, the GMD-Research Center for Information Technology (now Fraunhofer IAIS) in Sankt Augustin, Germany, and the Graz University of Technology, Austria. He holds diploma and PhD degrees in Telematics (Graz University of Technology, 1991 and 1996), and a "Habilitation" degree for the area hardware–software codesign (ETH Zurich, 2002). He is senior member of the IEEE, member of the ACM, member of the board of the Paderborn Center for Parallel Computing and faculty member of the International Graduate School Dynamic Intelligent Systems of the University of Paderborn and of the Advanced Learning and Research Institute (ALaRI) at Universita della Svizzera Italiana (USI), in Lugano. His research interests include reconfigurable computing, hardware–software codesign, and parallel architectures.