Des Autom Embed Syst (2012) 16:45–69
DOI 10.1007/s10617-012-9084-z

On fast development of FPGA-based SOA services—machine vision case study

A. Ruta · R. Brzoza-Woch · K. Zielinski

Received: 9 March 2011 / Accepted: 14 March 2012 / Published online: 30 March 2012

Abstract Development of FPGA-based, network-enabled embedded systems in Register Transfer Level hardware description languages is tedious. Despite the automation of this process with numerous EDA tools available, no well-established design patterns exist. Moreover, the entire production cycle requires appropriate theoretical background and hardware design intuition from the developer which discourages the software community. To improve productivity and minimize time to market when assembling such systems, we propose a new hardware/software co-design approach to building reconfigurable hardware web services. The proposed integrated development platform features a programmable FPGA board where computations of different nature and purpose are logically distributed among a sequential soft-core processor program, a massively parallel accelerator and an independent communication module that handles remote clients' requests. Our second contribution is a set of tools that make the development of the aforementioned services essentially a software design undertaking with the extensive use of high-level programming languages. The platform has been tuned to act as a flexible runtime environment for image processing services, thus providing functionality of an intelligent camera. Two example services built from scratch according to the new methodology are discussed. Reduced development time and significant performance gain observed prove validity of the proposed approach and unveil a large potential of the assembled prototype.

Keywords HW/SW co-design · FPGA · Parallel image processing · Service-oriented architecture · High-level synthesis

The research presented in this paper has been partially supported by the European Union within the European Social Fund program no. UDA-POKL.04.01.01-00-367/08-00 and the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

A. Ruta (✉) · R. Brzoza-Woch · K. Zielinski
Department of Computer Science, AGH University of Science and Technology, Kraków, Poland
e-mail: aruta@agh.edu.pl

R. Brzoza-Woch
e-mail: rabw@agh.edu.pl

K. Zielinski
e-mail: kz@agh.edu.pl

1 Introduction

In a modern fast-paced environment humans are more and more heavily dependent on various types of embedded devices, such as smart sensors or cameras. The need for processing large volumes of data and the growing popularity of the "Internet of things" concept will strengthen this tendency in the forthcoming decades. In this context the traditional computing paradigms relying on the stationary infrastructure often no longer match our expectations in terms of the quality of service and/or response time. Such stationary systems enforce the model of computation in which the role of sensors is reduced to merely data acquisition. This in turn means suboptimal resource utilization, large communication overhead and increased costs. In this light, gradual adoption of autonomous, network-enabled hardware services, capable of running complex data processing straight on chip, is inevitable.

Field-Programmable Gate Array (FPGA) is a leading technology offering reconfigurable devices where a broad range of algorithms can be implemented in a massively parallel way. We attempt to show that this feature makes FPGAs a particularly useful tool for building computation-intensive hardware web services. In our opinion the major problem that has so far prevented hardware designers from utilizing FPGAs in service-oriented architectures is the complexity of the development cycle. Unlike in the software engineering domain, hardware developers lack both high-level programming languages/environments and well-established design patterns. Besides, deploying even a simple computational process on an FPGA chip is not approachable to a person without appropriate theoretical background and a substantial hardware design experience. Our work aims at filling in this gap by allowing software-style design and implementation methodologies in essentially hardware-based systems development.

In this paper we present an integrated development platform that dramatically simplifies the process of building reconfigurable, FPGA-based services according to the *Service Oriented Architecture* (SOA) paradigm [11]. Our contribution includes a versatile hardware device that can host web services and flexible software tools which together form a novel alternative to the traditional FPGA development toolkit. Without compromising the generality of the architecture and the design methodologies adopted, we demonstrate our approach in a machine vision domain by tuning the platform to provide functionality of an intelligent camera.

Our hardware features an off-the-shelf FPGA chip and a set of appropriately interconnected peripherals that are optimized to give maximum performance for this particular class of systems. The model of computation adopted assumes a central role of a soft-core processor that manages data I/O, participates in the client-server-style dialogue between the device and the remote clients, and optionally performs less demanding sequential data processing. The desired computational speedup is achieved through offloading the most critical operations to dedicated logic elements where the advantages of the parallel nature of FPGAs can be fully exploited.

On the software side we introduce two applications. The first one manages the process of building and deploying a service by enabling the designer to run appropriate external Electronic Design Automation (EDA) software packages and configuration scripts from a single graphical interface. In addition, a service project file is maintained throughout the development cycle where all changes made so far can be saved and retrieved at any later time. The second application is a graphical design front-end for the *Impulse CoDevelper* software. It allows the designer to rapidly define the topology of the hardware accelerators. The exact definition of the parallel processes underlying these accelerators is further specified in an *Impulse C* language [21] that is automatically translated into VHDL or Verilog.

The rest of this paper is organized as follows. In Sect. 1.1 we review the related work on FPGA-based services and embedded image processing. In Sect. 2 we introduce the hardware platform underlying the proposed service runtime environment. Section 3 is focused on the software tools that were built to manage the process of designing, configuring and deploying services in FPGAs. In Sect. 4 two example services developed using the aforementioned tools are discussed. Finally, Sect. 5 concludes this paper.

1.1 Related work

Generally, FPGA is a relatively fresh technology with only a limited number of applications in embedded image processing. Therefore, it is difficult to find FPGA-based solutions to many complex problems that are central to computer vision. In most cases FPGAs are used for basic signal processing under the hardware-software co-design scheme. In this scenario the hardware acceleration affects the execution of only computationally the most expensive subtasks within a larger process, i.e. video encoding/decoding or digital filtering. Other tasks are executed on a soft-core processor, possibly under control of an embedded operating system, such as in [4].

Typically, the above mentioned image preprocessing is done using well-known algorithms, parallelized to meet the real-time execution requirements. For instance, as early as in mid 1990s Gent et al. [13] boosted a deformable template image segmentation algorithm by offloading it to an FPGA-based co-processor. Neoh and Hazanchuk [19] used FPGAs to implement a real-time Canny edge detector [6]. Djemal et al. [9] implemented a modified version of Nagao filter for edge-preserving, real-time video smoothing. Baumann and Tinembart [3] built a library of basic morphological operations for implementation in FPGAs. This library was later used to assemble more complex applications involving vision-based robots. Algorithms for accelerated feature-based image correlation proposed by Villalpando and Some [27] were used in a similar domain. Many commercial EDA tools used for analysis and synthesis of HDL designs, such as *Quartus II* software from *Altera*, offer their own IP core libraries for basic image processing.

There are also known applications of FPGAs to solving more advanced machine vision problems. For instance, Arias-Estrada and Rodríguez-Palacios [2] implemented a high-speed object tracker based on Hausdorff distance [16] in which part of the computations involved were offloaded to programmable logic. Ali et al. [1] combined a Xilinx's FPGA chip with a Microblaze soft-core processor to build an efficient object tracker based on a simplified kernel mean shift algorithm. Their system successfully tracks moving targets within the required frame rate constraint and has good prospects for future extensions. FPGAs have already been used for visual pattern recognition too. For instance, Meng et al. [18] developed a reconfigurable, FPGA-based video processing architecture and deployed a human action recognition system on it.

The idea of using web services on embedded devices, such as smartphones, or in wireless sensor networks is not new [15, 23]. A number of web service development toolkits exist for both *J2ME* and *.NET Compact Framework*, as well as a platform-independent *gSOAP* toolkit [26]. The implementation is normally done on a microcontroller or microprocessor with appropriately large computing power [5, 17]. The number of recognizable FPGA-based implementations is very limited though. For instance, Cuenca-Asensi et al. [8] presented an architecture based on a Celoxica RC203E FPGA board that supported SOAP web services. It was used to demonstrate a remote Wake-on-LAN service with the average response time shorter than that reported using the PC-based implementation. On the other hand, Chang et al. [7] deployed a HTTP-based REST-ful web service [12] for home device control application on an embedded system using a Xilinx Spartan-3E Starter FPGA board.

Another interesting approach was proposed by Gonzalez et al. [14]. They used an FPGA module located on a PCI board that was attached to a PC. For this module hardware implementations of C library functions were provided and for each such function a separate web service was created. As a result, remote users working on PCs not equipped with the aforementioned board could write efficient C programs by simply replacing local library function calls with the appropriate service invocations. This particular realization of a "remote coprocessor" idea is useful whenever the client implements the main application logic and is in the same time a source of all input data. Besides, acceleration rate of the operations being ported to hardware must be large enough to compensate the resulting network delays.

In contrast, the device introduced in this paper can "sense the world" itself, process the acquired data in an autonomous manner, and optionally react to the environmental changes while retaining a web service's functionality. This autonomy distinguishes the class of systems we target at from the body of previous FPGA applications. Moreover, it can be realized on such a high functional level (e.g. complete license plate recognition) that only very small amounts of data need to be sent over the network. This plays a crucial role in limiting the required network traffic that has to be handled by the system.

Another significant limitation of the previous approaches to embedded web services design lies in that their core logic was mostly coded using low-level Hardware Description Languages (HDLs), usually VHDL or Verilog. The system presented by Cuenca-Asensi et al. [8] is one of the exceptions (Handel-C was used in this case). HDLs have many properties of modern programming languages which makes them suitable for general-purpose use. However, implementing algorithms in HDL is generally difficult to the software developers as they must adopt a completely different programming philosophy where the issues normally transparent to them, such as parallel execution, process synchronization or elementary operation timing, now do matter. As a result, it is relatively easy for an inexperienced programmer to produce code that simulates successfully but cannot be synthesized into a real device.

