Using Partial-Run-Time Reconfigurable Hardware to accelerate Video Processing in Driver Assistance System

Christopher Claus, Johannes Zeppenfeld, Florian Müller, Walter Stechele
Technische Universität München, Lehrstuhl für integrierte Systeme
Theresienstrasse 90, 80333 München, Germany
Christopher.Claus@tum.de

Abstract

In this paper we show a reconfigurable hardware architecture for the acceleration of video-based driver assistance applications in future automotive systems. The concept is based on a separation of pixel-level operations and high level application code. Pixel-level operations are accelerated by coprocessors, whereas high level application code is implemented fully programmable on standard PowerPC CPU cores to allow flexibility for new algorithms. In addition, the application code is able to dynamically reconfigure the coprocessors available on the system, allowing for a much larger set of hardware accelerated functionality than would normally fit onto a device. This process makes use of the partial dynamic reconfiguration capabilities of Xilinx Virtex FPGAs.

1 Introduction

In future automotive systems, video-based driver assistance will help improve security. Video processing for driver assistance requires real time implementation of complex algorithms. A pure software implementation does not offer the required real-time processing, based on available hardware in automotive environments. Therefore hardware acceleration is necessary. Dedicated hardware circuits (ASICs) can offer the required real time processing, but they do not offer the necessary flexibility. Video algorithms for driver assistance are not standardized, and may never be. Algorithmic research is expected to go on for future years. So a flexible, programmable hardware acceleration is needed. Specific driving conditions, e.g. highway, country side, urban traffic, tunnel, require specific optimized algorithms. Reconfigurable hardware offers high potential for real time video processing and its adaptability to various driving conditions and future algorithms. In this paper we present the architecture of the Autovision project primarily described superficially in [9]. Today's systems for driver assistance offer features such as adaptive cruise control and lane departure warning. Video cameras and radar sensors are used to accomplish this. On highways and two-way primary roads a safe distance to previous cars can be kept automatically over a broad speed range [7] [8]. Basic concepts for modeling vehicle movement and vehicle environment have been developed by Dickmanns [4]. However, for complex driving situations and complex environments, e.g. urban traffic, there are no established and reliable algorithms. This is a topic for future research. Today's video-based driver assistance systems are mainly using dedicated hardware accelerators to achieve real time performance. An advanced example is the EyeQ chip from Mobileye [1]. The EyeQ chip contains two ARM cores and four dedicated coprocessors for image classification, object tracking, lane detection, and filtering. EyeQ offers real time support for a set of applications in driver assistance, but due to its dedicated coprocessor architectures the flexibility and adaptability to future algorithms is limited. The following section presents a typical scenario encountered by driver-assistance applications, along with the requirements that must be met by the video processing within such an application. In section 3 we introduce a system of reconfigurable hardware coprocessors that can be used to accelerate pixel operations, while keeping high level application code on standard CPU cores for flexibility. Section 4 describes an example reconfigurable scenario and its implementation on our target platform. A representative coprocessor architecture is presented in section 5, followed by synthesis results for its implementation in section 6. Finally in section 7 the paper is concluded.

2 Scenario for driver assistance

A video camera in the front of a car takes black/white images in VGA resolution. It is the task of the video processing to extract as much information as possible from the camera signal, by efficiently using the available processing power. This is done independently from other sensor signals, such as radar or infrared. Let us look at the following driving scenario: During the day, a car is driving on a highway and entering a tunnel. On the highway other road users like cars, trucks, bikes, motorbikes or pedestrians have to be recognized and distinguished. We plan to detect these traffic participants by their shape or silhouette. The pixel-level processing can be accelerated by a coprocessor, called ShapeEngine. In this coprocessor only a subset of the Region Shape Descriptor necessary to detect e.g. cars should be implemented. If the road is leading into a tunnel, the tunnel entrance will be marked as a Region of Interest (ROI) for later purposes by the TunnelEngine. A first version of Tunnel Entrance Recognition was implemented in Software [3]. However the execution times are far beyond real-time processing which necessitates the use of a hardware accelerator. At the tunnel entrance, the luminance varies strongly. The tunnel entrance itself is rather dark, the surroundings are bright. The most important area is the road inside the tunnel, behind the entrance. The tunnel entrance was marked as a ROI. Due to the dim lighting, it is impossible to recognize and distinguish road users. Instead, the contrast in the ROI can be enhanced in order to detect obstacles on the road inside the tunnel. This can be done in two steps: 1) Contrast enhancement with motion compensated noise reduction. Global motion can be compensated from speed and steer sensor signals. For noise reduction, the median luminance values at each pixel position can be taken from a set of previous ROIs before contrast enhancement. 2) Obstacle detection using a Sobel Edge Filter. While vertical edges and vertical luminance structures mainly belong to the road and to lane markers, horizontal edges mainly belong to obstacles, e.g. vehicles in the lane. The related pixel-level processing for both steps can be accelerated by a coprocessor, called Contrast/EdgeEngine. The ContrastEngine and the EdgeEngine are currently implemented as two separate coprocessors. In future they will be combined in the Contrast/EdgeEngine. In Addition the EdgeEngine performs a lane detection using the Hough transformation [7] which was fully implemented in hardware. Inside the tunnel, due to the low luminance level, only the taillights of other vehicles can be detected dependable. Headlights, taillights, and fixed tunnel lights have to be distinguished. This is done by luminance segmentation, grouping the taillights into pairs and measurement of position and movement of light areas. The processing on pixel-level can be accelerated by a coprocessor, called TaillightEngine. Example outputs from the Engines can be seen in Figure 1.

