HW-SW Codesign of a Positioning System. From UML to Implementation Case Study

Ángel Álvarez, Íñigo Ugarte, Patricia Martínez, Víctor Fernández
University of Cantabria
TEISA Department, ETSIIT, Santander, Spain
angel.alvarezr@alumnos.unican.es, {ugarte, pmartinez, victor}@teisa.unican.es

***Abstract*—During the last years, there has been a growing interest in systems related to the location of objects into three-dimensional environments and virtual reality applications. These systems, based on high-performance video-processing, have a big computational load, specially on image analysis phases. This work presents the process of HW-SW co-design and implementation of a positioning system. A methodology was applied in which the requirements and initial functionality was captured in UML-MARTE. After a high-level profiling of the system, an acceleration of most time-demanding stages is achieved by combining the hardware and software capabilities of Zynq platform targeting a low power embedded system. The performance obtained through hardware acceleration of critical parts of the application leads to a significant improvement in the throughput of the whole system. On the other hand, the presented work can also be seen as a proof of concept of the followed methodology.**

Keywords— HW-SW Codesign, HLS, Zynq, UML/MARTE

I. INTRODUCTION

In recent years, systems and products related to location of objects in three-dimensional environments and virtual/recreated reality technology are being increasingly deployed in a large number of sectors, such as robotics, medicine, gaming, among many others [1][2]. To obtain the position of the user or individual, those systems are based on the use of many different elements such as cameras, optical sensors, accelerometers, gyroscopes, GPS or other devices. The positioning system implemented (see [3] for details) on this document is able to locate an individual in any type of environment or light conditions. It is based on image processing of the region of interest where the individual is located using algorithms to detect reference LED markers. When only a reference marker is visualized, the use of stereo cameras and epipolar geometry is necessary. Otherwise, through triangulation by knowing the actual distance between markers and a single camera, it is possible to achieve the parameters to establish the real 3D coordinates of the target on the environment.

Model-driven development (MDD) [4], based on UML/MARTE [5] has been proposed to capture the functionality and requirements of the system. Video processing is one of the areas where high-level modelling and analysis may have a wider impact. From the system model, where all the high-level specifications, HW and SW resources and functional allocation into the platform resources are described, thanks to a software synthesis tool, it is possible to run the executable binary files of the whole system on the target platform.

This document goes beyond the software synthesis from UML/MARTE by carrying out a detailed profiling analysis with the evaluation in seconds or data/seconds of each system stage. The methodology presented takes advantage of the programmable logic in the SoC of the target Zynq [6] platform to design hardware modules in order to accelerate those stages with highest computational load.

This paper is organized as follows: Section II provides an overview of UML/MARTE high-level design and SW synthesis methodology. Section III presents the HW-SW partition based on the timing analysis of the algorithm. In Sections IV and V the HW synthesis workflow and the integration of the generated IP cores are described. The main results of the work are presented in Section VI. Section VII will wrap up the final conclusions.

II. DESIGN METHODOLOGY

The COMPLEX methodology for UML/MARTE modeling and design space exploration, exposed in [7][8] is applied. It follows a Model Driven Engineering (MDE), component-based, software-centric approach. The methodology supports the separation of concerns paradigm, keeping the functional and non-functional aspects well-differentiated. It can completely describe the system, enabling automatic generation of the input code. More concretely, the part of UML/MARTE based on graphical descriptions is used to capture all the required information of the system functionality, the HW/SW platform and the selected architectural mapping or resource allocation, all in a single-source.

A detailed explanation of the UML/MARTE model of the system goes beyond the scope of this essay. However, the Application View is presented for the dataflow of the positioning system in order to have a raw view (see Fig. 1). The Application View includes the description of the application components, the relationship among them, and their interconnection through ports by the set of required/provided services defined by the corresponding

This work has been supported by Project TEC2014-58036-C4-3-R, funded by MINECO.

978-1-5090-4565-5/16/$31.00 ©2016 IEEE

functionalities descriptions. The positioning system presented in this paper is divided into six components:

- InputData: is in charge of getting the view angle from a gyroscope and images from a stereo camera.
- MarkerDetection: obtains the image coordinates of the reference markers.
- MovementType: recognizes the type of movement performed by the user.
- Geometry: estimates the user position by applying the positioning algorithms (such as epipolar geometry and triangulation).
- PossiblePosition: compares different distance values calculated in the last stage for each type of movement and selects the correct ones.
- UserPosition: obtains the target object or individual position in the 3D environment.