The approach adopted in this work offers an adequate solution to the above problems. Namely, we propose a hardware-software co-design methodology which significantly reduces the development effort required. Complex and time-critical tasks are implemented using a C-to-hardware compiler which preserves much of the HDL code's performance but simplifies parallel implementation of algorithms. The computationally less demanding tasks (such as web server implementation) are conveniently implemented in a microcontroller core embedded in FPGA so as to maximize the ease of implementation and reusability of the freely available code (for example an XML parser). Finally, we propose high-level software design tools that facilitate creation of optimized source code and output firmware for the target SOA services.

2 Hardware platform

Embedded service architectures require carefully planned design of hardware components and their interconnections. Such architectures must not only provide necessary computational resources, but also deliver means of implementing the required functionalities in conformance with the software service design standards. Adopting such standards has a great impact on the choice of hardware components, as well as on the design of the internal and external interfaces of the target device. They must allow for clear separation between the service's interface and the underlying application's logic, as well as enable smooth interaction between the communicating parties over the network. Domain-specific applications, such as those related to image processing which is in the center of our interest, impose their own constraints, e.g. use of specialized sensors, specific data flow patterns or increased memory requirements. All above issues are discussed in the following sections.

2.1 Design requirements and assumptions

In this work by service we implicitly mean a software system designed to support machine-to-machine interaction over the network, i.e. a web service. Web services are application components exhibiting several unique properties. First, they are self-contained and self-describing. Second, they communicate using open protocols and support interoperability. Third, they are discoverable. When implementing web services in hardware, all above principles must be followed as in a regular software realization. To the desired services we assign the term *Hardware SOA*, or HSOA for short, to emphasize they are to be deployed on hardware according to the SOA architectural pattern [11].

For the sake of reconfigurability and computational efficiency required in our particular application context, FPGAs were chosen to implement the core logic of HSOA services. Therefore, an FPGA chip plays a managerial role in our hardware design. However, with FPGA alone an embedded application cannot be turned into a web service. First of all, a communication layer is required to admit bi-directional message passing between the device and the remote client applications. Besides, depending on the target functionality, various peripherals must be added to the design, together with their specific interfaces which will be used for internal control signal and data exchange with the central unit.

The aforementioned peripheral modules may for instance include a digital camera, a VGA display or some actuator, e.g. servo motor.

All implementation details of a web service must be made transparent to the outside world so that the clients only know what it offers and not how exactly it works. Traditionally, it is achieved by advertising public interface through *Web Service Description Language* (WSDL) files [28] that are put in special external registries, called service brokers, so that the clients can quickly discover at which URL a given functionality can be found. Then, a client application formats messages that carry requests for invocation of a chosen server method with appropriate arguments, as discovered in a registry. An XML-based protocol, SOAP,[1] is typically used for that purpose, which guarantees platform and language independency.

Mapping the above specifications to the hardware resources that can be integrated in a compact FPGA mainboard led us to an architectural concept in which:

- An independent Ethernet/WiFi module handles all aspects of network communication up to the transport layer of the TCP/IP stack, i.e. controls the physical link, formats and interprets data link and IP layer frames, and manages TCP/UDP client connections.
- FPGA runs the main program of the service and is responsible for controlling internal data flow path (image acquisition, processing and optional display in our case), formatting and interpreting the application layer data of the inbound/outbound messages (SOAP serialization/deserialization), executing client-requested server methods and managing communication with peripherals.
- If any computation-intensive data processing is required, it can be delegated to a separate set of programmable logic components of FPGA.

Figure 1 shows a schematic diagram of the proposed architecture. Currently, we simplify it by excluding the service broker. It is assumed that a WSDL file with service's interface

[1] Simple Object Access Protocol [24].

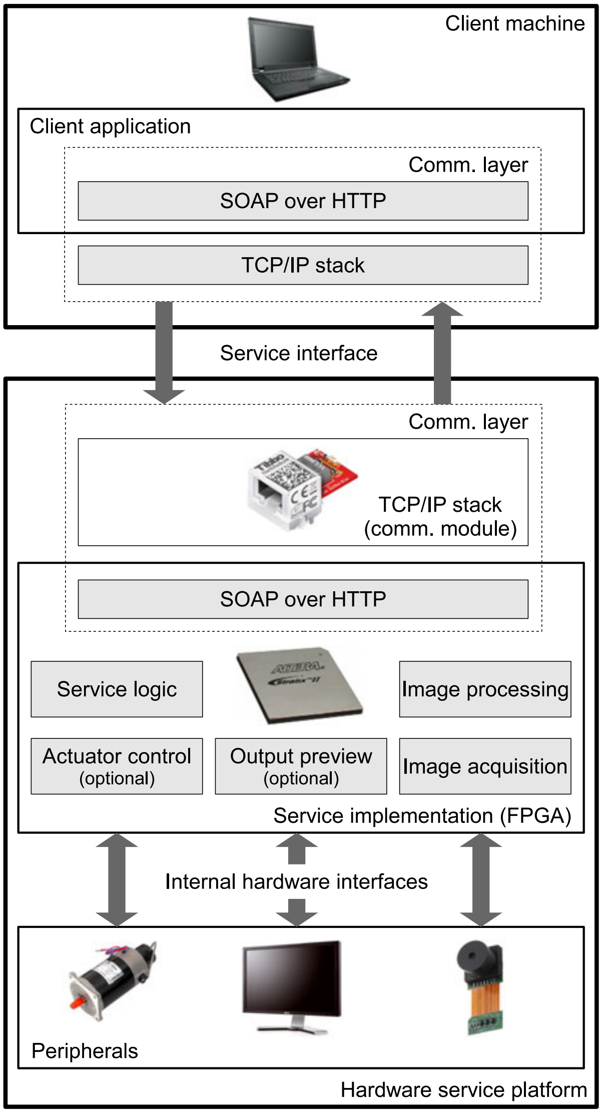


Fig. 1 General architecture of a HSOA image processing service

and web location can be advertised among the potential clients without recourse to external registry.

2.2 Hardware design overview

Altera family of FPGAs and the associated EDA software have been chosen for hardware platform development according to the general architecture outlined in Fig. 1. This initial decision was driven by our rich previous experience with this manufacturer's products, their popularity and the extensive technical support available. The FPGA families evaluated at the development stage varied from low-cost *Cyclone II* in the initial designs to more powerful *Stratix II* in the final implementation. We assumed a supervisory role of a *Nios II*

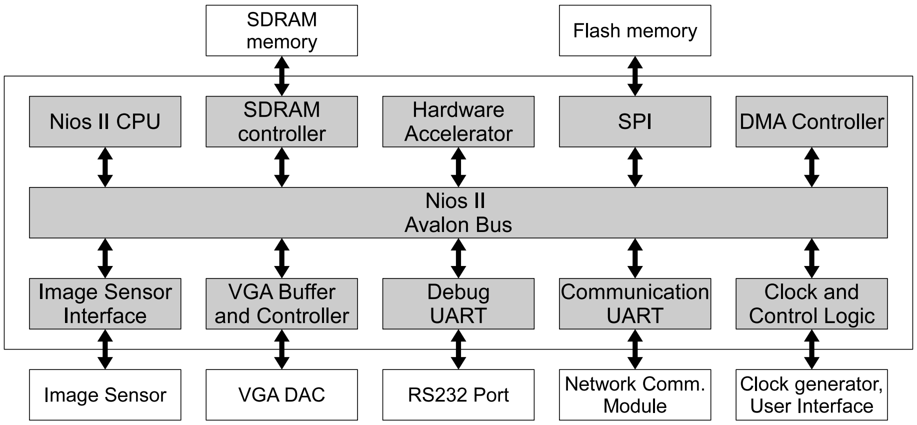


Fig. 2 Block diagram of the hardware platform customized for running specialized, network-enabled image processing services

software microcontroller. The main functionality of the service was coded as a sequential C++ program executed by the CPU. Additionally, computationally the most expensive image processing operations were offloaded to the hardware accelerator. In order to simplify and speed up the development, we generated it from a C-style specification using a C-to-hardware compiler, *Impulse C* [21]. Below we discuss in more detail the key features of the proposed hardware platform.

Figure 2 shows a block diagram of the device that has been customized to host specialized image processing services. Its physical realization is depicted in Fig. 3. The platform is based on the *Terasic's* TREX-S2-TMB board [25] with *Stratix II* EP2S60 FPGA and custom peripherals. The FPGA runs a *Nios II* software microcontroller. Interconnections between the CPU and each peripheral module are made with the *Avalon* bus.[2] The image processing accelerator is implemented using the aforementioned *Impulse C* compiler and the generated module is used as a peripheral in the *Nios II* system.

In the most basic version of the system, there is one block of operating memory with *Avalon* interface. This memory block is built upon two 32 MB SDRAM chips with 16-bit data bus, combined to work as a single double-size SDRAM block with 32-bit data bus. SDRAM modules operate at clock frequency of 100 MHz with CAS latency set to 2 cycles. The measured transfer performance in such a configuration reaches 332 Mbit/s in sequential, 32-bit word wide access using DMA channel. However, SDRAM is shared between the *Nios II* core and all hardware accelerator's processes, so it should not be overloaded with multiple queued transfers. Simultaneous accesses can be avoided by using the internal memory of FPGA whenever possible (see an example described in Sect. 4.2). Sharing the external memory between too many processes may lead to undesired system slowdown. We plan to address this limitation in the future hardware platform that will optionally support another SDRAM block and where both blocks can be used independently.