2.1 Requirements

Representative requirements for video processing in vehicles in our scenario are: a frame rate of 25 frames /sec (40 ms per frame), a frame size of 640 x 480 pixels VGA resolution, black/white, no color and luminance only and a quantization of 8 bit/pixel. This results in 2.5 Mbit per frame with a data rate of 62 Mbit/sec. Workload conserving processing leaves a time budget of 40 ms per frame. As an estimation for processing requirements, We implemented a simple function to detect spotlights as High level code running on one of the PPC CPUs (300 Mhz). Compared to our coprocessor (3.4 ms) the software solution was more than 175 times slower (around 600 ms). Measured CPU time on a Pentium 4 Processor was 26 ms, thus more than 7x slower as when using the coprocessor. If we imagine a more complex operation hardware acceleration is necessary to achieve real time processing.

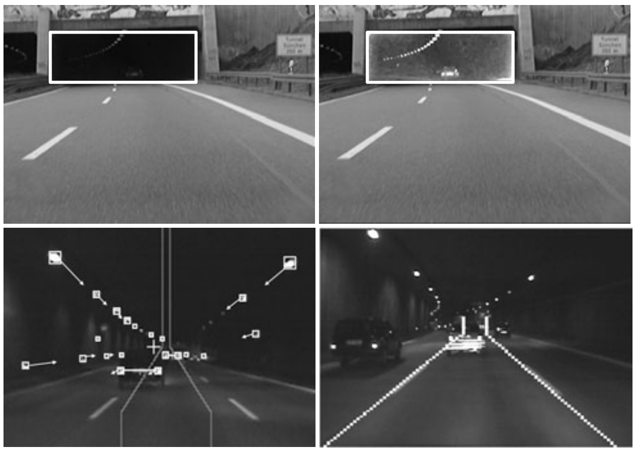
Figure 1. Detection of the ROI (upper left) and Contrast enhancement (upper right). In the lower left corner Taillight tracking is depicted. The lower right picture shows an edge detection performed by a Sobel filter and lane tracking through Hough transformation.

3 Architecture Platform

Algorithms for video processing can be grouped into high level application code and low level pixel operations. High level application code requires a high degree of flexibility and is therefore well suited for a standard processor implementation. Pixel manipulation on the other hand requires applying the same operation on many pixels and thus seems to be a good candidate for hardware acceleration. The TaillightEngine serves as an example for this. Searching for spotlights, thresholding and transferring lights into a list together with its pixel coordinates (to reduce the amount of data) are done in hardware. The CPU only works on this list. Grouping of light spots, detection of static lights e.g.

tunnel lights, detection of light pairs and license plate recognition can then performed as higher level application code on the CPU. An architecture platform has to support both the implementation of high level code on a CPU, and the implementation of low level operations on coprocessors. High level application code is always implemented on the CPU. Pixel-level operations can be implemented for test purposes on the CPU as well, but for real time applications, they are to be accelerated by a coprocessor (xxxEng).

4 Partial Run-Time Hardware Reconfiguration

In general, a large number of different coprocessors could be implemented on a System-on-Chip (SoC) in parallel. However, this is not resource efficient, as depending on the application, just a subset of all coprocessors will be active at any given time. So it would be advantageous to time share the FPGA resources among several coprocessors. With hardware reconfiguration, hardware resources can be used more efficiently. Hand in hand with a software library for pixel operations, a library for coprocessor configurations is planned. Coprocessor configurations can be loaded into the FPGA whenever needed, depending on the application. In our scenario for driver assistance, the coprocessors are used as shown in Table 1.