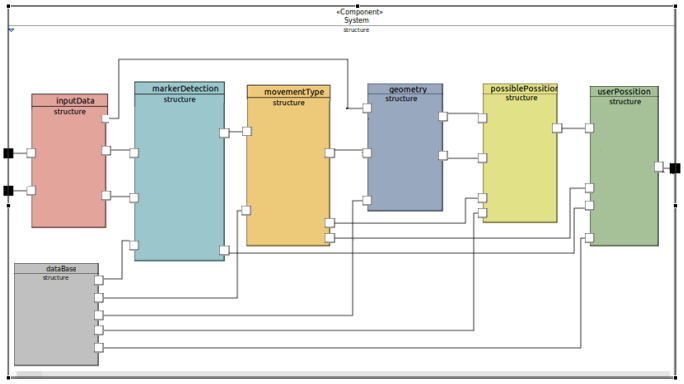

Fig. 1. *UML Application View.*

Once the complete UML/MARTE model has been developed and all the functionalities are defined, simulation and synthesis are the next steps. It is possible to easily create an executable model of the system in the host PC using eSSYN tool [9]. Only when the PC binaries work properly in the PC workstation, the process is repeated with eSYYN tool to generate the selected platform binaries, in order to verify its correctness in the final execution platform.

The system design flow includes the profiling of the timing performance for each component. A first approximation to the model and a time analysis is done, showing that *InputData* and *MarkerDetection* use more than 95% of the whole processing time with low frame rate achievements. The following sections explain how to improve these results with a better utilization of the final platform resources. The *InputData* module has a high dependency on the HW resources of the final platform and, therefore, with few possibilities of improvement via HW-SW codesign decisions. The focus will be on the *MarkerDetection* part of the system.

III. HW-SW PARTITION

The first approach to perform a hardware acceleration of a system is to carry out a detailed timing analysis of the algorithm execution. This profiling can reveal whether or not it is worth the effort to design hardware modules – which usually imply a larger amount of time than software development. If acceleration of the algorithm is required, a complete profiling will provide information about the parts that take most computing time.

There is a first version of the algorithm for marker detection written in C++ (and based on the OpenCV library) and tested on a host PC. Although execution is fast enough on an Intel processor, the final architecture is a Zynq SoC that will make possible to run the algorithm on a low power embedded system. Therefore, profiling is performed running the algorithm on the ARM Cortex A9 of the Zynq. Execution over this platform turned out to be too slow to be used on a fast response system. When processing VGA images (640x480), a frame rate of 11 fps (frames per second) is achieved. If Full HD images (1920x1080) are processed, the throughput goes down to only 1 fps, which is definitely unsatisfactory. The higher the resolution of the images, the sharper is the detection of markers– so processing high definition images can be useful.

The profiling of the software revealed that two OpenCV functions take most of execution time: *cv::medianBlur* and *cv::inRange*, as shown in Fig. 2. Note that percentages are similar for different resolutions, while execution time varies widely.

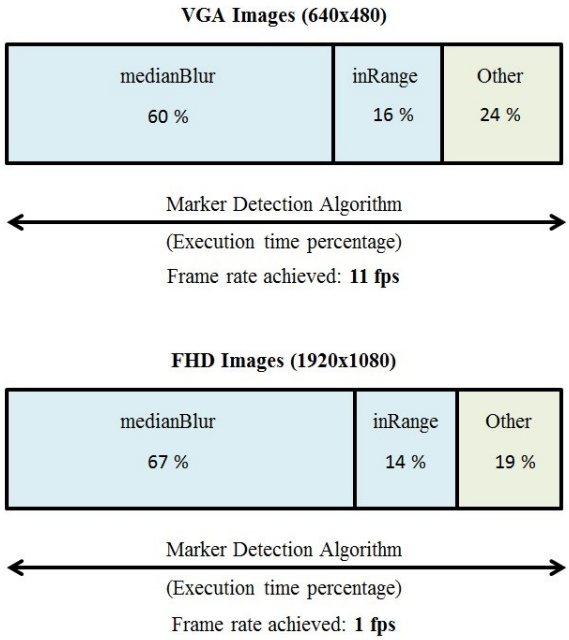

Fig. 2. *Execution time of algorithm showing percentage of most time demanding functions, for VGA and FHD images.*