The VGA interface is added only for preview and debug purposes and will be disabled in the final version of the platform. The VGA driver consists of a block of SRAM memory,

[2]Default and the most basic type of interconnection in *Nios II* systems.

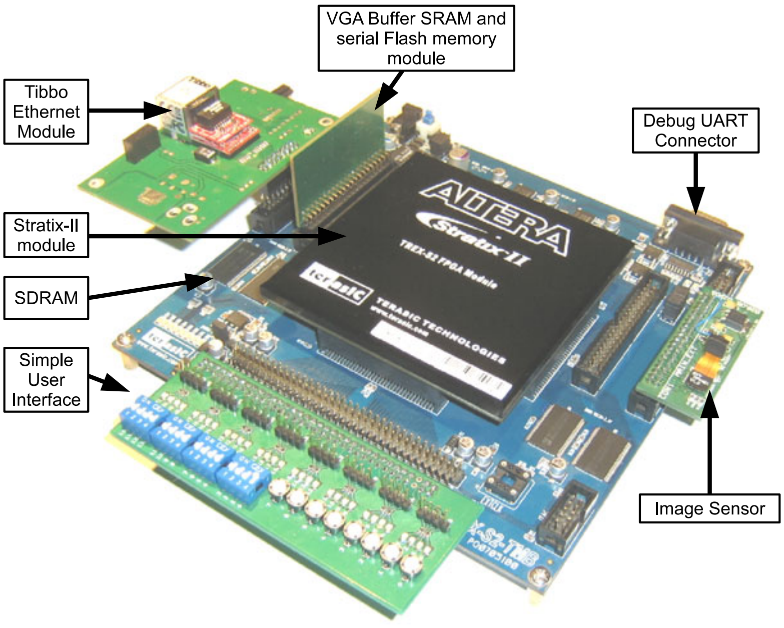


Fig. 3 Physical realization of the hardware platform from Fig. 2

acting as frame buffer, and a HDL wrapper that allows two-port memory access and drives video DAC (Digital-to-Analog Converter).

Network communication is handled by EM1206, a specialized programmable Ethernet module from *Tibbo Technology* [10]. It can operate using a simple asynchronous serial interface with optional handshaking and hence only one UART (Universal Asynchronous Receiver Transmitter) is required on the hardware side. The functionality of *Tibbo* modules can be easily tailored to the specific design using provided *Tibbo Basic* language. In the current version of the hardware platform the EM1206 module handles client connections over TCP transport while the higher-layer communication protocol is implemented in the *Nios II* program. The above approach to handling communication with the service clients dramatically reduces the development time (on both server and client side) and FPGA resource usage. Another advantage of using EM1206 is a possibility of expanding its functionality to WiFi-based communication.[3]

In order to ensure flexible, non-volatile FPGA configuration backup and *Nios II* program storage, a few auxiliary memories are used in the system. The FPGA configuration is stored in EPCS16 Flash memory and loaded to FPGA each time the system is powered on. Once the FPGA has been configured, a sequence of reset signals is applied and the *Nios II* processor starts executing its program from the reset vector. The latter is located in an internal block of FPGA memory which is configured as ROM. Since that memory has small capacity, it stores only a simple bootloader program. The *Nios II* program is stored in an external Flash memory. Prior to its execution it is copied to SDRAM by the bootloader.

[3]It can be done using *Tibbo*'s GA1000 WiFi SPI expansion module.

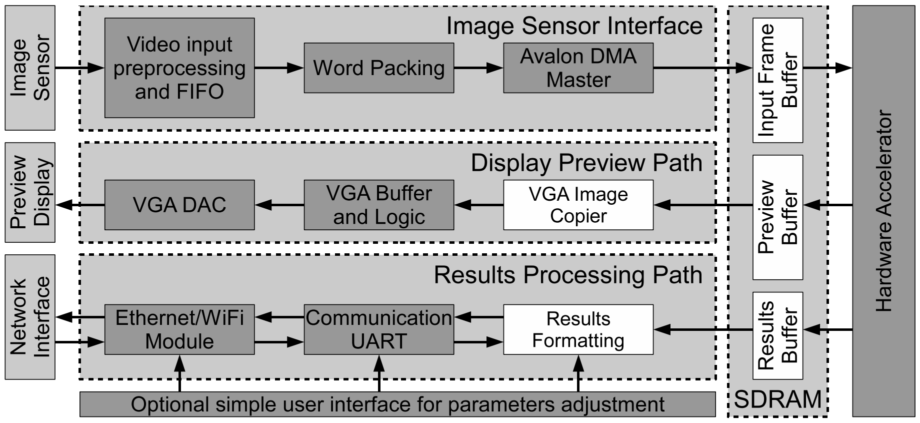


Fig. 4 Image data flow diagram. *Dark gray* fields denote hardware processing modules, and *white* denote software components

An additional feature of the development version of our platform is that it allows one to modify the selected internal hardware and software parameters of the service being run using a simple user interface attached to the platform's mainboard, as shown in Fig. 2. It consists of eight programmable push-buttons and four 4-bit DIP switches.

2.3 Image processing pipeline from the hardware perspective

Figure 4 shows the data processing path. A PPV401 image sensor from *PixelPlus* is responsible for video data acquisition. The sensor is connected to the system using custom-made hardware drivers (Image Sensor Interface block in Fig. 4). In the first step, the driver reads video data from the sensor, which by default outputs pixels in YCbCr422 format, and extracts gray-scale information. It is also possible to synthesize a color-aware version of this block.

In order to increase DMA transfer efficiency, image data is packed into a contiguous stream of 32-bit words using FIFO buffers with additional logic. Packed data is sent to the predefined SDRAM locations (Input Frame Buffer in Fig. 4) with *Avalon DMA Master* module. Using additional synchronization signals, software can request image acquisition to Input Frame Buffer and determine end of the buffer filling operation. Once this buffer has been filled with new data, it can be used by the Hardware Accelerator module, if present. There is also a possibility of generating preview image for accelerator development and debugging purposes. The preview functionality is implemented within the Display Preview Path shown in Fig. 4.

Depending on its format and size, output from the Hardware Accelerator is either stored in shared operating memory (e.g. fragments of the processed images) or received by the *NIOS II* program via buffered streams (e.g. images processed in a pixel-by-pixel manner), signals or registers (e.g. single numeric values). Further processing, if necessary, is done sequentially within the main program of the service. It is aimed at producing data of a format and meaning that is defined by the interface of the operation referred to in the currently processed client request. The obtained results are ultimately packed into an XML envelope that is passed to the *Tibbo* communication module. The latter emits the resulting XML as a response of the service.

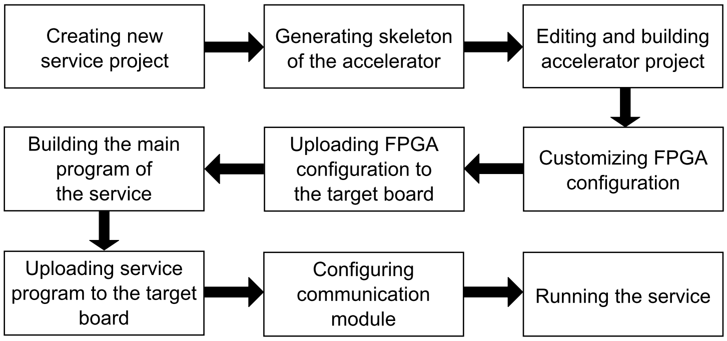

Creating new service project
Generating skeleton of the accelerator
Editing and building accelerator project
Customizing FPGA configuration
Uploading FPGA configuration to the target board
Building the main program of the service
Uploading service program to the target board
Configuring communication module
Running the service

Fig. 5 Main steps of the HSOA service development cycle

3 Software development tools

The proposed embedded service platform requires several independent EDA tools to prepare the constituent hardware components for operation and integrate them together. The tasks involved in this process include among others:

– programming an FPGA-based co-processor to handle computationally the most demanding image processing operations,
– implementing the service's main program that will be executed on an embedded processor,
– building the communication interface of the service based on the programmable Ethernet/WiFi module,
– deploying the service.

All these tasks, when performed in an unstructured way, introduce large burden and in effect decrease the productivity of the build process. To manage the above design, implementation and installation steps, as well as the heterogeneity of the used tools, languages and techniques, appropriate development software has been introduced. It is available as a *HSOA Service Builder* suite.

The rest of this section briefly discusses the usage of the suite. Specifically, in Sect. 3.1 the overall service development flow is described. In Sect. 3.2 we focus on the *Impulse C* language and the corresponding visual designer tool that, compared to the traditional HDL programming approach, dramatically simplify the implementation of parallel algorithms to be executed in FPGAs. Section 3.3 outlines the service's core logic programming in C++ language. Finally, in Sect. 3.4 we describe how to deploy an already built service.

3.1 Service development flow

The proposed methodology of building hardware services for intelligent image processing has been illustrated in the flow diagram in Fig. 5. The service designer starts with creating a new hardware service project or opens an existing one. The project settings are stored in an XML file. These settings include among others the name of the service, its root directory, where all project-related files are kept, endpoint URL and (optionally) a path to the associated FPGA accelerator project file, which will be discussed in more detail in Sect. 3.2. In addition, the designer is supposed to provide the details of the operations exposed by the service, including their names, as well as arguments, return values and their datatypes.