Environment	Shape Engine	Tunnel Engine	Contrast/ EdgeEng.	Taillight Engine	PPC
Highway	x	x			x
Tunnelentrance		x	x		x
Tunnel inside				x	x

Table 1. Usage profiles of coprocessors

Normally, a small set of pixel-level operations is used in a large variety of algorithms. There is an overlap here as well, so one algorithm may use various classes of pixel-level operations [5]. For each class of pixel-level operations there is a corresponding coprocessor configuration. Each application uses a set of algorithms and the related coprocessors. Our target hardware architecture will use a Xilinx Virtex-II Pro FPGA (XC2VP30) with two embedded PowerPC (PPC) cores on the XUP Development board from Digilent. An overview of the architecture is given in Figure 2. One of the PPCs will be used for high level application code, the other PPC will be used for control and management functions. The configurable logic parts of the FPGA will be used for coprocessors, on-chip bus, data I/O, and memory interface. On-chip Block RAM (BRAM) will be used for local memory with the coprocessors. Loading and replacement of the xxxEngines will be controlled by a Reconfiguration Manager implemented in one of the PPCs.

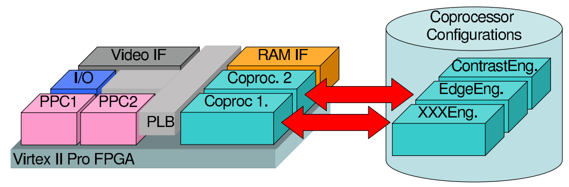

Figure 2. Target Architecture with reconfigurable coprocessors

4.1 Floorplanning

To establish a secure connection between the area occupied by the engine to be reconfigured and the static area, the so called base part, busmacros are necessary. Busmacros are predefined units of logic and wiring that lock the routing between reconfigurable modules and the base system. Instead of using tristate buffer based busmacros like recommended in [11] we utilize LUT-based busmacros which were introduced primarily by Hübner et al. [6]. The coprocessor implementation as described in section 5 requires the use of an IP Interface (PLB IPIF) [14] to be connected to the Processor Local Bus (PLB). It facilitates the connection of user modules to the PLB. The PLB is part of IBMs CoreConnect family of data buses and associated infrastructure. It would make sense to place the busmacro between the PLB and the PLB IPIF. Thus the PLB IPIF would be part of the reconfigurable area and a new coprocessor could be attached to the PLB whenever needed. However, in our system every coprocessor will use PLB IPIFs of the same type. Hence it is not reasonable to locate the PLB IPIF in the reconfigurable area. All signals, except clock signals, have to pass through a busmacro.

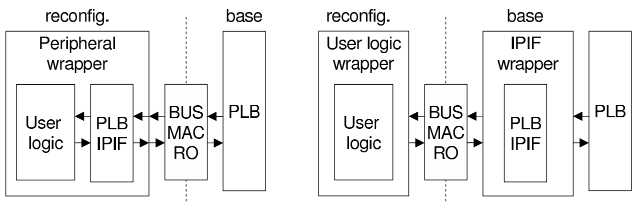

Figure 3. Busmacro Placement: Locating the PLB IPIF in the base part (right picture) instead of placing it in the reconfigurable module (left picture) results in a huge saving of signals that have to cross the reconfigurable boundary determined by the busmacros.

In order to reduce the reconfiguration times by keeping the reconfigurable area as small as possible the PLB IPIF was moved to the base part of the system. This results in an additional advantage. If the PLB IPIF is placed in the reconfigurable area 433 signals between the PLB and the PLB IPIF have to pass through busmacros. Between the PLB IPIF and the user logic there are only 273 signals that have to cross the area boundary. Different options for busmacro placement are depicted in Figure 3. The layout of our first reconfigurable architecture is shown in Figure 4.

