J Real-Time Image Proc (2016) 11:559–569
DOI 10.1007/s11554-013-0351-4

CrossMark

ORIGINAL RESEARCH PAPER

A hardware/software prototyping system for driving assistance investigations

Jakob Anders · Michael Mefenza · Christophe Bobda · Franck Yonga · Zeyad Aklah · Kevin Gunn

Received: 13 July 2012 / Accepted: 24 April 2013 / Published online: 7 May 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract A holistic design and verification environment to investigate driving assistance systems is presented, with an emphasis on system-on-chip architectures for video applications. Starting with an executable specification of a driving assistance application, subsequent transformations are performed across different levels of abstraction until the final implementation is achieved. The hardware/software partitioning is facilitated through the integration of OpenCV and SystemC in the same design environment, as well as OpenCV and Linux in the run-time system. We built a rapid prototyping, FPGA-based camera system, which allows designs to be explored and evaluated in realistic conditions. Using lane departure and the corresponding performance speedup, we show that our platform reduces the design time, while improving the verification efforts.

J. Anders
Department of Computer Science, University of Potsdam, Potsdam, Germany
e-mail: jakob.anders@uni-potsdam.de

M. Mefenza · C. Bobda (✉) · F. Yonga · Z. Aklah · K. Gunn
CSCE Department, University of Arkansas, Fayetteville, AR, USA
e-mail: cbobda@uark.edu

M. Mefenza
e-mail: mmefenza@uark.edu

F. Yonga
e-mail: yfrancku@uark.edu

Z. Aklah
e-mail: zaklah@uark.edu

K. Gunn
e-mail: kcgunn@uark.edu

Keywords System on chip · Prototyping · Hardware/software system · Image processing · Design flow · Driver assistance · FPGA · Hardware acceleration

1 Introduction

Progress in automotive technology is leading to the use of cameras for driving assistance systems, such as lane departure warnings, autonomous cruise control, and occupant pose analysis [14]. The use of cameras has become a promising alternative to conventional range sensors due to their advantages in size, cost, and accuracy. A couple of car manufacturers have developed solutions for driving assistance in the past, some of which have found their ways into high-end production cars. Self-driving cars have recently been approved for use in California. In the near future, these self-driving cars will likely be approved for use in every state in the US, and quite possibly in other countries around the world. Despite these developments, there is still a long way to go in developing this technology to reach a high level of reliability, performance, and affordability.

Algorithms for vehicle or obstacle detection and avoidance will mainly define the behavior of self-driving cars. However, the development environment for such systems should allow a reduction in the time-to-prototype as well as time-to-market, as these are two major cost factors in the development process.

Despite the huge amount of academic research potential offered by driving assistance applications, the investigation of these systems are taking part almost exclusively in industry. Raising the interest of the machine learning and image processing research communities in this field would allow the investigation and deployment of high-qualitative solutions for intelligent transportation. A possible path to

reach this goal is to eliminate the need for those communities to build the low-level, cumbersome architectures required for a real-time, low-cost, and power-efficient processing of complex image processing algorithms. This strategy would allow experts in the fields of artificial intelligence and image processing to focus on the development of intelligent applications that could seamlessly be mapped to efficient computing platforms.

In this work, we present an integrated environment that would help solve this problem and provide designers of driving assistance systems the proper tools to implement, verify, and evaluate their systems in a real environment.

Our focus is on building a generic embedded hardware/software architecture and providing the symbolic representation to allow programmability at a very high abstraction level. We propose a four-level flow, starting from a specification in C/C++ using OpenCV library. Applications are then partitioned at a transaction level and captured by a combination of OpenCV and SystemC representation. Subsequent refinements with a hardware design language will produce a hardware implementation at the register transfer level, which will then be synthesized and verified in a FPGA-based computing infrastructure.

Our main contribution in this work is to leverage existing knowledge in computer vision (1) to provide a holistic design flow for capturing computer vision applications at the highest abstraction level, with subsequent refinements and verification down to the hardware/software implementation, (2) to derive a generic computing path and architecture for complex driving assistance applications, and (3) to design a viable FPGA-embedded camera infrastructure for rapid prototyping and emulation in the field.

The rest of this paper is organized as follows: in Sect. 2, we provide an overview of object detection using visual approaches. In Sect. 2, we present a review of camera systems used in embedded areas. Section 3 presents our design and verification environment, as well as the underlying design flow. We first give a brief description of the design environment, emulation platform, and our motivation behind the design decisions. Thereafter, generic hardware and software modules of our design environment are presented; the system-on-chip design inside the main FPGA is explained with the help of the lane departure application. Finally, the evaluation of our solution is presented and discussed at the end of this section, followed by a conclusion and some indications on future work.

2 Related work

In this section, we focus on published works most close to our contribution, namely embedded system architectures and design flow for video processing. Recently Rinner et al. [12] provided a review of embedded smart camera systems. From their classification, a clear picture of the existing systems can be drawn. Computation is performed in most existing systems in software on a general purpose processor, sometimes optimized for multimedia computation. High-end PCs are used [10] for prototyping purposes, but also to provide the required computational power. The system in [5] uses several digital signal processors on different PCI boards, while the CITRIC [6] relies on the Intel XScale PXA270 processor. The Mesh-Eye [8] uses an Atmel AT91SAM7S microcontroller and computing unit, and the UCLA Cyclops [11] runs on the Crossbow's MICAz platform.

Considering the growing complexity of applications, the computational requirements of embedded cameras can be reached only through a combination of hardware and software, which are integrated today on a single system-on-chip. Usually, hardware and software are developed separately and the integration is done very late. Bugs become difficult to correct and hardware architectures are sometimes not optimal. Capturing the system behavior at a very high abstraction level, and refining the specification along the path to the hardware, will reduce the amount of bugs and allow for more efficient exploration of better architectures.