In the next step the designer can decide upon whether or not to offload part of the service's functionality to the dedicated block of programmable logic for faster computation. Note that in the case of video processing, i.e. moving target tracking, it will often condition real-time execution. From the main window of the *HSOA Service Builder* application the *FPGA Accelerator Designer* tool is first invoked to quickly define the architecture of the accelerator. Then, the generated project skeleton can be edited in the *Impulse CoDeveloper* environment so as to define the behavior of the desired hardware module. The HDL code of the accelerator is automatically synthesized from the C language specification, so essentially no hardware programming skills are required.

We provide a basic FPGA configuration that contains *Nios II* processor core, the controllers of necessary peripherals and the bootloader program. The just created accelerator is treated as a standalone hardware module that can communicate with the *Nios II* service program through the *Avalon* bus. To enable this communication, one has to customize the hardware design using standard *Altera* tools: *Quartus II* and *SOPC Builder*. Again, they can be called directly from the *HSOA Service Builder* window. The previously generated accelerator is automatically recognized as a custom IP core. The built configuration is then uploaded to the flash memory of the target board using *Quartus II Programmer* tool.

The key step in the development process involves building the main program of the service. To do that, one edits a template *Nios II IDE* project by adding service-specific C++ code. In this code the developer should define how the incoming video stream will be processed, as well as provide the implementation of the service's public interface. Moreover, methods of XML serialization and deserialization are provided so that the client requests can be interpreted and the responses containing processing results properly formatted. If the FPGA accelerator is used, from the program's main loop appropriate *Impulse C* API calls can be made to exchange data with it. Once the logic of the service has been defined, the project has to be compiled and the resulting binary program uploaded to the data flash memory on the target board.

Finally, the designed electronic circuit must be turned into a network-enabled device. It is done by invoking the *Tibbo IDE* software from the *HSOA Service Builder* window and again editing a template project using vendor-specific programming language, called *Tibbo BASIC*. The provided *Tibbo* program requires minimal customization. It manages data exchange with the clients at the TCP/IP layer using sockets API. However, it does not interpret client requests. These are relayed via UART to the *Nios II* program for interpretation. Similarly, server responses are sent back to the *Tibbo* program and further relayed to the original server method's callers. The compiled project is uploaded to the Ethernet/WiFi module as a binary file.

Upon completion of the above steps the device is ready for operation. To advertise the service in the web and enable fast implementation of client programs in conformance to the Web Service standards, a *Web Service Description Language* (WSDL) [28] file can be generated automatically from the *HSOA Service Builder* window. The following sections give a more specific description of each development step and in Sect. 4 concrete implementations of two example image processing services are discussed.

3.2 Building hardware accelerators

Offloading execution of the computationally expensive operations to a dedicated hardware unit for performance speedup is a common design pattern [1, 13]. In embedded processor based architectures the role of such specialized co-processors is particularly important as soft-core CPUs are known to be relatively slow. For instance, the *Nios II* core used in our experiments can run at the maximum clock speed of only 100 MHz which is 20–30× lower than the clock rates of modern microprocessors. In our context what requires speedup are various operations that scan and/or modify the contents of the images.

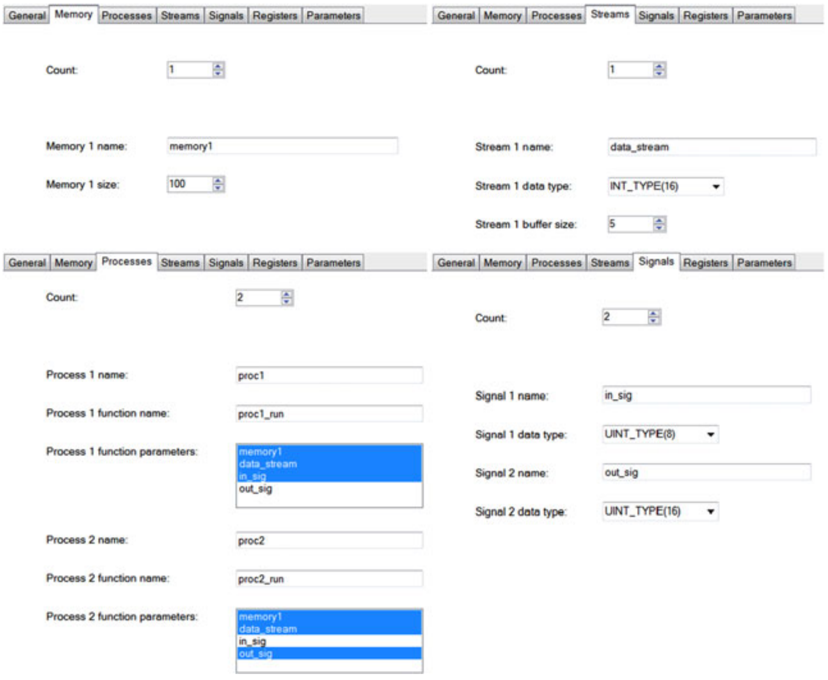
To implement algorithms that exploit the massively parallel architecture of FPGAs, we employ the *Impulse CoDeveloper* software from *Impulse Accelerated Technologies* and an additional utility application that serves as its graphical front-end. The former software package is based on the familiar C programming language that has been extended by a set of additional constructs providing convenient abstractions of coarse- and fine-grained parallelism [21]. It allows the developer to quickly code parallel algorithms in a form of C functions and simulate their behavior in a software test-bench. To generate functionally equivalent hardware modules, HDL code can be automatically synthesized from the C specification.

One problem with the *Impulse C* compiler (as well as other C-to-hardware compilers of this kind), is that it does not support dynamic data structures. As a result, the processes and the components used for their interconnection must be explicitly declared at compile time. This, in turn, produces large code that is difficult to write from scratch, even if multiple processes provide exactly the same functionality, but on different data. *FPGA Accelerator Designer* utility helps resolve this problem. It offers a convenient GUI that allows the service designer to indicate the number of required components of each type: memory blocks, processes, streams, signals, registers and constant parameters, as well as their names, mutual relations and supported data types. When the design is ready, the user simply clicks a button to generate the entire *Impulse C* project with all necessary declarations and function skeletons in place. The remaining work to be done only involves filling in bodies of these skeleton functions according to the intended accelerator's behavior, followed by HDL synthesis. This way, even very complex accelerator topologies that span several thousand lines of C code, can be programmed in a reasonable time.

An example of how specific settings made in *FPGA Accelerator Designer's* GUI map to the generated *Impulse C* code is given in Fig. 6. Only the top-level configuration function with necessary declarations and process initialization is shown. Note that in the *Processes* tab page the user can define the actual links between the parallel processes by selecting each process function's parameters from among the components previously added.

Finally, the generated accelerator module must be connected to the *Nios II* system. To do this, the designer is supposed to use *SOPC Builder*, a standard *Altera's* tool for software-hardware components integration. The tool can be invoked from *Quartus* software that is in turn started with the associated project upon clicking an appropriate button in the *HSOA Service Builder* application window. The developer must then connect all of the accelerator module's signals to the *Nios II Avalon* bus. Upon generating the system in *SOPC Builder* the module can be referenced in the service program's code by its uniquely assigned base address.

A well-weighted architecture of the *FPGA Accelerator Designer* application is worth noticing. It supports accelerator topology templates with a generic template allowing all types of hardware components to be connected in any semantically acceptable way. In this case the service designer takes responsibility for how sensible the resulting design might be in a concrete application context. More specialized templates, a few of which are provided in the current version of the *HSOA Service Builder* suite, impose various constraints on the components (and the related GUI controls) the designer may include in the topology and the means of tying them together. This design pattern prevents generating erroneous accelerator code and allows easy addition of new templates.



Memory tab:

Count: 1

Memory 1 name: memory1

Memory 1 size: 100

Streams tab:

Count: 1

Stream 1 name: data_stream

Stream 1 data type: INT_TYPE(16)

Stream 1 buffer size: 5

Processes tab:

Count: 2

Process 1 name: proc1

Process 1 function name: proc1_run

Process 1 function parameters: memory1, data_stream, in_sig, out_sig

Process 2 name: proc2

Process 2 function name: proc2_run

Process 2 function parameters: memory1, data_stream, in_sig, out_sig

Signals tab:

Count: 2

Signal 1 name: in_sig

Signal 1 data type: UINT_TYPE(8)

Signal 2 name: out_sig

Signal 2 data type: UINT_TYPE(16)

```
#include "co.h"
#include "TestAccelerator.h"
#include "TestAccelerator_parameters.h"

void module_config(void *arg)
{
    // Memory declarations
    co_memory memory1;

    // Stream declarations
    co_stream data_stream;

    // Signal declarations
    co_signal in_sig;
    co_signal out_sig;

    // Process declarations
    co_process proc1;
    co_process proc2;

    // Initialization
    memory1 = co_memory_create("memory1","heap0",102400);

    data_stream = co_stream_create("data_stream",INT_TYPE(16),5);

    in_sig = co_signal_create("in_sig",UINT_TYPE(8));
    out_sig = co_signal_create("out_sig",UINT_TYPE(16));

    proc1 = co_process_create("proc1",(co_function)proc1_run,3,memory1,data_stream,in_sig);
    proc2 = co_process_create("proc2",(co_function)proc2_run,2,memory1,data_stream,out_sig);

    // Process-to-hardware assignment function calls
    co_process_config(proc1,co_loc,"pe0");
    co_process_config(proc2,co_loc,"pe0");
}

co_architecture co_initialize(int param)
{
    return co_architecture_create("TestAccelerator","altera_nios2",module_config,NULL);
}
```