The functionality of the two functions is studied and it is stated that a hardware synthesis is feasible. Moreover, these two functions are used one followed by the other, so that the output data of the first one are the input data of the second one. This makes that data transfer between software and hardware has to take place only once to execute the two functions. Taking all into account, a HW synthesis of OpenCV's *inRange* and *medianBlur* functions is suitable to perform a good hardware acceleration of the algorithm.

IV. HW SYNTHESIS

Due to the fact that Zynq SoC includes both an ARM based Processing System and Programmable Logic, the objective is to take advantage of the FPGA logic fabric to design some hardware IPs that implement the functionality chosen in the previous section.

To design the hardware IPs cores, Xilinx Vivado HLS (High-Level Synthesis) is used. This tool is capable of converting C-based designs (C/C++) into RTL design files (VHDL/Verilog). However, to obtain a high performance solution, some modifications must be done. An overview of the followed design flow can be seen in Fig. 3.

Although the HLS tool provides a video library that includes some video processing functions with equivalent behavior to corresponding OpenCV functions, the ones desired are not available [10].

The first step is to understand how the required OpenCV functions work and derive a C++ version. Then, they are tested to check they provide the same result as the originals. *InRange* checks if 3-channel pixel values lie within a range of values and outputs one channel images. *MedianBlur* is a 2D median filter applied to images for smoothing, using a certain *NxN* size window [11][12]. In this case of study it works over one channel images. It has the higher computational load.

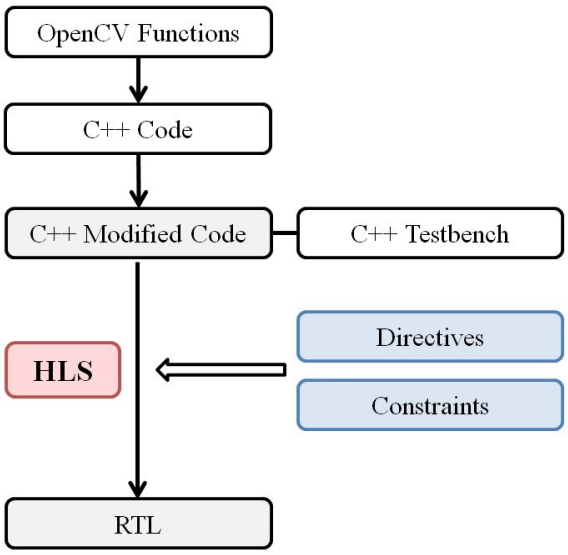

Fig. 3. *Design flow for required hardware IP cores using High Level Synthesis.*

A. Designing the IP core for inRange function

The input and output interfaces for the IPs are AXI Stream. That means that pixels are received and transmitted serialized, one by one. The IP for *inRange* function receives one pixel and outputs one pixel – there is no need for storing other pixels to compute. Input pixel *(x,y)* is enough to generate immediately output pixel *(x,y)*. A pseudocode for *inRange* is shown below:

```
for each row{
        for each column{
                Read input pixel
                Check RGB values
                Write output pixel
        }
}
```

Being things that simple, the HLS synthesis results are good enough for ordinary C++ code input. However, the throughput of the IP can be improved by adding a pipeline directive to the inner loop with an initialization interval of 1 (II=1). Then, an output pixel is produced every clock cycle. The estimated timing performance results, based on HLS synthesis tool reports (not integrated in the final HW-SW system), for both solutions are shown in Table I.