The camera in [2] uses an Intel PXA270-based development board clocked at 520 MHz. The use of FPGA-based platforms was investigated in [1, 2, 4, 7, 13]. However, the FPGAs were used as co-processors for a single task fixed at compile time.

FPGAs provide the advantage of performance on one hand, but also flexibility on the other. Designs can be emulated and verified in real environments before a dedicated IC device is even built. Using FPGAs in this project is advantageous because of the possibility of changing the design at any stage in the design process; thus speeding up the verification process through a fast execution on the FPGA as well as increased iteration in testing. A design successfully prototyped and verified on an FPGA could directly be used to build a dedicated device, with only minor modifications to the original implementation.

Advanced architectures, in particular reconfigurable architectures like the PACTXPP [3] and Matrix [9], have been developed to provide higher performance and reconfigurability. While the design time is shortened and the efforts reduced with these processors, the resulting hardware are essentially dataflow machines built from coarse-grained computing elements whose scope is limited to the instructions provided by the processing elements. With more specialized computation, as in the case of image processing where a kernel is required, dedicated vector operations (as well as the corresponding buffers) are difficult to build. Their emulation also slows down the overall performance of the system.

In this work, we provide a design flow that will make investigation of embedded imaging, particularly in driving assistance applications, easier. Starting from a high-level specification of an application, subsequent refinements are applied until the generation of the hardware/software system-on-chip structure is completed.

3 Design and verification environment

The high-speed requirements of embedded image processing applications are best served with modern system-on-chip systems. They provide a hardware/software structure in which low level, time consuming, data intensive, and repetitive functions are moved into hardware, while the high-level reasoning is done in software. The hardware parts of the system are usually designed by experienced hardware engineers, using hardware description languages like VHDL. The integration of the hardware and software is classically done at the end of the design process. This approach is error prone, due to either incorrect specifications or misunderstanding during the translation from high-level specifications to low-level implementation.

3.1 Design flow

We propose the design environment of Fig. 1, which consists of four phases: system specification, high-level hardware/software system, register transfer level, and emulation.

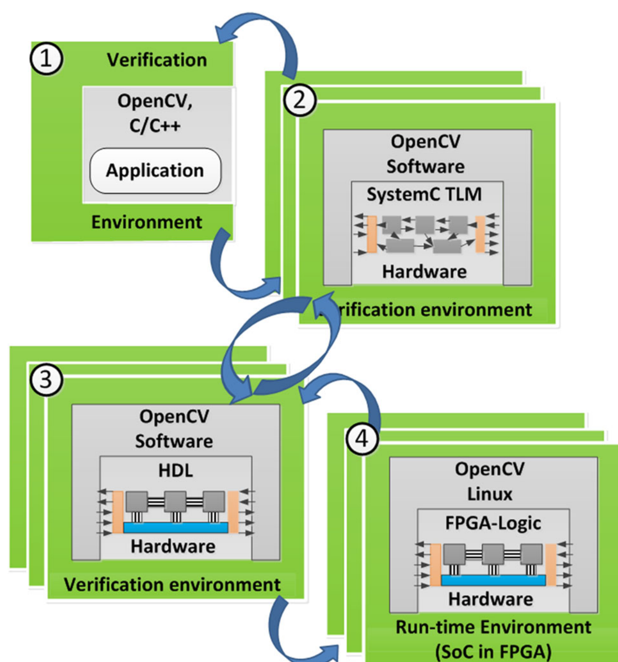

Fig. 1 Our design and design flow

System specification: The first step in system specification is to describe the application, regardless of the target architectures. Applications are defined in executable form in C/C++; the OpenCV library is leveraged. At this level, the verification environment consists of a set of synthetic videos created to simulate driving on a road with obstacles. Several versions of video footage exist, all representing the view of the road and obstacle from a certain perspective and angle. Videos used for verification at this level will be used throughout the entire design process for verification.

High-level hardware/software system-definition: Having now specified the application in OpenCV, the second step consists of performing the first refinement which will produce the hardware/software partition. Here, the behavior of the entire system is refined to a Transaction Level Modeling. Transaction Level Modeling (TLM) is becoming increasingly popular as the ultimate tool to capture and verify systems consisting of several software processes and hardware components. TLM is particularly appealing because of the compatibility with environments that use native C/C++ enriched with abstract communication libraries for sending and receiving messages. This level does not focus on communication implementation details, though, which helps to perform the simulation more quickly. As shown in Fig. 1, we have integrated OpenCV with SystemC in the verification environment to define the hardware/software parts. OpenCV provides all the functionality to provide images, perform reasoning on extracted features, and display results, while the SystemC describes the hardware part for computational speedup as a set of blocks with abstract communication among the blocks. Channels are used as an interface between the software and the hardware, as well as between components. At this level, entire images are transported from one function to the next. Since all the processes and hardware components access the same memory, it is not necessary to actually copy pictures from one memory location to another, we just pass pointers between channels. Since we do not have a tool to automatically partition the design, the designer is in charge of selecting functions to be mapped onto the hardware. This can be done based on profiling that shows the computational "hot spots" in an application. Usually, we would map the object extraction into the hardware, since this part uses only a small number of kernel functions on a huge amount of data. The result is a small amount of data containing information about objects in the picture. Reasoning on this part is not computationally demanding, but requires a complex control structure, which makes implementation in software more viable.

Because the structure of low-level image processing is largely known, we have implemented a set of templates (convolution, thresholding, segmentation, etc.,) that the user can select from in order to quickly build their implementation.