Fig. 6 Example usage of the *FPGA Accelerator Designer* tool. At the top a user-defined accelerator's topology is enforced by the appropriate state of the GUI controls. Beneath, a listing from the generated *Impulse C* project code shows the top-level configuration function reflecting the visual designer's specification

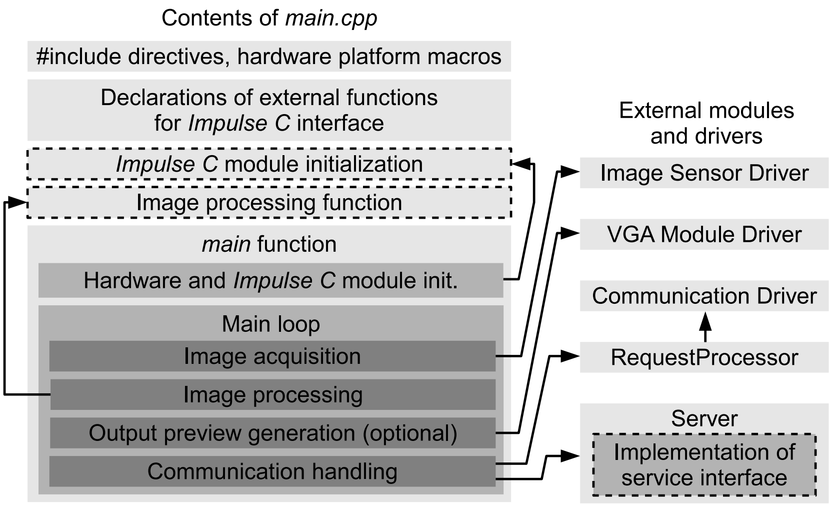

Contents of *main.cpp*
#include directives, hardware platform macros
Declarations of external functions for *Impulse C* interface
Impulse C module initialization
Image processing function
main function
Hardware and *Impulse C* module init.
Main loop
Image acquisition
Image processing
Output preview generation (optional)
Communication handling
External modules and drivers
Image Sensor Driver
VGA Module Driver
Communication Driver
RequestProcessor
Server
Implementation of service interface

Fig. 7 Layout of the source code defining the *Nios II* program. The developer is supposed to modify the parts framed with *dashed line* while the remaining parts are provided as a template or auto-generated at the time of service project creation. *Arrows* represent references to sections defined elsewhere in the code

3.3 Customizing the main program of the service

Services developed using the proposed hardware platform run under control of a sequential C++ program that is executed by the *Nios II* processor. For ease of development, a template of this program has been prepared and each new hardware service project created in *HSOA Service Builder* application automatically makes a copy of this template. The primary tasks of the service's designer at this point include:

– specifying the operations to be exposed by the service,
– customizing the service program code using *Nios II IDE* software,
– compiling and uploading the service program to the target board.

As mentioned in Sect. 3.1, specification of the service's public interface is given by the developer as early as at the time of creating the service project. As a result, source files with appropriate method declarations and empty definitions are generated. These files are also automatically added to the list of source files constituting the main program of the service. Further tasks involve customizing the behavior of this program by editing a *Nios II IDE* template project which comes with an associated system library. The latter provides a hardware abstraction layer to the service program.

Figure 7 schematically shows the layout of the service program's source code where `main.cpp` is the top-level file defining the entry point. Bulk of this code is either provided as a template or auto-generated at the time of project creation. As the template service only performs a dummy input-output image copy operation and uses no hardware acceleration, the programmer is expected to fill in the missing fragments with own code, depending on the target service's functionality and the hardware accelerator's interface. Similarly, implementation of the service's public interface must be provided. Major code sections requiring modification by the developer are highlighted in Fig. 7.

Regarding interaction with the FPGA accelerator, if needed, the most important aspect is the *Impulse C* module initialization. It requires appropriate declarations of the internal hard-

```
// Declaration of global variables
static volatile co_signal sig1, sig2;
static volatile co_memory mem;

// Initialization in the ICInit function
mem = co_memory_create("name","heap0",SIZE,malloc);
sig1 = co_signal_create("name1");
sig2 = co_signal_create("name2");


// Writing data to shared memory
...

// Sending a signal to the accelerator
co_signal_post(sig1,val1);

// Possibly doing some extra processing
...

// Receiving a signal from the accelerator
co_signal_wait(sig2,&var2);

// Reading data from shared memory
...
```

```
// Declaration of global variables
static volatile co_stream str1, str2;

// Initialization in the ICInit function
str1 = co_stream_create("name1",datatype1,npackets1);
str2 = co_stream_create("name2",datatype2,npackets2);


// Opening streams
co_stream_open(str1,mode1,datatype1);
co_stream_open(str2,mode2,datatype2);

// Sending data to the accelerator
co_stream_write(str1,buf1,nbytes1);

// Receiving data from the accelerator
co_stream_read(str2,buf2,nbytes2);

// Closing streams
co_stream_close(str1);
co_stream_close(str2);
```

Fig. 8 Two methods of communication between the main service program and the hardware accelerator: via shared memory and signal-based synchronization (*left* listing) and via buffered streams (*right* listing). Only the software-side code is shown and the declarations of some variables are omitted

ware components interfaced on the software side, such as shared memory blocks, streams or registers, as well as their instantiation based on the addresses generated by the *SOPC Builder* and defined in the system library project. In addition, the top-level image processing function, called in every iteration of the program's main loop, has to be augmented with additional code where the accelerator module is called and the optional image pre-/post-processing is done.

If FPGA algorithm acceleration is enabled, the actual structure of the code will depend on the type of communication between the software program and the *Impulse C* module. For instance, a DMA image transfer pattern necessitates the use of *Impulse C's* shared memory interface and signal-based synchronization. It is appropriate for all operations that require random access to the pixels of an image that is already available in memory as a whole. On the other hand, pixel-by-pixel image processing, adequate for many simple operations, such as digital filtering, makes it sufficient to use functions from the *Impulse C* stream API. These two communication patterns are depicted in Fig. 8. Note that the extra processing done on the software side between the signal post and wait calls must be cheap enough to avoid deadlock when the return signal from the accelerator is not captured on time. For details on how to use all communication mechanisms offered by the *Impulse C* language, refer to [21].

As far as service orientation is concerned, we regard HTTP and XML protocols as a basic medium for encoding clients' requests and service's responses. From the developer's perspective all aspects of network communication up to the transport layer of the TCP/IP stack are handled by the *Tibbo's* EM1206 module which acts as a simple server and offers a convenient sockets API. This server project is provided as a template and its copy can be customized upon invoking the *Tibbo IDE* tool from the *HSOA Service Builder* window. When the device is powered on, the server opens a socket and waits for the incoming client connection requests on a predefined port. If such a request is received, the server only forwards (via serial interface) the HTTP/XML message content to the *Nios II* program where deserialization of arguments and the appropriate service method's invocation is made. The service's response, once XML-serialized, follows exactly the same way but in the opposite direction. Although multiple simultaneous client connections can be established, currently only one request at a time is processed. As a result, other clients that simultaneously request access to the web resource immediately receive a HTTP 503 "service unavailable" notification.

3.4 Deployment

As mentioned in Sect. 2.3, new service's configuration is accompanied with a specialized bootloader program which is uploaded to the FPGA together with its configuration. The bootloader's objective is to copy the service's main program code from the serial Flash chip to SDRAM memory and then start its execution. It can also perform some additional tasks, such as:

– transferring *Nios II* executable code from Debug UART to SDRAM via Xmodem protocol, and then executing it (leaving serial Flash memory contents unchanged, which is a convenient feature for development purposes),
– uploading *Nios II* executable code to Flash using Debug UART and Xmodem protocol.

The actual boot mode selection can be made using one of the DIP switches available in the user interface extension board mounted to the hardware platform's mainboard.

To ensure proper FPGA configuration and smooth start of the service, the developer should power up the platform, program the EPCS16 Flash memory with FPGA configuration, and then upload the *Nios II* main executable code to serial Flash using the above described bootloader program. The actual bootloader's operation mode can be selected by providing correct logic levels on its control inputs during power-on or system reset.

Finally, to facilitate client-side code creation, our tool allows the service provider to generate a WSDL file which contains the standardized specification of the public interface to the web service based on the information provided at the time of project creation (refer to Sect. 3.1 for details). For this purpose we employ the open source *gSOAP* toolkit [26] which is included in the distribution of the *HSOA Service Builder* suite. WSDL generation is triggered upon clicking an appropriate button in the main window of the managing application. The resulting web service description file can be further advertised among the potential service clients.

4 Use cases

To demonstrate the advantages of the proposed methodology of building embedded services, we outline below two example applications that were developed using the *HSOA Service Builder* suite. In Sect. 4.1 a moving object detection service is introduced. Section 4.2 describes an algorithmically more complex design – trainable object classifier.