B. Designing the IP core for medianBlur function

Applying a median filter is a very frequent technique in image and video processing. This kind of filtering provides an excellent reduction of noise while preserving the edges in the image. A pseudocode for *medianBlur* is shown below:

```
Read all input pixels
for each row{
        for each column{
                Load 3x3 window pixels
                Calculate median value of window
                Write output pixel
        }
}
```

Unfortunately, a 2D median filter needs several pixels in order to produce an output pixel – in this case, a 3x3 window centered around the considered position in the input image. That is, when a pixel is received, previous and following pixels are needed. The first procedure may consist on reading all input pixels and store the full image into a local buffer. Then, C++ code for median filtering can be applied. When a pixel is being considered and surrounding pixels are needed, they are available to the hardware to compute. This way, a hardware synthesis can be done quickly from ordinary software-like C++ code. However, this can be a pretty bad input for HLS to synthetize high speed hardware. The throughput of the generated IP is lower than 0.5 fps, as shown in Table II.

The reason for the low overall system performance is the absence of appropriate memory buffer architecture. The HLS tool does not automatically manage or create memory buffers. The designer must explicitly describe these structures in the code so that they are generated into the RTL. To improve the design, double buffer architecture is used, as recommended in [13]. The first memory where input pixels are stored is a three-line buffer which acts as a shift register. When a pixel is received, it is loaded into the corresponding position of the third line buffer. Simultaneously, pixels previously stored are moved to the same position on the line buffer above. Secondly, a window buffer capable of storing 3x3 elements from the three buffered lines is implemented. The algorithm computation – calculation of the median value – is applied on

the elements of the window buffer. Line and window buffers are exposed in Fig. 4.

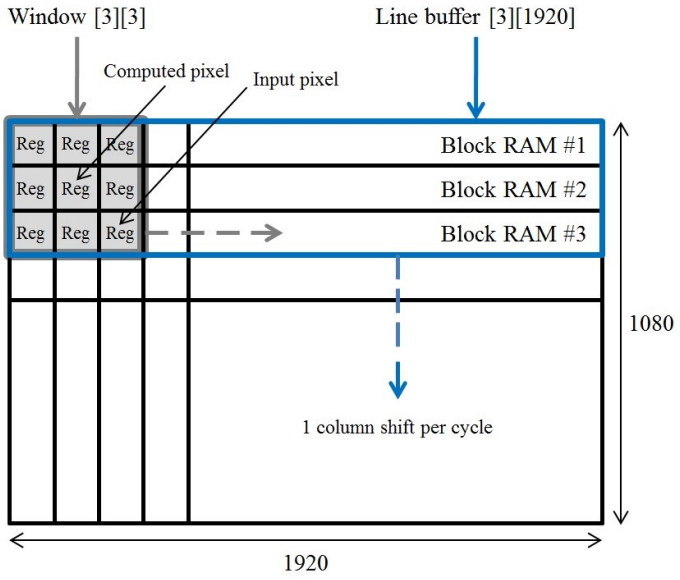

Fig. 4. *Line and window buffers computing with a 3x3 pixels window for high speed video processing.*

Regarding the computation kernel of the algorithm, it is based on calculating the median value of the window pixels for each position on the image. The output pixel in the same position is the result of the calculation. The median value can be obtained by ranking the elements of the window and taking the one on the middle position. Sorting arrays is one of the most critical computations on embedded systems. The non-parallel bubble sort algorithm used in the software version is one of the simplest methods. Nevertheless, performance can be highly improved on FPGAs by describing a sorting network. These structures take advantage of parallel execution of several comparisons on hardware. Networks are built with two-input comparators and registers. Comparators can swap the input values if necessary and have a latency of one clock cycle. Registers are needed to delay elements that do not need to be compared at a certain stage or to add pipeline. To sort the values in window array, an odd-even sorting network is used. A simplified schematic of the sorting network is shown in Fig. 5.

Synthetizing the buffered C++ modified user code does not provide a high performance by itself, but he code is prepared to benefit from some HLS directives. The inner loop is fully pipelined with II=1 (including the buffer management and the sorting network to apply the computation kernel to each pixel).

Therefore, the sorting network has a latency of 9 cycles. After that, a sorted array is produced every cycle. One output pixel is produced at any clock cycle, too. Respecting the memory structures, the 3-line buffer is partitioned into 3 separate line arrays, so that each line is mapped into a different dual-port block RAM. Elements from the three lines can be accessed at a single cycle. Also, the window buffer is fully partitioned so that HLS maps it into 9 registers. Memory mapping of buffers is also indicated in Fig. 4. Combining the code modifications with the synthesis directives, the throughput of the IP reaches more than 74 fps. All estimated timing performance figures are shown in Table II.