Register transfer level description: At this third step, the abstract TLM description of step two is further refined into a final structure that can be synthesized by hardware compilers. The refinement includes the pin and cycle accurate implementation of the communication interface between software and hardware, a detailed description of the bus model, and a detailed implementation of buffers and memory. At the TLM level, images were transmitted as pointers referring to an entire section of memory. In hardware, however, pictures are transmitted pixel by pixel on a bus. This constraint must be reflected in the Register Transfer Level (RTL). Most image processing functions operate on a certain size neighborhood, which requires some form of sliding windows and additional buffers in hardware to capture the neighborhood of a pixel currently being considered. In our environment, we provide a structure consisting of a buffer to hold each line of pixels for the sliding window and a mask to capture the neighborhood of a pixel. The buffers are configurable in length and number to match the size of the picture and neighborhood being used. With such a clear structure for the image processing block, the function applied on the masks is also a template, whose implementation can simply be set by the user. The current available masks that can be set are: convolutions, edge detection, segmentation, and thresholding. In order to translate the SystemC–TLM description into an RTL implementation that can be synthesized, we implemented the templates in VHDL and automatically mapped the description of level 2 into level 3, without requiring intervention of the user. Transaction parts that rely on object structures are mapped to software along with the part running in OpenCV, while the remaining RTL part of SystemC is mapped to our VHDL module implementation. Interfaces are then introduce in hardware an software along with the drivers (Fig. 1).

Emulation: The last step in the design process is the emulation of the system. For this step, we have designed a versatile FPGA-based smart camera, the *RazorCam* to allow for testing in a real-life environment. Our platform implements image processing directly inside the camera, instead of propagating the image to a workstation for processing. Its compact size and performance facilitates the integration in cars. The processing module consists of: one Xilinx Spartan-6 FPGA, one flash drive, and one connector each for an infrared camera, digital camera sensor, as well as analog camera sensor, respectively. A TFT display can be connected to the platform as well, this allows the user to check the results of algorithms in real-time during an experiment. The verification is done throughout the entire design-flow using the videos available from our verification environment or live video from one of the available image sensor. *RazorCam* features an embedded version of Linux, on top of which OpenCV was installed.

3.2 Prototyping applications

Emulation allows for testing designs in real conditions before moving them on the target platform in the run-time environment. In our case, a system-on-chip architecture can be built inside the FPGA around a softcore processor (the MicroBlaze in this case), running the Linux operating system. In order to provide a familiar design environment to the image processing community, and exploit a widely available image processing library, Intel's computer vision library OpenCV was ported to our platform. This allows applications developed and tested on desktop to be executed on the target embedded system without modification. It is this combination of FPGA, Linux, and OpenCV that makes our platform perfectly suitable for system emulation, with computational image processing mapped directly into the hardware. The FPGA provides maximum performance while still maintaining flexibility, while a fully featured operating system like Linux allows compatible and stable programming without having to worry about system internals and learning processor-specific commands.

To access hardware functions from the software environment, we use the *SDI controller*, which offers a generic connection between hardware and software through shared memory regions. The SDI uses DMA to service memory regions accessible from the software through OS-drivers. The controller's reading and writing regions are defined separately. The reader entity reads data from a defined memory region and pushes it into a connected FIFO. This FIFO is accessible via SDI from the outside, thus the contained data can be propagated to any module implementing the SDI. In contrast, the writer entity takes data from a connected FIFO (which can be filled by the SDI) and writes them to a predefined memory region. Both entities can be started and stopped independently or simultaneously. The structural decision of separating the reader and writer allows for interesting setups. On the one hand, it can be used for intermediate software processing in a hardware chain (for example, by connecting a camera to the writer and a VGA output to the reader) to debug and refine hardware results (Fig. 2). On the other hand, it can also be used for intermediate hardware processing by placing a processing chain of modules between reader and writer (Fig. 2). The camera image is simply placed in the reader's memory region, then propagated to the processing chain, and ultimately written to the writer's memory, where the image is further processed in software.

3.3 Hardware modules

One of the main goals of our design environment is to reduce the design time of system-on-chip for video applications, particularly in driving assistance systems. Those

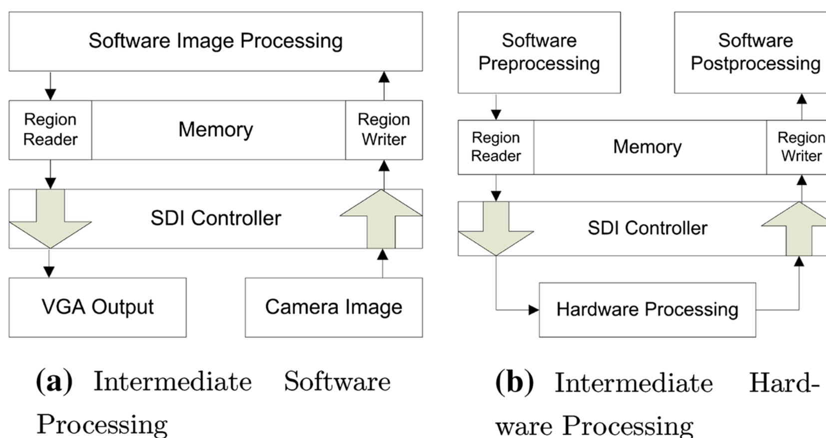

Fig. 2 **a** Usage of the SDI controller for intermediate software processing. **b** Usage of the SDI controller for intermediate hardware processing

applications usually rely on a computing structure which is well known, with data intensive low-level processing in hardware, while reasoning is done in software. Providing a generic structure and components that can be used for many applications (lane departure, obstacle detection, pose analysis, car-to-car communication, etc.,) in driving assistance systems would allow many applications to be as simple as composed from those structures instead of being designed from scratch, thus reducing design time while improving verification.