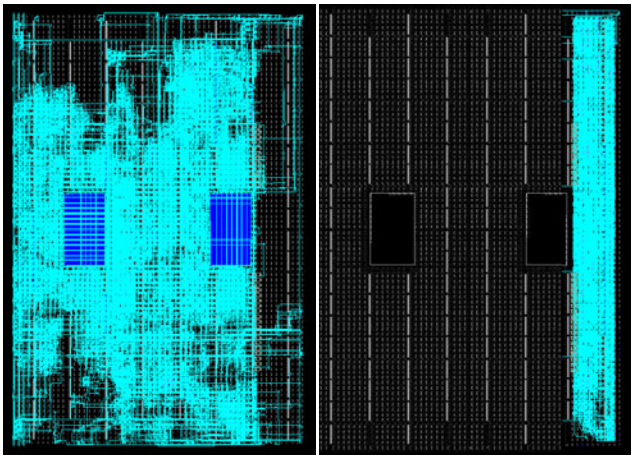

Figure 4. Base design on the left side and reconfigurable module (AddressEngine) on the right side

A coprocessor, the AddressEngine which is introduced in the following section, is constrained to the right side. The remainder of the system, the base design, that includes e.g. the video input and output remains fully operational during the reconfiguration process. The nets that span the reconfigurable area to connect the base design to the I/Os on the right side of the device (left picture of Figure 4) are not affected by the dynamic partial reconfiguration due to the glitchless switching capabilities of VirtexII FPGAs. These nets that cross the reconfigurable (blank) area are not used in the reconfigurable module. Therefore no busmacros are needed to span the reconfigurable area. The future architecture will have the ability to run and reconfigure two separate coprocessors in parallel to be conform with the scenario shown in Table 1. We are working on processes to keep the reconfiguration times as short as possible. In that context we introduced our Combitgen Tool [2] that is used for filtering out the different frames between all possible coprocessor configurations. Combitgen then combines only these frames that are necessary to change the functionality of one coprocessor to any other coprocessor configuration in partial bitstreams. As the number of frames in a partial bitstream is directly proportional to the reconfiguration time a reduced number of frames in configuration bitstreams results in shorter reconfiguration times. Considering a frame rate of 25 frames/sec results in 40 ms to process one image. If the image processing can be done in e.g. 35 ms and the reconfiguration of a coprocessor can be done in 5 ms then no frame has to be dropped. Thus we are interested in accelerating the reconfiguration process as much as possible. A partial bitstream for a representative coprocessor is almost 300 Kbyte in size. If we could achieve a throughput of configuration data around 100 KBytes/ms (100 Mhz ICAP clock and 8-bit ICAP input width) the reconfiguration could be done in 3 ms. This part of our future work.

5 Coprocessor Implementation

The first coprocessor that was implemented is the AddressEngine (AE), a general purpose engine that can be used for a large variety of pixel-level operations. A preliminary version of this coprocessor has previously been presented in [10]. Despite its generic nature, the AE serves as a good representation of the functionality found within other coprocessors, and can indeed be used to emulate most other coprocessor configurations. The driving idea behind the AE is that pixel addressing is a very repetitive process, yet can actually require more processing time than the pixel calculation itself. Thus it makes sense to accelerate the addressing requirements of a pixel processing function using a hardware coprocessor. In addition to providing faster address calculations, this approach allows for hardware acceleration of the processing function by implementing the addressing code as a wrapper around a small, user-configurable pixel operation. This user-defined function then receives individual pixels in the correct order, and can focus solely on the processing required by that one pixel, making the hardware implementation of such a function fairly straightforward. This user-defined function determines whether it is a TaillightEngine, a ShapeEngine or any other XXXEngine.

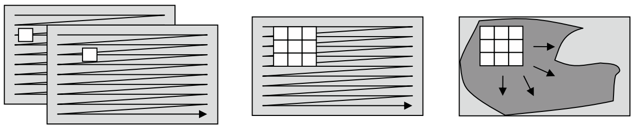

Figure 5. Pixel addressing schemes: Inter (left), Intra (middle), and Segment (right) addressing. Arrows indicate the direction of pixel processing

There are three basic types of pixel addressing as depicted in Figure 5. Inter-addressing is used to concurrently process pixels from multiple source images, needed for example to calculate the difference between two images. Intra-addressing provides a neighborhood of pixels surrounding the pixel of interest, useful for most filtering applications. Finally, segment addressing is used to traverse an image not row by row but instead according to a set of homogeneity criteria. This allows for the detection of contiguous regions, especially useful for object identification. The AE currently supports both inter- and intra-addressing, leaving segment addressing as a candidate for future expansion.