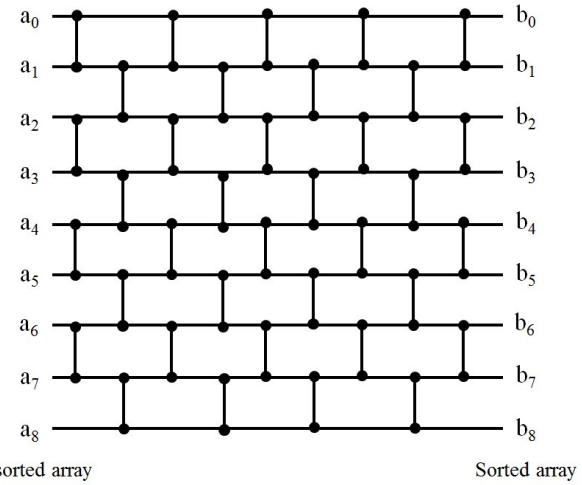

Fig. 5. *Odd-even sorting network for 9-element arrays. Vertical interconnections represent comparators.*

TABLE I. INRANGE PERFORMANCE ESTIMATES

inRange HW (1920x1080)			
HLS Input	Clock period	Clock cycles	IP throughput
Ordinary C++ Code	3.37 ns	4149362	71.51 fps
Ordinary C++ Code & Synthesis directives	3.89 ns	2077922	**123.71 fps**

TABLE II. MEDIANBLUR PERFORMANCE ESTIMATES

medianBlur HW (1920x1080)			
HLS Input	Clock period	Clock cycles	IP throughput
Ordinary C++ Code	4.31 ns	553656603	0.42 fps
Modified C++ Code	4.26 ns	442318177	0.53 fps
Modified C++ Code & Synthesis directives	6.39 ns	2091737	**74.82 fps**

V. INTEGRATION

The application will be running over Linux on an ARM-based Zynq SoC. Some functionality remains in software, such as the video capture using OpenCV library, part of the

processing (including OpenCV library functions) and presentation of results (coordinates of markers). Some other functionality is accelerated through the generated IP cores for *inRange* and *medianBlur*.

The complete system is built and implemented using Vivado IP Integrator. In order to access image data captured by software, the hardware IPs are connected to an AXI VDMA (Video Direct Memory Access) IP. Using DMA allows to read from and write to the main system memory (512 MB DDR3 RAM) without using the CPU. All hardware peripherals are memory-mapped – the registers and memories of the hardware devices are mapped to address values, along with physical RAM. Therefore, it's possible to configure and control the hardware from Linux. The bus interface for user IPs is given by HLS and the bus interface for the AXI VDMA can be found in the IP product guide by Xilinx.

One of the issues that arise when using an operating system and an MMU (Memory Management Unit) is the separation between virtual memory addresses and physical addresses. The instruction *mmap()* allows the user to get a virtual address from a physical address. On the other hand, it is not possible for the user to obtain the physical address corresponding to a virtual address given by the operating system. The DMA needs to be configured with the physical addresses of RAM where data must be read (Read Channel) or written (Write Channel). Using *mmap(),* virtual addresses can be obtained. However, OpenCV image declarations return image buffer virtual addresses that the user cannot choose. So to make data transfer possible, there is a need for matching virtual addresses that OpenCV library functions allocate and virtual addresses corresponding to the physical memory that the DMA accesses. The solution adopted is shown in Fig. 6 and explained below.

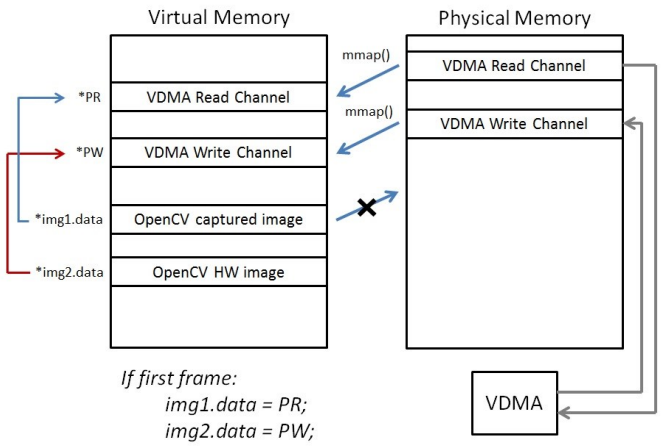

Fig. 6. *Matching virtual addresses given by OpenCV for Mat images and virtual addresses mapped from memory used by the VDMA.*