We have populated our design environment with functions frequently used in those applications. These functions can then be used at the TLM-level to specify the system structure following system specification. The functions are fully parameterizable, the value of the parameters being defined by: picture size, lighting conditions, applied threshold, etc. The library can also be extended to new or complex functions. For each module in TLM, an equivalent RTL version is available to allow a seamless mapping from TLM descriptions into RTL ones. This step is currently conducted by hand, but our future work will seek to automate this process, with efficient design space exploration strategies. In the next section, we provide a brief explanation of the modules currently available in our library.

3.3.1 Convolution filters

Many low-level morphological operations in image processing rely on convolution operators. Noise reduction, smoothing, edge detection, median, and averaging all operate with convolution filters. Our implementation uses a certain number of lines and a sliding window (which consists of previously read pixels) to build the neighborhood of a sufficient distant pixel. The size of the windows can be configured, along with the depth and number of buffers, depending on the degree of accuracy desired.

3.3.2 Threat estimation

During the calibration process, the user can also specify a certain danger zone; all objects found are checked whether or not they are within this zone. For example, for the proposed system (lane departure), the zone is by default placed to comprise of the adjacent lane, extending from the camera to one car length behind the car. When the driver shows the intention of changing lanes, any other vehicle in this zone poses an immediate threat to the host vehicle, thus being considered a dangerous maneuver. As the current speed of other vehicles is taken into account, vehicles at a greater distance (but higher speed) can become a potential risk.

3.3.3 Minimum brightness

To calculate the image's minimum brightness, a generic module providing the lowest in a series of values is implemented. So, mathematically speaking, this means calculating

$$\text{brightness}_{\min} = \min(P) \tag{1}$$

where P is the set of all pixels generated by

$$P := \{p_{x,y} \mid \forall x \in [0, \text{imagewidth} - 1], \\ \forall y \in [0, \text{imageheight} - 1]\}. \tag{2}$$

The expected number of values (pixels in the image) has to be known. The module starts by buffering the highest possible value (taking data width into consideration). For every value (pixel) passing through the module, it checks whether the current value is smaller than the buffered one. If true, the current value is buffered, replacing the original value. When the expected number of values passing through is reached, the buffer contains the minimal value. This value is made available to the outside for the next frame, and the rest of the module is reset.

3.3.4 Average brightness

A configurable module was implemented, which accepts a pre-defined number of values and calculates their mean average. The general approach is to sum the values and then divide the overall sum by the number of values. For the set of all pixels P, this means calculating

$$\text{brightness}_{\text{average}} = \frac{\sum_{p \in P} p}{|P|}. \tag{3}$$

Due to its complexity, division is an operation best to be avoided in hardware. To circumvent the final division, it is replaced by subtracting the number of values from the overall sum whenever possible. If subtracted, a separate counter is incremented to keep track of the number of performed subtractions. When the process is finished (the number of values is reached), the counter holds the result of the division.

3.3.5 Thresholding

Taking both minimal and average brightness from the last frame as input, this module calculates and applies the current threshold. The result is an inverted binary image corresponding to the original image and the thres-hold. The set B of all binarized pixels is calculated by

$$B := \left\{ p'_{x,y} \left| \begin{matrix} p'_{x,y} = 0 & \forall p_{x,y} < \text{threshold} \\ p'_{x,y} = 1 & \forall p_{x,y} \geq \text{threshold} \end{matrix}, \quad p_{x,y} \in P \right. \right\}. \tag{4}$$

For any value passing through the module, the calculated threshold based on minimum and average brightness is applied. Any value below the threshold is converted to 1, any value above is converted to 0.

3.3.6 Integral image

This module is responsible for converting the pixel str-eam from binary pixels to the corresponding integral image. It calculates the set I using the equation:

$$I := \left\{ p'_{x,y} \left| p'_{x,y} = \sum p_{m,n}, \quad m \in [0,x], n \in [0,y], p \in B \right. \right\}. \tag{5}$$

It buffers one line of the image, which is initialized with zeros. In addition, the sum of original pixels left of the pixel under consideration is calculated. With these two sources, the module calculates the integral image pixel by adding the processed pixel directly above it to the sum of original pixels left of it (including its own value in the original image). The result is buffered to form the basis of the subsequent line and then sent to the output stream.

3.3.7 Sum of environment

Although usually in fixed positions from the center of the rectangular environment, the position of the coefficients might change due to the environment overlapping the image borders. In some cases, coefficients have different positions from the center (for example, C and D when the mask overlaps the bottom image border), while in some cases they are not needed at all (for example, A and B when the environment overlaps the top image border). This module calculates the set S using

$$S := p'_{x,y} \left| p'_{x,y} = \sum p_{m,n}, \right.$$
$$\begin{cases} m \in [\max(0, x - \text{env}_{\text{width}}),\ \min(x + \text{env}_{\text{width}}, \text{env}_{\text{width}})] \\ n \in [\max(0, y - \text{env}_{\text{height}}),\ \min(y + \text{env}_{\text{height}}, \text{env}_{\text{height}}] \\ p \in B \end{cases} \tag{6}$$

The process is divided into several stages. In general, a number of lines are buffered as required by the size of the mask. As a first step, the relative position of the coefficients is calculated by checking for overlaps. Afterwards, the resulting positions are retrieved from the buffer using $Result = A - B - C + D$

3.4 Software modules

Like in the case of hardware, a number of software functions interacting with hardware are utilized to implement commonly used processes. The software of the system consists of a small main program, which passes the image coming from the hardware onto and between software modules which are implemented as functions working on the same data structures.

3.4.1 Extract data

The aim of this function is to extract the highlighted shadow areas from the image, thereby switching from complete data represented by the image to only the relevant data represented by tuples.