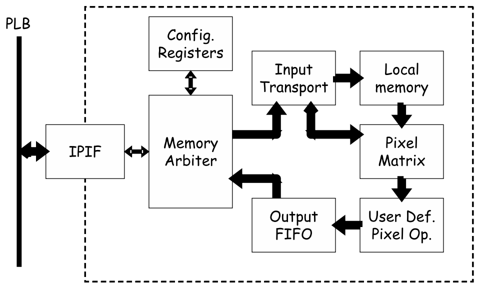

Figure 6. Architecture overview of the AddressEngine. Note the configuration registers (top) and processing pipeline (right)

Figure 6 shows a block diagram of the AE, which is composed of two parts: a processing pipeline responsible for addressing pixels and carrying out the requested pixel operations, and a set of configuration registers used to store all processing parameters and other runtime configurable settings. These registers fall into one of three categories. The first is the set of registers that are used to parameterize the operation of the AE itself. The second category of registers is a set of global variables that can be used to provide additional application-specific parameters to the user defined pixel operation, and are not otherwise used by the AE. Finally, the processing pipeline requires a few secondary parameters derived directly from those supplied by the user. The configuration registers unit is responsible for generating these additional values, and for distributing them to the system along with all other registers. All user-defined pixel operations are executed in the processing pipeline, along with the addressing required to retrieve pixels from memory. The input transport works in conjunction with the pixel matrix to perform these address calculations and write the complete processing window into the local memory of the AE. In addition to iterating across all pixels within an image, the input transport is responsible for scaling of the image, adjusting addresses for an origin offset, generating default values for pixels outside of the source image, and providing pixel data from multiple images when required for inter-addressing. Since accessing pixels from memory is relatively slow, a local memory is used as intermediate storage for all pixels currently required by the processing function. This prevents the same pixel being accessed multiple times from external memory, enhancing performance especially during intra-addressing with large neighborhoods, where a single pixel can be required many times during the processing of surrounding pixels. Storing the pixels in a local memory also allows the neighborhood matrix to access an entire column of pixels perpendicular to the scan direction simultaneously. This makes it possible to shift the matrix one pixel horizontally in only a single clock cycle. Other parts of the processing pipeline include a simple FIFO used as temporary storage for result pixels until they can be written to the main memory. The memory arbiter is responsible for scheduling all transactions across the PLB IPIF [14] to the Processor Local Bus (PLB). It accepts any incoming requests from external devices to read or write configuration registers, and also initiates memory transactions when requested by blocks within the processing pipeline. Finally, the user defined pixel operation is responsible for actually calculating a result from the pixel neighborhood provided by the matrix. The overall design of the AE, with configuration registers used to parameterize a pixel processing pipeline, can also be applied to most other designs. Indeed, by removing unneeded support for the highly parameterized addressing schemes found within the AE, portions of its pixel pipeline code can be reused to satisfy the pixel addressing demands of other coprocessors. Due to its user-configurable pixel operations, the AE can satisfy the requirements of a large number of image processing algorithms. It cannot always be used in place of other coprocessors, however, since a coprocessor dedicated to a certain pixel operation can be optimized specifically for that operation, which generally reduces both the required chip area and processing time. Thus there may be multiple coprocessors that are useable for a certain operation, which makes a common setup of function parameters for different coprocessors desirable.

6 Synthesis Results

The AddressEngine has been implemented as a device on the Processor Local Bus (PLB) of a Xilinx Virtex II Pro.

Number of Slices:	1523	out of	13696	11%
Number of Slice Flip Flops:	1346	out of	27392	4%
Number of 4 input LUTs:	2710	out of	27392	9%
Number of bonded IOBs:	273	out of	556	49%
Number of BRAMs:	9	out of	136	6%
Minimum period: 9.370ns (Max. Freq: 106.724MHz)				

Table 2. Device Utilization and timing summary of the AddressEngine

The synthesis results for a typical AE configuration are shown in Table 2. Only a single pixel operation was implemented in this case, namely a 8-bit, 3x3 sobel filter. Generally it is possible to have many more operations supported by a single instance of the AE, though multiple pixel operations will of course increase the required chip area. The width of each individual pixel also has a profound impact on the area requirements of the AE, especially for the large array of pixel values that make up the matrix. Despite employing a local memory to minimize the number of memory transactions, the bottleneck of the system still lies in fetching and storing pixels from the main image memory. While a fairly simple pixel operation (such as the 3x3 median filter used here) takes only a single clock cycle to process each pixel, retrieving a single pixel from memory can take upwards of ten clock cycles. All of the Engines developed and implemented so far are able to run at the PLB clock frequency of 100 MHz or even above. Other engines like the TaillightEngine occupy 20% of the slice resources and 13% of the BRAM blocks. This should serve as benchmark for a typical Coprocessor implementation.