It is observed that OpenCV's function *Videocapture()* reads the pointer to the user data of the previously captured image to allocate the same memory buffer when used in a loop. Thus, when the first frame of video is captured from the camera, the pointer to the data of the image given by OpenCV is replaced with the value of the virtual address where the DMA Read Channel has been configured to take the data and maintained by *Videocapture()* function during following frames. Regarding to the hardware output image, the pointer to the data of the image given by OpenCV is replaced by the value of the virtual address where the DMA Write Channel leaves the processed data.

It is also important to consider how OpenCV stores image data in memory. For color images (three 8-bit channels), pixels are stored in memory in form B-G-R; B-G-R; etc. For black and white images (one 8-bit channel) pixels are stored in form BW; BW; etc. That is, memory bytes are used continuously. Since the AXI VDMA takes 32-bit words from memory and designed HW IPs interfaces work with one pixel (either color or black and white) per 32-bit word, two – input and output – interface adaptation modules are needed. A simplified model of the final hardware system is shown in Fig. 7.

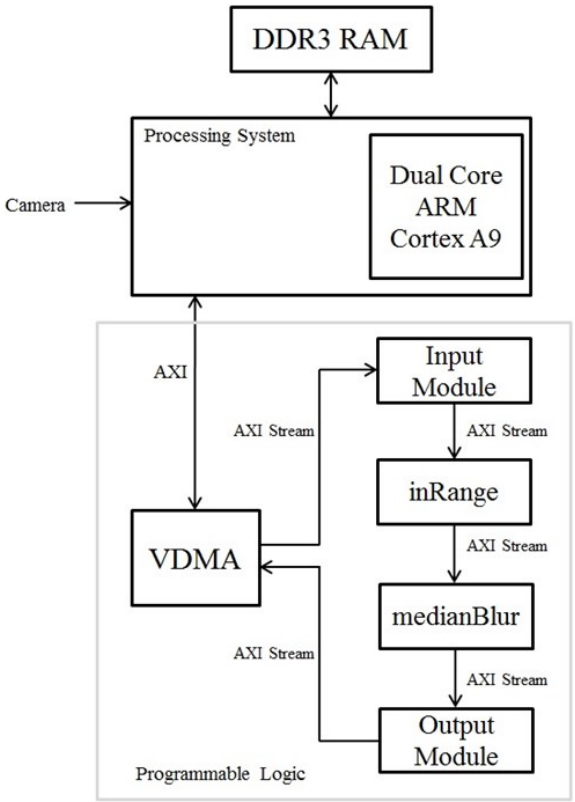

Fig. 7. *Simplified model of the complete hardware acceleration system, the processing system and the DDR3 memory.*

The adaptation IPs are generated with HLS and are pipelined with the rest of the system – operation of the modules is explained in Fig. 8.

VI. Results

In this section, the performance of the application running on the target platform is shown. The FPGA clock period for all hardware IPs is set to 10 ns (100 MHz).

When processing VGA images (640x480), the algorithm execution in software (complete C++ code on the ARM processor) has a throughput of 11 fps (frames per second).

When using the hardware system designed (hardware acceleration), a frame rate of 58 fps is achieved.

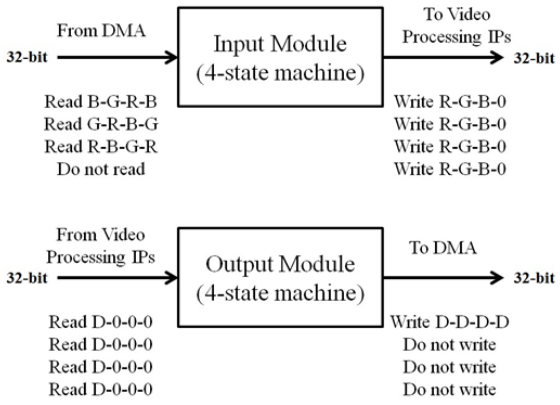

Fig. 8. *Functionality of the interface adaptation IPs: from continuous bytes software storage to one 32-bit word per pixel and vice versa.*

With FHD images (1920x1080), the software version of the algorithm can only process 1 fps. When using hardware acceleration, a frame rate of 8 fps is reached. Fig. 9 shows the execution time of the algorithm for FHD images, both in software and using hardware.