The algorithm moves through the image line by line, creating a tuple with start and endpoints for each segment of consecutive shadow pixels on the line under investigation. These tuples are saved to a storage data structure and returned to the main program.

3.4.2 Merge data

So far, only horizontal proximity was taken into account for highlighted pixels. The purpose of this function is to check for vertically adjacent tuples and merge them if

appropriate. As a result, each highlighted blob should be represented by a single tuple, storing the coordinates of the bounding box of the blob.

All extracted tuples are processed from the bottom to the top of the image, ordered from left to right. All tuples in the storage have a valid flag, which is set to true for all entries at the start. For each valid tuple in the storage, check all following valid tuples (thereby situated above the current tuple) for vertical proximity and horizontal overlapping. If possible, the two tuples are merged by updating the lower tuple to comprise the upper tuple and invalidating the previous upper tuple.

4 Case study: embedded lane departure in FPGA

The hardware and software modules previously explained and present in our library may be configured and put together. The designer can therefore build a complete system-on-chip from the highlevel description in Open-CV and select the functions to be mapped into hardware. The mapping process will automatically insert the corresponding blocks at different levels in the design flow. Figure 3 illustrates the structure of a system-on-chip for driving assistance at the RTL, consisting of a hardware processing chain for efficient low-level processing of incoming images. Extracted features are passed to the software for high-level reasoning. The SDI interface between the hardware and software is automatically added. The configuration parameters are derived from the designer input (i.e., image size, processing speed, etc…). The emulation is done on the *RazorCam*, which can get the input images from either a server (through the Ethernet port) or from one of the three camera inputs on-board. Results of the processing can either be saved back on the server or displayed on the on-board TFT monitor.

In the previous example, the goal was to build a system-on-chip application that can detect vehicles on the lanes next to the host vehicle at a distance as large as possible. This information could then be used to estimate whether an approaching vehicle is posing a threat to the host vehicle if the driver's intention is to change lanes towards the detected vehicle. For threat estimation, we use a critical area around the host vehicle to provide an understanding of the level of danger a vehicle implicates. We utilize rear cameras (located in the rear window mirror of the host vehicle) with a field of view wide enough to provide information on objects around the host vehicle. The vehicle detection can also be extended by speed estimation of all detected vehicles to make predictions on the future situations.

The system should react in real time with a speed of at least 15 frames per second. This fast image processing goal can be reached in an embedded environment only with the help of customized hardware. The system-on-chip was built using modules of our library, from the TLM level to the RTL implementation. The detailed description of the processing chain within the hardware, as well as the hardware/software interface, is illustrated in Fig. 4.

We tested the system with a set of videos taken from our verification environment, which was provided to the run-time system through the Ethernet port. Even though the platform can handle higher resolutions, 8-bit greyscale images of 320 × 240 pixels were used in this case, since this resolution is completely sufficient to detect nearby objects (up to a distance of about 20 m).

5 Performance evaluation

The goal of performance evaluation is to measure the effectiveness of our platform, as compared to a traditional design environment. This comparison is difficult to quantify with numbers because of the various programmer skills, the type of machine used and additional tasks performed in parallel. Nevertheless, the buildup of our environment allows the user to rely on available basic building

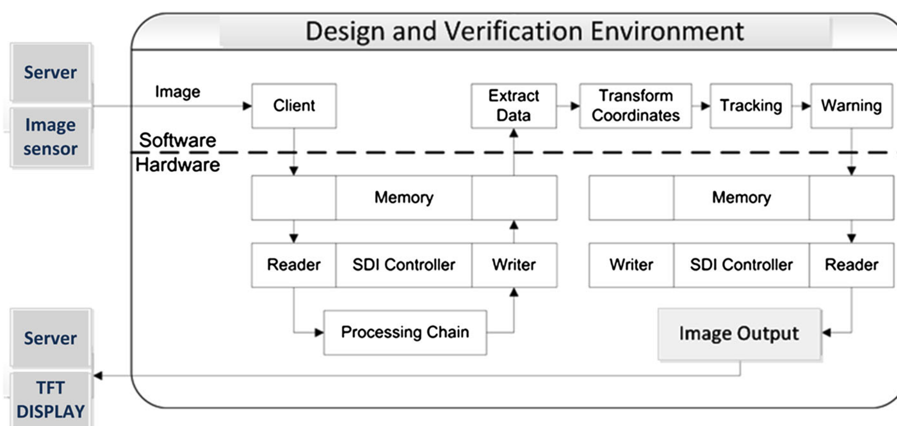

Fig. 3 A complete overview of the implemented system. The extraction and merging phases were combined in a single extraction phase, as well as the threat estimation and reaction phases that form a joint warning phase

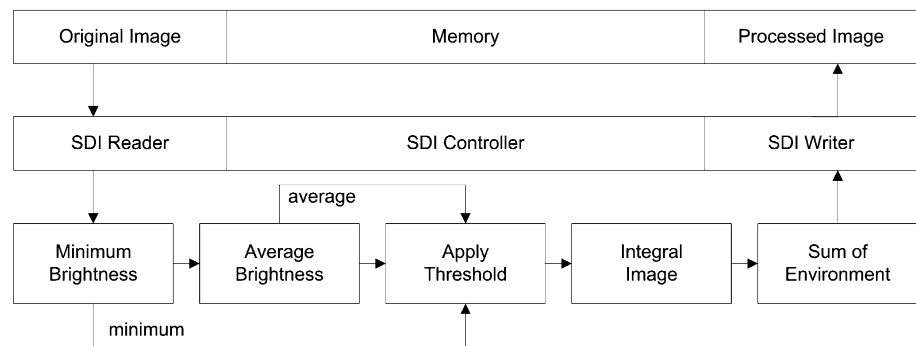

Fig. 4 The final hardware architecture. The image is passed to a processing chain based on the tasks discussed in the concept

blocks and generic structures tailored for imaging applications. Also the entire flow from the high-level specification to the implementation is integrated, with the possibility of speeding-up some tasks during the design process into hardware, thus increasing the number of iteration steps. For many applications that would have taken months to develop in traditional design environments, it was just a matter of days to have a workable and well-tested version on our platform. For illustration purpose we have compiled basic pixel-neighborhood functions with various window sizes in hardware and in software on a desktop development PC. The comparison of execution time in Table 1 shows that, using our design environment and emulating low-level computations on the *RazorCam* will increase the number of iterations, thus speeding-up the design and verification process by at least two orders of magnitude, compared to the PC version. This does not include the time needed to develop, simulate, synthesizes and validate the same hardware modules.