4.1 A simple example: motion detector

A simple implementation of motion detector has been chosen to demonstrate the usage of our hardware/software platform in the HSOA service development process. This detector operates by subtracting the current input video frame from the background frame, followed by appropriate thresholding. In the resulting motion map a connected component analysis (CCA) is run to identify consistent moving blobs. The background frame is periodically updated. In the demo implementation the service is available at a pre-defined IP address and port number and a WSDL file with the interface description is placed in a public repository.

The service exposes several operations that help the client discover whether and, if so, where and when exactly any motion was detected in a scene observed by the device-mounted camera. We have built a simple client application where the user can specify the appropriately quantized past time interval of interest along with the portion of the scene where the

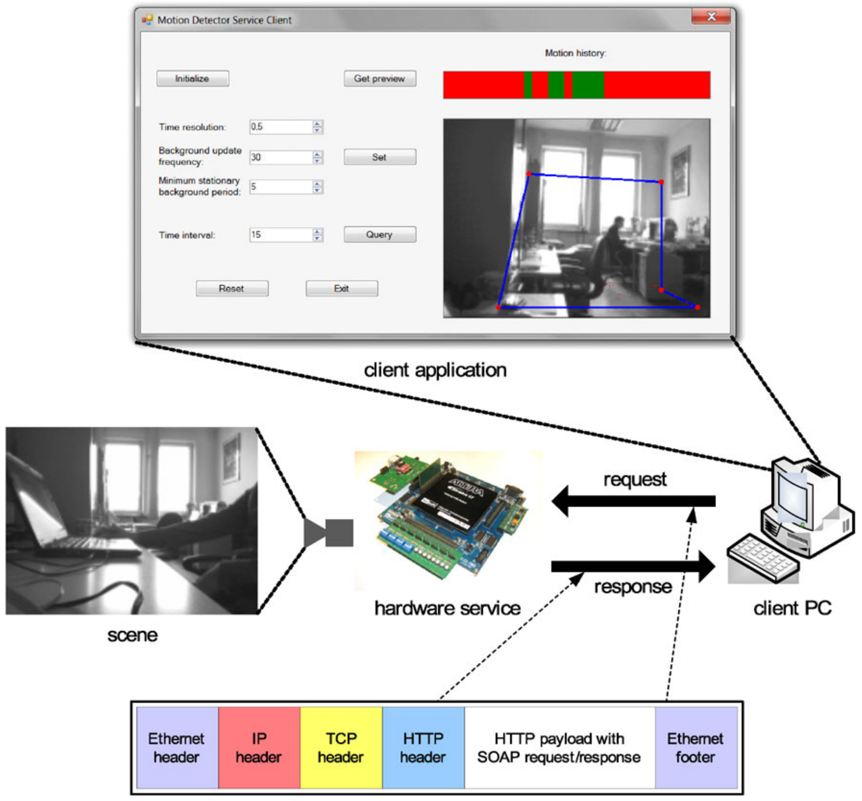


Fig. 9 Realization of the motion detection service

presence of moving objects should be checked. Upon reception of the motion query on the server side the device responds with a byte array encoding the detected objects' positions in the image over the requested period. Figure 9 shows how motion detection service works.

To make the detector insensitive to noise, which would normally cause many accidental pixels to be classified as motion pixels, we convolve each difference image with a sum filter and threshold the block-aggregated values. This simple algorithm has been efficiently implemented as an FPGA-based accelerator which we built using the tools presented in Sect. 3.2. Within the same accelerator also the aforementioned connected component analysis is run. However, in this case no significant computation speedup is achieved due to the recursive nature of this algorithm and the non-contiguous access to the motion map that must be entirely stored in shared memory. In Fig. 10 we illustrate the idea of accelerated smoothed difference image computation.

Assuming $n \times n$ pixel block[4] the algorithm proceeds as follows. First, $n-1$ beginning rows of each input image are read and stored in $n-1$ out of n on-chip memory buffers and the first $n/2$ rows of the output image are filled with zeros (no motion). Then, the subsequent row of pixels from each input image are read into the corresponding n-th buffer. Regarding the output image, the first $n/2$ pixels of the new row are again set to zero. Value of the next

[4]In general, n must be an odd number.

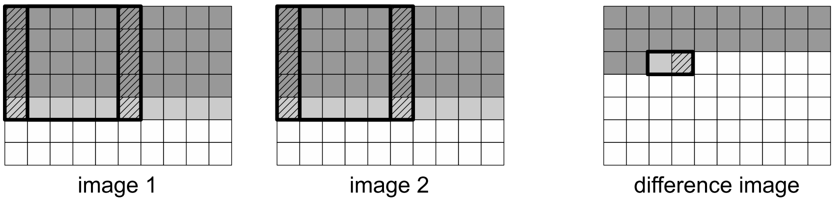


Fig. 10 The idea of the block-wise difference image computation with block size equal to 5 pixels

output pixel (output image's cell at position (3, 3) in Fig. 10) is determined by computing the modulus of the sum of the input pixel differences within the top-left corner $n \times n$ pixel block. As the values of pixels from this block are stored in the on-chip memory, the entire sum can be calculated in a single clock cycle. The modulus is compared to a pre-defined threshold to determine if the block can be classified as a motion block.

The following output pixel (the thick-framed and strip-patterned output image cell in Fig. 10) does not require computation of the full sum of differences. To obtain the motion flag for the corresponding $n \times n$ block it is sufficient to subtract from the old sum-of-differences value the sum of pixel differences within the block's left-most column, shift the block to the right by one, add the sum of pixel differences within the block's right-most column and then apply threshold. Again, as all required pixel values are already stored in the independent buffers of the FPGA internal memory and thanks to the loop pipelining offered by the *Impulse C* compiler, each new output value computation requires a single clock cycle to complete. It should be noted that full sum must be again calculated only at the beginning of each new row.

Once the entire region of the input frames has been scanned, the number of the above-threshold output values is counted and the resulting value is compared against another threshold which determines whether or not the scene can be thought of as containing motion. If, so the aforementioned connected component analysis is run on the obtained motion map so as to determine the locations of the individual moving objects. In the practical implementation, the average processing speed of this part of the system, including CCA and optional output image generation, is estimated at 59 clock periods per pixel. It is sufficient to achieve frame rates of 11.1 fps for QVGA resolution or 5.4 fps for VGA using the main clock signal at the frequency of 100 MHz.

On the *Nios II* program side the communication with the accelerator module is handled using a pair of signals according to the scheme depicted in Fig. 8. From the main program's perspective the outbound signal notifies the accelerator of the new video frame being available in SDRAM. The inbound signal contains the number of objects detected in the current frame. If it is non-zero, saved objects' positions are retrieved from the shared memory. A queue containing motion history over a reasonably long time period is maintained in the service's main program. After each (adjustable) quantization period, e.g. 1 second, the positions of up to n objects detected within this period are pushed into the queue, where n is configurable.

The video processing path runs independently of the code responsible for capturing client requests. If such an even occurs, the current state of the motion history queue is XML-serialized according to the *Web Services* specification. The appropriately formatted SOAP response is sent back to the communication module via UART and then to the client over the network. A simple C++ client application, thumbnail of which is shown in Fig. 9, receives the motion history and displays it overlapped on a miniature of the scene image. This miniature can be obtained upon calling another operation of the same web service. Additionally, a rectangular panel indicates the time points within the requested interval when the objects appeared in the scene.

The implementation of motion detector service requires several software tools to build the configurations and programs which reside in the programmable logic. A template for *Impulse C* processing core was generated using the *FPGA Accelerator Designer* utility and then filled with application-specific code—in this case the motion detection algorithm described above (3.2). A similar procedure simplified creation of the firmware for embedded *Nios II* microprocessor. The core of the program was auto-generated according to given interfaces description and then, using the *Nios II* system, only the main server program logic was added (3.2). Other tasks, which were automated in the motion detector application include:

– Generating *Nios II* microcontroller system with standard peripherals including the previously designed hardware accelerator,
– Creating hardware description for the system using Verilog HDL,
– Generating code template for the *Tibbo* communication module,
– Compiling and uploading the configuration and firmware files to the appropriate devices.

It should be noted that a large amount of code (especially C++ and HDL templates) is reusable, i.e. common to many hardware services deployed in the proposed device.

4.2 An advanced example: trainable object classifier

While simple image filters, such as the one presented in Sect. 4.1, can be executed sufficiently fast on sequential machines, there are algorithms that give computational speedup by even two orders of magnitude if implemented on parallel architectures. To demonstrate the usefulness of the proposed embedded services design methodology in implementing such algorithms, we have built a trainable object classification service.

The underlying classifier is based on the so-called *Kernel Regression Trees* introduced in [22]. Such trees are trained from pairs of images labeled "same" or "different", depending on whether or not they depict the same object. The training procedure is executed on a standard PC. Abstracting from the details, which can be found in the above cited reference, the learning algorithm builds a tree-like hierarchy of fuzzy decision stumps. Each stump, called kernel function, is adjusted such that it possibly best separates the pairs containing images that are similar with respect to a selected image descriptor, e.g. local gradient orientation histogram, from the pairs of images that significantly differ with respect to this descriptor.