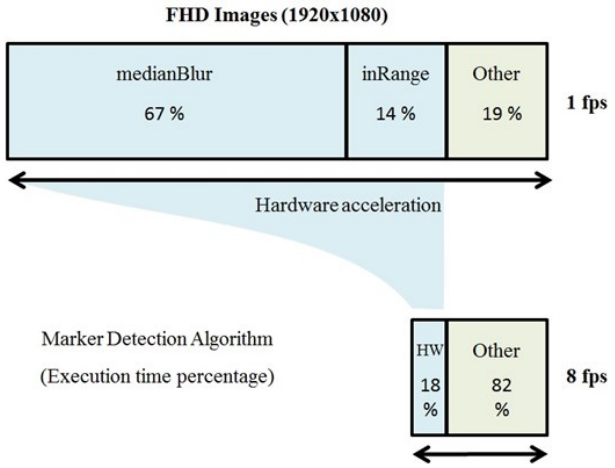

Fig. 9. *Execution time of algorithm for FHD images, comparing software and hardware performance.*

This approach may also be easily scaled to 4K (3840x2160) videos. Higher resolutions lead to an increased benefit from dedicated hardware and a higher bottleneck on the CPU. Therefore, combining HW and SW with increased resolutions such as 4K makes the use of a better microprocessor advisable. Re-generating the designed hardware for these resolutions from HLS is straightforward.

VII. Conclusions

This paper exposes a HW-SW Codesign proof of concept exercise. The initial system requirements and structural specifications were defined using the COMPLEX UML/MARTE methodology. Then, based on a profiling analysis over a preliminary all-SW version, some bottlenecks were detected. Most time demanding parts were derived to HW. An optimum automatic synthesis from C++ requires a deep understanding of the HLS process and the performance and connection characteristics of the target Platform. C++ was transformed in order to admit synthesis directives and with the aim of maximizing the data rate between SW and HW. Final results demonstrate a good speed-up of the positioning system.

References

[1] Nakano, Y. Izutsu, K., Tajistu, K., Kai, K., Tatsumi, T. (2012) *Kinect Positioning System (KPS) and its potential applications.* International Conference on Indoor Positioning and Indoor Navigation.

[2] Mossel, A., Kaufmann, H. (2013). *Wide area optical tracking in unconstrained indoor environments.* IEEE 23rd International Conference on Artificial Reality and Telexistence.

[3] Villar, E., Martínez, P., Alcalá, F., Sánchez, P., & Fernández, V. (2014). *Método y sistema de localización espacial mediante marcadores luminosos para cualquier ambiente.* P.N.ES-2543038-B2.

[4] Schmidt, D. C. (2006). *Model-driven Engineering.* IEEE Computer, V.39, N.2, pp. 25-31.

[5] OMG, UML Profile for MARTE: *Modelling and Analysis of Real-Time Embedded Systems*, Version 1.1, Dec., 2012. Available from: <http://www.omgmarte.org>.

[6] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[7] The COMPLEX project (247999*), Codesign and power management in platform-based design space exploration,* Last visited, 2013. Available from: <http://complex.offis.de>.

[8] Herrera, F., Posadas, H., Peñil, P., Villar, E., Ferrero, F., Valencia, R., & Palermo, G. *The complex methodology for UML/MARTE Modeling and design-space exploration of embedded systems.* Journal of Systems Architecture, V.60, N.1, elsevier, pp.55-78.

[9] Peñil, P. (2014) *UML-Marte Methodology for Heterogenius System design.* Microelectronics Engineering Group, TEISA Dpt., University of Cantabria.

[10] http://www.wiki.xilinx.com/HLS+Video+Library

[11] http://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html

[12] http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html

[13] Fernando Martinez Vallina, *Implementing Memory Structures for Video Processing in the Vivado HLS Tool*, XAPP793