The performance comparison provided below gives an indication on the achievable speedup of the basic block implementations on our platform.

5.1 Threshold

The examined thresholds range from minimum bright-ness (0 %) to 20 % of the difference between minimum and average brightness added to the average brightness (120 %), thus 100 % represents average brightness. The range between minimum and average brightness was chosen, as the perfect threshold is assumed in that region; the values above average brightness were evaluated to support this hypothesis.

Figure 5 shows the results of the thresholding regarding the percentage of present objects that were successfully detected, false positives, merged objects, highlighted image area, and pixels per tuple.

The difference between the hit rate and the sum of merged objects and overexposed images was chosen to find the threshold with the best properties regarding hit rate and errors. The resulting curve was normalized to the range [0; 100] and added to the results (6) to show the best threshold amount. The maximum of the curve (and thus the best threshold) was marked by a dotted gray line to highlight this result.

Table 1 Execution time of basic modules on pure software on development workstation compared to emulation speedup using the *RazorCam* environment

	Hardware *RazorCam*		Software dual core@1.2 Ghz	
Window size	3 × 3	5 × 5	3 × 3	5 × 5
Convolutions	Speed (fps)	Speed (fps)	Speed (fps)	Speed (fps)
Image size	320 × 240	320 × 240	320 × 240	320 × 240
Sobel	268	262	0.2	0.08
Gauss (σ = 1)	274	253	026	0.12
Mean	253	157	027	0.12

5.2 Noise reduction

For intelligent noise reduction, the size and ratio between height and width of the mask are the key features. In the following, the best width and height will be determined experimentally.

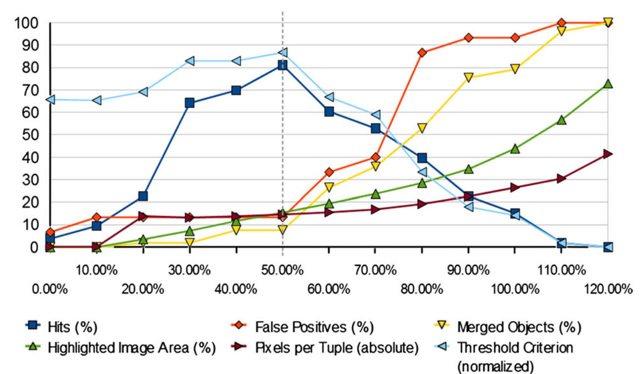

Fig. 5 The complete results of the thresholding experiment. The threshold criterion was added to show that a threshold placed at 50 % of the difference between minimum and average brightness performs best (shown by the *dotted grey line*)

For the experiment, the same image series as in the threshold experiment is used, while a constant threshold of 50 % is applied. To evaluate the general behavior of different ratios between width and height, a series of square, horizontal, and vertical masks are examined and compared regarding the mask's area.

In the first diagram (Fig. 6), the behavior of differently sized masks is examined concerning their performance in correctly detecting objects. Although a small, square 3x3 mask yields the best detection rate, with an increasing area horizontal masks become the best choice, while vertical masks show the worst performance overall.

In the second diagram (Fig. 7), the effectiveness of each mask size is examined by comparing the amount of pixels per generated tuple. As the amount of highlighted pixels stays the same due to the constant threshold, this value now directly stands for the resulting performance in the merging phase, a higher value being desirable.

As a result of the demand to minimize processing effort (with regard to the targeted high frame rate), the opportunity to select a mask with a larger, horizontal area should be taken. Considering the detection graph (Fig. 6), the

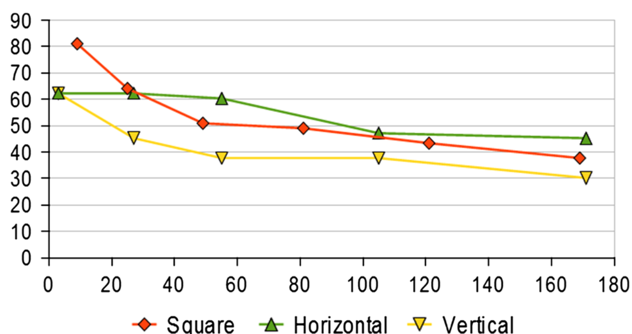

Fig. 6 The performance of differently sized masks regarding the percentage of hits. With increasing size, *horizontally* shaped masks perform best compared to other shapes with a comparable size

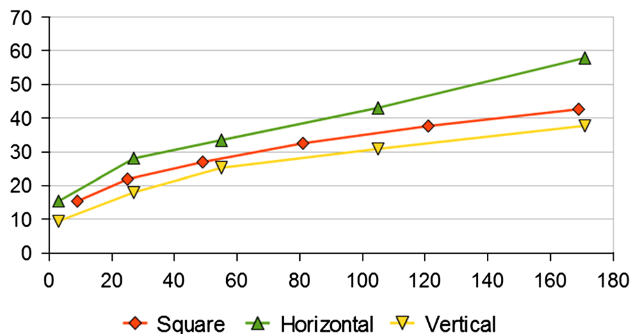

Fig. 7 The performance of differently sized masks regarding the amount of used pixels per tuple. Again, with increasing size, *horizontally* shaped masks perform best compared to other shapes with a comparable size

mask with the area of 55 pixels (11×5) is a good choice, as it is the best choice at this size and the detection performance decreases significantly towards the next examined mask (15×7).