The specific property of *Kernel Regression Trees* is that due to the local discrimination ambiguity the image pair passing through a given node may simultaneously follow paths leading to both subtrees. Formally, such a pair, $\mathbf{x}$, does not strictly belong to any child node of a given node. Instead, it is assigned a degree of membership to each node D, $\mu_D(\mathbf{x}) \in [0,1]$. The degree of membership to the root node, $\mu_{Root}(\mathbf{x})$, is by default set to unity. The degree of membership to the child nodes D_L and D_R of a given node D is defined recursively:

$$\begin{aligned} \mu_{D_L}(\mathbf{x}) &= \mu_D(\mathbf{x}) f_{D,k,\boldsymbol{\Theta}}(\mathbf{x}), \\ \mu_{D_R}(\mathbf{x}) &= \mu_D(\mathbf{x})(1 - f_{D,k,\boldsymbol{\Theta}}(\mathbf{x})), \end{aligned} \tag{1}$$

where $\mu_{D_L}(\mathbf{x})$ and $\mu_{D_R}(\mathbf{x})$ denote the degrees of membership of the pair $\mathbf{x}$ to the left/right child node of node D, $f_{D,k,\boldsymbol{\Theta}}(\mathbf{x})$ is the response of the local kernel function, k is the index of the selected image descriptor underlying this function and $\boldsymbol{\Theta}$ denotes a vector of its parameters.

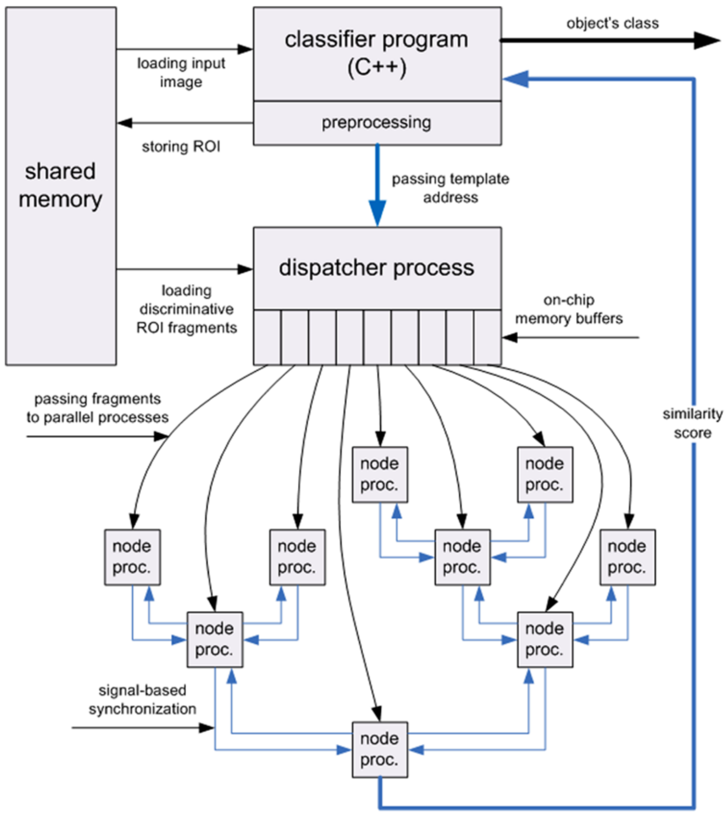


Fig. 11 Architecture of the object classification system implementing a parallelized version of a *Kernel Regression Tree* classifier

The ultimate response of the tree expresses overall similarity between two images. For a given input pair it is obtained according to (2) by summing the products of the terminal node labels, L_{D_i}, which give the local estimation of the output along each path, and the probabilities $\mu_{D_i}(\mathbf{x})$ of reaching these terminal nodes by the input pair:

$$\hat{y} = \frac{\sum_{i \in \text{leaves}} \mu_{D_i}(\mathbf{x}) L_{D_i}}{\sum_{i \in \text{leaves}} \mu_{D_i}(\mathbf{x})} \quad . \tag{2}$$

To classify a new image, it has to be compared to the prototypes of each class. The prototype maximizing pairwise similarity determines the unknown object's category. Due to the fuzziness introduced, the average number of local image comparisons and kernel evaluations grows exponentially with the tree's depth. With an observation that the increase in the number of classes leads to more complex trees, it can be concluded that a sequential implementation of the classifier is not scalable and hence unsuitable for real-time execution. However, *Kernel Regression Trees* have a nice property of being highly parallelizable which we have exploited implementing the classifier in FPGA.

Standard steps were followed when developing the service using the *HSOA Service Builder* suite. The main difference between this service and the previously discussed motion detector lies in the complexity of the hardware accelerator and the C++ code that manages a communication with it. The system's architecture is shown in Fig. 11.

A shared SDRAM memory block acts as a storage for a copy of the original input image to be analyzed, all class prototypes and various auxiliary data structures utilized by the software-side program code. When a new image is to be recognized, this program sends a triggering signal to the accelerator's module. It contains the address in memory where the first prototype image is located. The dispatcher process, which knows the trained classifier parameters, reads appropriate portions of both input images to the local on-chip memory buffers. Then, each node process receives the corresponding portion from the dispatcher, evaluates the underlying image descriptor, calculates the local distance, determines the response of the kernel function and upon completion waits for a signal sent by the parent node process. This signal contains the parent's membership degree and kernel response for the input image pair. When received, the current node calculates its own membership degree, posts an update signal to both successor nodes, if present, and then blocks. This enables correct recursive computation of the membership degree of an input example to the terminal nodes according to (1).

Once a given component of the sum in (2) has been calculated by the corresponding terminal node process, a signal is sent to the parent node process. A cascade of such asynchronous signals reactivate the blocked processes which allows the ultimate tree's response to be computed on the way back from the recursion. The operation of a single non-terminal node process is shown in Algorithm 1.

Algorithm 1 Single cycle of operation of a non-terminal node process participating in a distributed computation of the *Kernel Regression Tree's* response for an input image pair according to equations (1) and (2). $\hat{y}_{D_L}$ and $\hat{y}_{D_R}$ denote the output of subtrees rooted at the child nodes of node D and $\hat{y}_D$ is the output of a subtree rooted at D.

```
1: receive portion of the input images $\mathbf{x}$ via a stream from the dispatcher process
2: calculate $f_{D,k(D),\mathbf{\Theta}(D)}(\mathbf{x})$
3: wait for the parent node process' signal with $f_{P,k(P),\mathbf{\Theta}(P)}(\mathbf{x})$ and $\mu_P(\mathbf{x})$
4: calculate $\mu_D(\mathbf{x})$
5: post a signal with $f_{D,k(D),\mathbf{\Theta}(D)}(\mathbf{x})$ and $\mu_D(\mathbf{x})$ to the child node processes
6: wait for child node processes' signals with $\hat{y}_{D_L}$ and $\hat{y}_{D_R}$
7: calculate $\hat{y}_D = \hat{y}_{D_L} + \hat{y}_{D_R}$
8: post a signal with $\hat{y}_D$ to the parent node process
```

The root node process finally sends a signal with the response to the software-side classifier's program. This in turn triggers estimation of the similarity of an input image to the next prototype. When all prototypes have already been compared to, the most similar ones are determined according to a *k Nearest Neighbors* rule. Note that in the above distributed tree evaluation scheme the most intensive computations, i.e. computation of the image descriptor and local distance followed by kernel evaluation (line 2 in Algorithm 1), are done in parallel.

Regarding the test implementation, we trained a sample classifier to recognize 12 car models from the dataset used in [22]. The resulting tree consists of 17 nodes, as shown in Fig. 12. The *Impulse C* code of the accelerator module implementing the distributed tree evaluation algorithm contains over 2000 lines of code, most of which were automatically generated using the *FPGA Accelerator Designer* tool. In order to simplify this process even more, a specific design template was defined so as to restrict the types of hardware components and their relationships to just those required by a tree topology.

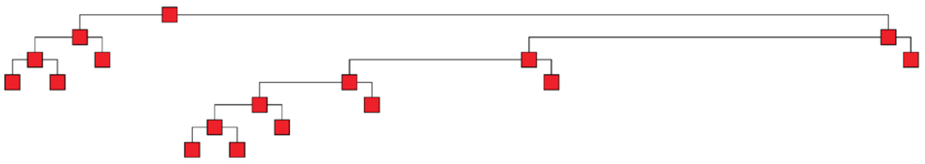


Fig. 12 Topology of the test *Kernel Regression Tree* classifier

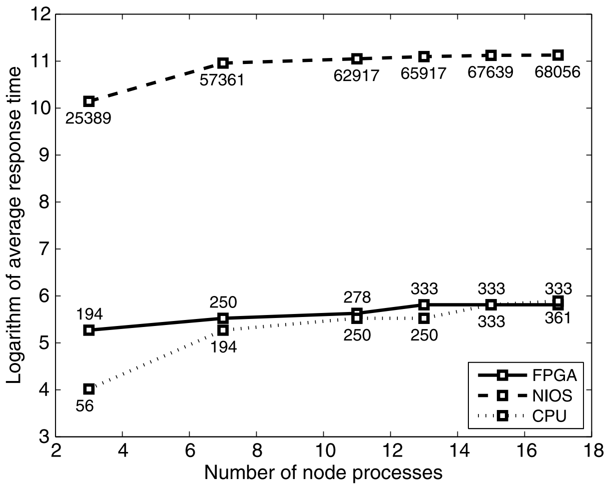


Fig. 13 Comparison of the average response time of the *Kernel Regression Tree* from Fig. 12 using: a parallel FPGA-based co-processor controlled by the *NIOS II* program, both working at 100 MHz, and the sequential implementations on: (1) a PC equipped with 2-core, 32-bit CPU (Intel T9600, 2.8 GHz), 4 GB RAM, 1 GB RAM GPU, and (2) *NIOS II* embedded processor without hardware acceleration. The graphs are shown in logarithmic scale as functions of n, the number of tree nodes participating in the distributed computation. The data points have been annotated with the original execution times measured in microseconds

A hardware service exposing the object recognition functionality, which is currently under development, was designed according to a simplified *publish-subscribe* model. Specifically, the only public methods available to clients allow them to enroll on or to disenroll from the list of recipients of notifications which are broadcast whenever an object is recognized. We do not yet have a working detection module allowing the device to trigger the recognition automatically when an interest object arises in the field of view of the camera. Therefore, it is currently triggered manually by the operator who presses one of the push-buttons on the device's mainboard.