7 Conclusion and further work

This paper has introduced a flexible hardware design for accelerating the processing required by future video-based driver assistance systems. By using hardware coprocessors to accelerate individual pixel operations, the software based high-level application code remains flexible and adaptable to future algorithms. In addition, multiple algorithms requiring the same pixel operation can share a coprocessor to minimize the necessary chip area. We presented our target architecture with two embedded CPU cores and a set of FPGA-based, reconfigurable accelerator engines. Finally, the Address-Engine has been introduced as a representative coprocessor architecture, along with some of the hurdles that had to be overcome during its development. Further work includes extending the functionality of this coprocessor; adding further coprocessor configurations to the library, i.e. ShapeEngine or a separate lane detection engine to detect curved roads; and evaluating the dynamic reconfiguration behavior of all the coprocessors. In addition we plan to implement a novel PLB Controller for the Internal Configuration Access Port (ICAP) that should enable on-chip reconfiguration and achieve a data throughput near the theoretical boundary of 100 Kbyte/ms (ICAP's input width is 1 byte, ICAP frequency is 100 Mhz). Information about ICAP can be found in [12], [13] and [15].

8 Acknowledgements

This work is supported by the German Research Foundation DFG (Deutsche Forschungsgemeinschaft) in the focus program No. SPP1148. We want to thank the Institute for Information Processing Technology (Institut für Technik der Informationsverarbeitung - ITIV) in Karlsruhe, especially professor J. Becker and M. Hübner for their support in the field of dynamic partial reconfiguration. Exceedingly we want to thank professor A. Herkersdorf for his insightful comments. Finally we want to thank Xilinx for providing development boards for our research activities.

References

[1] www.mobileye.com.

[2] C. Claus, F. H. Müller, and W. Stechele. "Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro devices". *Workshop on reconfigurable computing Proceedings (ARCS 06)*, pages 122–131, March 2006.

[3] C. Claus, H. C. Shin, and W. Stechele. "Tunnel Entrance Recognition for video-based Driver Assistance Systems". *Proceedings of 13th International Conference on Systems, Signals and Image Processing (IWSSIP 2006)*, September 21 - September 23 2006.

[4] E. D. Dickmanns and B. D. Mysliwetz. "Recursive 3-D Road and Relative Ego-State Recognition". *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):199–213, 1992.

[5] S. Herrmann, H. Mooshofer, H. Dietrich, and W. Stechele. "A video segmentation algorithm for hierarchical object representations and its implementation". *IEEE Transactions on Circuits and Systems for Video Technology, Special Issue on Object-Based Video Coding and Description*, 9(8):1204–1215, December 1999.

[6] M. Hübner, T. Becker, and J. Becker. "Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration". *Proceedings of the 17th symposium on Integrated circuits and system design (SBCCI 04)*, pages 28–32, 25-29 April 2004.

[7] J. B. McDonald, J. Franz, and R. Shorten. "Application of the Hough Transform to Lane Detection in Motorway Driving Scenarios". *Proceedings of the Irish Signals and Systems Conference*, pages pp.340–345, July 2001.

[8] K. Sobottka, E. Meier, F. Ade, and H. Bunke. "Towards Smarter Cars". *Sensor Based Intelligent Robots, International Workshop, Dagstuhl Castle, Germany, Selected Papers*, 1724:120–139, September 28 - October 2 1998.

[9] W. Stechele. "Video Processing using Reconfigurable Hardware Acceleration for Driver Assistance". *Special interest workshop on Future Trends in Automotive Electronics and Tool Integration (DATE 06)*, March 6 - March 10 2006.

[10] W. Stechele, L. A. Carcel, S. Herrmann, and J. L. Simon. "A Coprocessor for Accelerating Visual Information Processing". *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 26–31, 2005.

[11] Xilinx, Inc. "XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based". v4.0, 9th September 2004.

[12] Xilinx, Inc. "XAPP662: In-Circuit Partial Reconfiguration of RocketIO Attributes". v2.4, 26th May 2004.

[13] Xilinx, Inc. "OPB HWICAP (v1.00.b) Product Specification". pages 1–13, 4th March 2005.

[14] Xilinx, Inc. "PLB IPIF (v2.02a)". *DS448*, 15th April 2005.

[15] Xilinx, Inc. "UG0012: Xilinx Virtex-II Pro and Virtex-II Pro X FPGA User Guide". v4.0, 23 March 2005.