Although the choice is not as straightforward as with the threshold, after considering all aspects, a mask size of 11×5 pixels is selected to be the best and, therefore, used for the implementation.

5.3 Lighting conditions

While it is obvious that a shadow-based system has difficulties with the lighting conditions at night, the following experiment aims to determine how and why the system fails when faced with increasing difficulty.

For this experiment, 60 time-stamped pictures of the same intersection were taken over a period of 92 min in the evening (18:25–19:57, sunset at 19:43) with increasingly difficult lighting conditions due to the fading sunlight. The pictures were taken with a camera using integrated brightness correction based on the histogram analysis to make the images comparable with regard to the thresholding issue. This setup rules out the possibility of general brightness a effecting the results, making it possible to trace any results back to the way the scene is lit, not how much.

To support this assumption, the first diagram (9) shows the development of the contrast (the difference between minimum and average brightness) over the whole image series. The linear regression function was added to the diagram to show the basic constant behaviour, although decreasing slightly.

The more complex second diagram (10) shows the development of the detection rate and the percentage of scene participants having their front lights turned on. The results of the experiment can be seen by applying linear regression functions to both sets of points. With increasing number of light sources (and therefore lighted objects), the detection rate decreases significantly.

5.4 Overall system performance

The aim regarding performance was to implement a prototype running with at least 15 frames per second. As a prototype has completely different aims for different situations (for example, extensive visualization output for demonstration purposes or minimum output for performance tests), the implementation therefore offers different modes of operation (Figs. 8, 9).

Most importantly, hardware acceleration can be switched on and off. The original prototype was done completely in software, and is still used in *software only* mode, using the exact same algorithms that were moved to

hardware later. In *hardware accelerated* mode, the hardware processing chain is enabled and used by the main program to increase the performance.

In *full output* mode, the system shows various processing substeps as images and draws a map of the perceived environment. Although this mode is good for de-monstration and visualization, it is not suitable for performance tests. As there will be no such output necessary in a real application, *minimal output* mode offers a more realistic setup for performance estimation by showing no pictures and printing only what is absolutely necessary.

Table 2 shows the results of a performance test with an image sequence of 200 frames. The frames per second were calculated by measuring the time taken for all 200 frames, then dividing 200 by that measurement.

We have shown that the image processing in software took so much time that even minimizing the output and considering transmission time had hardly any effect. The presented hardware acceleration gave the system an impressive speedup of 15.5. The resulting overall frame rate of an average of 22 frames per second successfully exceeds the targeted 15 frames per second for real time. Table 3 shows detailed synthesis results of all implemented modules for the interested reader.

Table 2 Performance in frames per second of the final prototype

	Full output		Minimal output	
	Incl. ethernet	Excl. ethernet	Incl. ethernet	Excl. ethernet
Software only (fps)	1.29	1.33	1.37	1.42
Speed-up	×7.05	×8.81	×10.43	×15.50
Hardware accelerated (fps)	9.10	11.72	14.29	22.02

Removing the time for transmitting the image via Ethernet, the hardware acceleration achieves a frame rate of 22 frames per second and an impressive speedup of 15.5 compared to the software version

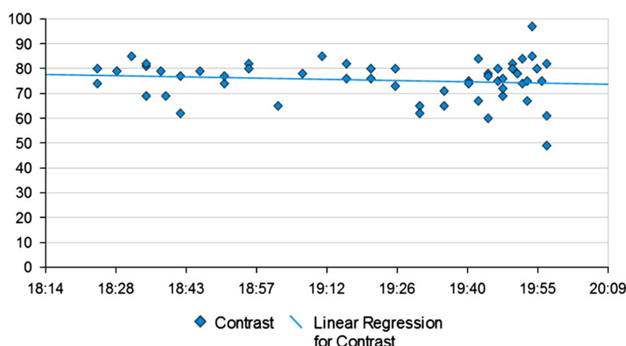

Fig. 8 The development of the contrast (average minus minimum brightness) over time, staying basically the same due to *histogram* analysis performed by the camera used

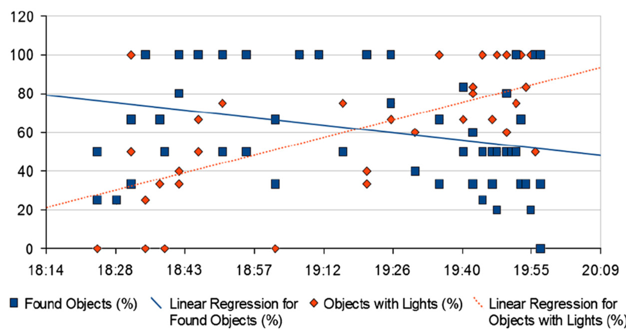

Fig. 9 Detection performance over time in relation to the number of active lights in the scene. The linear regressions are added to the graph to make the development more obvious, while the contrast stays the same (see Fig. 8), the detection performance decreases with the growing amount of light sources in the scene

Table 3 Synthesis results for each implemented module. All modules are capable of running well over 100 MHz

Module	Slices	Flip flops	RAMB16s	Max freq. (MHz)
Serializer	96	104	0	234.71
Minimum	37	49	0	245.27
Average	110	81	0	144.97
Threshold	44	59	0	221.02
Integral	210	244	2	177.97
Integral sum	521	458	16	133.79
Deserializer	87	97	0	202.36

The *Serializer* and *Deserializer* modules are modules that extract single pixels from the 64-bit-wide SDI bus (and put them back together). This way, each intermediate module deals with single pixels on the bus, rather than having to redundantly deal with the extraction/combination individually

6 Conclusion

We presented a hardware/software environment for prototyping driver assistance applications. Our goal was to deploy a generic embedded hardware/software architecture and providing the symbolic representation to allow programmability at a very high abstraction level. We proposed a four-level flow, starting from a specification in C/C++ or OpenCV. Applications are then partitioned at a transaction level and captured by a combination of OpenCV and SystemC representation. Subsequent refinements with a hardware design language produce a hardware implementation at the RTL, which is then synthesized and verified in a FPGA-based computing infrastructure.