The experiments done using our device led to the correct classification rate reaching over 90 %, which corresponds to the accuracy of the sequential implementation reported in [22]. Neglecting this aspect, we focused on measuring the computational speedup contributed by the FPGA usage. Figure 13 shows a comparison of the processing speed of a single frame from the input video between the parallel and sequential classifier implementations. The obtained results clearly show that when implemented using hardware accelerator, the classifier runs comparatively fast to the sequential version deployed on a modern PC with nearly 30 times faster system clock. In the same time it is dramatically faster than the sequential algorithm executed solely by an embedded processor.

The computational speedup results from both: parallel implementation of the algorithm and the independence of working memory blocks for each parallel execution path. Since each of those memories are synthesized from FPGA internal memory blocks, they can operate simultaneously at the main clock's frequency of 100 MHz and with low latency regardless of the access manner (sequential or random). Moreover, the achieved speedup fully compensates the data transmission overhead as the web service requests and responses are avoidably small in size compared to the volume of data processed straight on chip. In addition, the FPGA-based realization of the service has a not-to-be-missed property of consuming much less power than the implementation run on a PC—less than 5 W in full-operation mode. This opens up possibilities of utilizing the design in miniaturized smart cameras with economical battery backup or adopting non-standard power supply, such as solar power.

5 Conclusions

In this study a comprehensive approach to building embedded web services has been presented. It is based on an innovative combination of custom-built hardware, off-the-shelf EDA software and our own development tools, altogether referred to as *HSOA Service Builder* suite. The intention of our work was to show that using the herein proposed hardware/software platform one can rapidly build networked applications that, although run in palm-sized devices, act and are perceived as regular web services.

Regarding the hardware side, we make use of a general-purpose FPGA development kit that was augmented with custom peripherals and tuned to act as an intelligent camera. Through the ease of reconfiguration, this flexible design allows the resulting device to replace its functionality, both in terms of internal data processing path and the public interface exposed to the service clients. It also offers other benefits. First, redirecting expensive computations to parallel logic reduces required maximum clock rate, power consumption and execution time. Second, the use of customized DMA channels and burst reads/writes for peripheral-to-memory and memory-to-memory data transfers enables further gain in parallelism and maximizes memory bandwidth utilization. Third, the platform is equipped with a highly customizable network interface which takes over the communication with the external world from the microcontroller. Finally, the proposed architecture gives freedom in adding further peripherals to the system, which would have been much more difficult for the systems based on a general-purpose digital signal processor or microcontroller.

On the software side we introduce two development tools that greatly facilitate the design, implementation and deployment of FPGA-based services according to the SOA architectural pattern. The first tool enables one to create a new service project and persist its state in a form of several constituent sub-projects, each giving control over a given programmable component of the hardware platform. This state can be restored and modified at any time by invoking appropriate third-party EDA package from a common graphical interface. A set of related scripts further automate the development process by ensuring naming and storage path consistency across sub-projects and enabling one-click actions, such as compilation or flash memory programming. The other tool provides a graphical design front-end to the *Impulse CoDeveloper* software which we adopted for rapid implementation of parallel algorithms in an ANSI C based language and automatic HDL synthesis. Our utility comes into play at an early stage of the design process. It facilitates quick GUI-based specification of the accelerator's topology and internal interconnections so that the design best fits the resources of target FPGA.

Future work will focus on two main aspects. On the software side we would like to build a library of parallelized, low-level image processing functions and associated accelerator design templates that could be re-used in various application scenarios. The planned improvements of the hardware platform include changes in the architecture that will allow both the FPGA configuration and the service program to be upgraded remotely and on the fly. Moreover, future version of the platform will be developed on a custom-made, reduced-size printed circuit board.

References

1. Ali U, Malik MB, Munawar K (2009) FPGA/soft-processor based real-time object tracking system. In: Proc of the 5th southern conf on programmable logic, pp 33–37
2. Arias-Estrada M, Rodríguez-Palacios E (2002) An FPGA co-processor for real-time visual tracking. In: Proc of the 12th international conf on field-programmable logic and applications, pp 710–719
3. Baumann D, Tinembart J (2005) Designing mathematical morphology algorithms on FPGAs: an application to image processing. In: Proc of the 2005 international conf on computer analysis of images and patterns, pp 562–569
4. Ben Atitallah A, Kadionik P, Masmoudi N, Levi H (2008) FPGA implementation of a HW/SW platform for multimedia embedded systems. Des Autom Embed Syst 12(4):293–311
5. Bucci G, Ciancetta F, Fiorucci E, Gallo D, Landi C (2005) A low cost embedded web services for measurements on power system. In: Proc of the IEEE international conf. on virtual environment, human-computer interface, and measurement systems, pp 7–12
6. Canny J (1986) A computational approach to edge detection. IEEE Trans Pattern Anal Mach Intell 8(6):679–698
7. Chang CE, Mohd-Yasin F, Mustapha AK (2009) An implementation of embedded RESTful web services. In: Proc of the 2009 conference on innovative technologies in intelligent systems and industrial applications, pp 45–50
8. Cuenca-Asensi S, Ramos-Morillo H, Llorens-Martinez H, Mácia-Pérez F (2008) Reconfigurable architecture for embedding web services. In: Proc of the 4th southern conf on programmable logic, pp 119–124
9. Djemal R, Demigny D, Tourki R (2005) A real-time image processing with a compact FPGA-based architecture. J Comput Sci 1(2):207–214
10. EM1206 BASIC-programmable Ethernet Module, (2011) http://www.tibbo.com/products/modules/x20x/em1206.html, [last accessed: March 2]
11. Erl T (2005) Service-oriented architecture: concepts, technology, and design. Prentice Hall, New York
12. Fielding RT, Taylor RN (2002) Principled design of the modern web architecture. ACM Trans Internet Technol 2(2):115–150
13. Gent GJ, Smith SR, Haviland RL (1994) An FPGA-based custom coprocessor for automatic image segmentation applications. In: Proc of the IEEE workshop on FPGAs for custom computing machines, pp 172–179
14. Gonzalez I, Sanchez-Pastor J, Hernandez-Ardieta JL, Gomez-Arribas FJ, Martinez J (2004) Using reconfigurable hardware through web services in distributed applications. In: Proc of the 14th international conference on field programmable logic and applications, vol 3203/2004, pp 1110–1112
15. Groba C, Clarke S (2010) Web services on embedded systems—a performance study. In: Proc of the 1st international workshop on the web of things, pp 726–731
16. Huttenlocher DP, Klanderman GA, Rucklidge WJ (1993) Comparing images using the Hausdorff distance. IEEE Trans Pattern Anal Mach Intell 15(9):850–863
17. Machado GB, Siqueira F, Mittmann R, Vieira CAV (2006) Embedded system integration using web services. In: Proc of the international conference on networking, international conference on systems and international conference on mobile communications and learning technologies, pp 18–24
18. Meng H, Freeman M, Pears N, Bailey C (2008) Real-time human action recognition on an embedded, reconfigurable video processing architecture. J Real-Time Image Process 3(3):163–176
19. Neoh HS, Hazanchuk A (2004) Adaptive edge detection for Real-time video processing using FPGAs. In: Proc of the 2004 global signal processing expo and conference, pp 27–30

20. Olaru C, Wehenkel L (2003) A complete fuzzy decision tree technique. In: Fuzzy Sets and Systems, vol 138, pp 221–254
21. Pellerin D, Thibault S (2005) Practical FPGA programming in C. Prentice Hall, New York
22. Ruta A, Li Y, Liu X (2010) Robust class similarity measure for traffic sign recognition. IEEE Trans Intell Transp Syst 11(4):846–855
23. Schall D, Aiello M, Dustdar S (2006) Web services on embedded devices. Int J Web Inf Syst 2(1):45–50
24. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)—W3C Recommendation 27 (2007). http://www.w3.org/TR/soap12-part1/ [last accessed: March 2, 2011]
25. Stratix II FPGA Prototyping System (2011) http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=44&No=66 [last accessed: March 2]
26. van Engelen RA, Gallivan K (2002) The gSOAP toolkit for web services and peer-to-peer computing networks. In: Proc of the 2nd IEEE international symposium on cluster computing and the grid, pp 128–135
27. Villalpando C, Some R (2010) Reconfigurable machine vision systems using FPGAs. In: Proc of the 2010 NASA/ESA conf on adaptive hardware and systems, pp 31–35
28. Web Services (2007) Description language (WSDL) Version 2.0 Part 1: Core Language—W3C Recommendation 26. http://www.w3.org/TR/wsdl20/ [last accessed: March 2, 2011]