The viability of the platform was demonstrated using an exemplary lane departure warning system. Given suitable lighting conditions (daytime, no difficult weather conditions like rain or snow), the system is capable of detecting vehicles in dynamic traffic scenes by finding the shadow area beneath each vehicle. A dynamic threshold based on

the overall brightness of the image was used to find these shadows. A concept was presented where the necessary image processing time was minimized. The complete image processing part of the system was then moved to the customized hardware to further accelerate the system to a real-time frame rate of 22 frames per second.

References

1. Aghajan, H., Cavallaro, A.: Multi-Camera Networks: Principles and Applications. Academic Press, London (2009)
2. Appiah, K., Hunter, A., Kluge, T., Aiken, P., Dickinson, P.: Fpga-based anomalous trajectory detection using sofm. In: Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications, ARC '09, pp. 243–254. Springer, Berlin (2009)
3. Becker, J., Vorbach, M.: Pact xpp architecture in adaptive system-on-chip integration. In: Plaks, T.P. (ed.) Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, June 23–26, 2003, Las Vegas, Nevada, pp. 21–30. CSREA Press (2003)
4. Blackham, B.: The Development of a Hardware Platform for Real-Time Image Processing. The University of Western Australia, Australia (2006)
5. Bramberger, M., Doblander, A., Maier, A., Rinner, B., Schwabach, H.: Distributed embedded smart cameras for surveillance applications. IEEE Comput. Soc. **39**, 68–75 (2006)
6. Chen, P., Ahammad, P., Boyer, C., Huang, S., Lin, L., Lobaton, E., Meingast, M., Oh, S., Wang, S., Yan, P., Yang, A.Y., Yeo, C., Chang, L.-C., Tygar, D., Shankar Sastry, S.: Citric: A low-bandwidth wireless camera network platform. In: Second ACM/IEEE International Conference on Distributed Smart Cameras, pp. 1–10 (2008)
7. Filippov, A.: Encoding high-resolution ogg/theora video with reconfigurable fpgas
8. Hengstler, S., Prashanth, D., Fong, S., Aghajan, H.: Mesheye: a hybrid-resolution smart camera mote for applications in distributed intelligent surveillance. In: IPSN'07, pp. 360–369 (2007)
9. Mirsky, E., DeHon, A.: MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In: Field-Programmable Custom Computing Machines (1996)
10. Pham, D.T., Alcock, R.J.: Smart vision applications. In: Smart Inspection Systems, pp. 157–191. Academic Press, London (2003)
11. Rahimi, M., Baer, R., Iroezi, O.I., Garcia, J.C., Warrior, J., Estrin, D., Srivastava, M.: Cyclops: In situ image sensing and interpretation in wireless sensor networks. In: SenSys, pp. 192–204. ACM Press, New york (2005)
12. Rinner, B., Winkler, T., Schriebl, W., Quaritsch, M., Wolf, W.: The evolution from single to pervasive smart cameras. In: Distributed Smart Cameras, 2008. ICDSC 2008. Second ACM/IEEE International Conference on, pp. 1–10 (2008)
13. Shi, Y., Tsui, T.: An fpga-based smart camera for gesture recognition in hci applications. In: ACCV (1), pp. 718–727 (2007)
14. Wilson, A.: Auto cameras benefit from cmos imagers (2009)

Author Biography

Christophe Bobda received the Licence in mathematics from the University of Yaounde, Cameroon, in 1992, the diploma of computer science and the Ph.D. degree (with honors) in computer science from the University of Paderborn in Germany in 1999 and 2003 (In the chair of Prof. Franz J. Rammig) respectively. In June 2003 he joined the department of computer science at the University of Erlangen-Nuremberg in Germany as Post doc, under the direction of Prof Jürgen Teich. Dr. Bobda received the best dissertation award 2003 from the University of Paderborn for his work on synthesis of reconfigurable systems using temporal partitioning and temporal placement. In 2005 Dr. Bobda was appointed assistant professor at the University of Kaiserslautern. There he set the chair for Self-Organizing Embedded Systems that he led until October 2007. From 2007 to 2010 Dr. Bobda was Professor at the University of Potsdam and leader of The working Group Computer Engineering. Dr. Bobda is Senior Member of the ACM. He is also in the program committee of several conferences (FPL, FPT, RAW, RSP, ERSA, RECOSOC, DRS), the DATE executive committee as proceedings chair (2004, 2005, 2006, 2007, 2008, 2009, 2010). He served as reviewer of several journals (IEEE TC, IEEE TVLSI, Elsevier Journal of Microprocessor and Microsystems, Integration the VLSI Journal) and conferences (DAC, DATE, FPL, FPT, SBCCI, RAW, RSP, ERSA), as guest editor of the Elsevier Journal of Microprocessor and Microsystems and member of the editorial board of the Hindawi International Journal of Reconfigurable Computing. Dr. Bobda is the author of one of the first most comprehensive books in the rapidly growing field of Reconfigurable Computing.