

Hardware/software architecture of an algorithm for vision-based real-time vehicle detection in dark environments

Nicolas Alt, Christopher Claus, Walter Stechele
Technische Universität München, Lehrstuhl für Integrierte Systeme
Theresienstrasse 90, 80333 München, Germany
n.alt@mytum.de, christopher.claus@tum.de, walter.stechele@tum.de

Abstract—Hardware/software partitioning of algorithms is gaining more and more importance in order to benefit from the advantages of both worlds. Pure software implementations are easy to change but the processing time is rather high. By contrast pure hardware implementations usually result in faster processing due to inherent parallelism but they do not offer the necessary flexibility for quick changes and adaptations. In this paper the hardware/software co-design of a self-developed algorithm to detect cars by their taillights as well as its implementation on an embedded system (FPGA) is presented. Instead of utilizing expensive sensors such as RADAR which also can be used to detect obstacles in dark environments, the detection method presented here is based solely on grayscale images taken by a low-cost on-board camera which was mounted on a moving vehicle. Only computationally intense parts - namely pixel or sliding window operations - are implemented in hardware to achieve the necessary real-time requirements. The remainder of the algorithm - the so called higher level application code - is running on standard embedded CPU cores. With this architecture it is possible to process the incoming video-stream (25 frames/s) and detect cars in real-time on an embedded system.

Keywords: driver assistance, real-time video processing, hardware acceleration, taillight detection

I. INTRODUCTION AND RELATED WORK

Current algorithms in the driver assistance domain are not standardized, and might change rapidly. Therefore a flexible platform is necessary. The proposed vision-based 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 CPU cores to allow flexibility for new algorithms and quick changes. Different engineers will implement their own algorithms for vehicle detection. All these algorithms, however, will need to extract candidate spotlights, so called feature points, from the image. The detection of these spotlights is mainly based on pixel operations which is a process that is very repetitive and easy to parallelize. So this part is obviously suitable for an implementation in hardware and therefore no tool for complex automatic HW/SW partitioning was required. The desired framerate is 25 frames/s, which results in a maximum processing time of 40 ms per frame. Many authors have published their work on detecting cars by their taillights. Betke

et. al. [1] and Chern et. al. [3] present methods to detect cars by their taillights based on thresholding. Thresholding is an important method for image segmentation in which pixels with similar brightness values are grouped together. In the simplest case this is a binary decision based on whether the brightness values are above or below this threshold. The threshold value itself is a critical parameter which is responsible for the degree of segmentation obtained. One of various techniques to detect a kind of optimum threshold is described in [6]. Cucchiara et al. describe in [5] a method to detect cars by their headlights under night illumination. After a thresholding step headlight detection via morphological analysis is performed by considering criteria such as shape, size and the minimal distance between vehicles. After that the algorithm searches for luminance values in horizontal direction. The authors in [9] use stereo-vision and color information to track the taillights from a leading car. Cabani et. al. also detect taillights of cars in [2] by exploiting the colour information in a video frame. Kim et. al. propose in [8] a method to detect cars by their front- and taillights by introducing an additional classification step in which the size of the light regions is taken into account.

The paper is organized as follows: In section II the algorithm to detect cars by their taillights is described followed by the system implementation on an reconfigurable hardware (FPGA) in section III. Experimental results namely execution times of the algorithm are depicted in section IV. Section V concludes this paper with an outlook on future research activities.

II. AN ALGORITHM TO DETECT CARS BY THEIR TAILLIGHTS

The system described here was designed specifically for tunnel or nighttime driving on roads with separated lanes for each direction, such as freeways.

Fig. 1 (left) shows a typical input image seen in those environments. Mainly active light sources, such as taillights of cars or static lights from the tunnel, are clearly visible. Therefore, this system detects cars and trucks solely based on taillight pairs and the license plate illumination visible in a camera image, suggesting the name TaillightEngine.

First, an algorithm was designed and evaluated that searches for pairs of lights that possibly stem from a car. Multiple

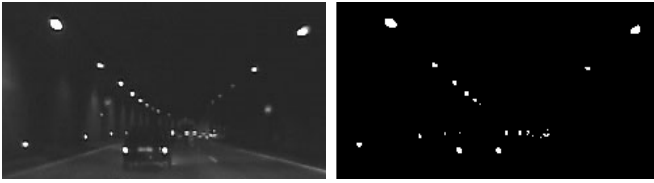


Fig. 1. A typical input image (left) and the intermediate output of the SpotlightEngine (right)

properties of each light are matched against those of other lights. Also, the space between two lights is searched for the illumination of a license plate. The algorithm was separated into a hardware and a software part, and later both parts were implemented. It was shown that, thanks to the hardware acceleration, real-time processing can be achieved on an embedded processor. The demonstration system processes a video recorded on a real drive and draws the detected objects on a monitor, overlaid on the original image (see fig. 2).



Fig. 2. Final output image of the system. Markers show all detected lights, their motion vectors and detected vehicles.

Fig. 3 depicts a flowchart of the proposed detection algorithm. The input image is recorded by a standard video camera, sensitive to visible light, and recording 25 grayscale frames per second. The hardware accelerator consists of the SpotlightEngine and the LabelingEngine. Processing in SpotlightEngine and subsequent thresholding results in a binary image that only shows spotlights, as shown in fig. 1 (right). A spotlight is defined to be a roughly round region of bright pixels. Typically, active light sources, such as taillights or tunnel lighting, produce spotlights in the video. The LabelingEngine searches the binary image for regions of connected white pixels and creates a label for each region. It converts the binary image to a list of labels, each entry corresponding to one spotlight. This list is the output from the hardware accelerator and is handed over to the CPU as can be seen in figure 3. The list of labels is smaller than the bitmap, so the amount of data has been reduced considerably during this step. For reasons discussed later, the labeling process is split into two separate parts. The following software modules, except PlateSearch, operate only on the list of labels and do no longer need to access the bitmap data. The number of labels is relatively low, making it possible to perform complex operations on them without hurting the real-time criteria. Typically, between 20 and 35 entries were seen while driving in a tunnel.

Static lights (see section II-C) are detected first, based on

motion vectors. These are lights that do not belong to a car, but are fixed to the tunnel walls or to the road. Next, lights within a certain proximity to each other are paired, because each car is assumed to be represented in the video as a pair of taillights. This means that motorbikes or cars with a damaged taillight are not detected currently. Several properties are used to score how well these so-called candidate pairs (corresponding to candidate vehicles) match, and the best scores are used to form a final list of pairs. One of these scores is determined by the module PlateSearch, which scans the area between the taillights for the illumination of a license plate. Afterwards, the temporal continuity of each pair is determined. Only pairs that have already been detected in previous frames at a similar position are marked valid. Finally, valid light pairs with a very high score are selected as corresponding to an actual vehicle. The score is a measure of the likelihood that the detection was correct.

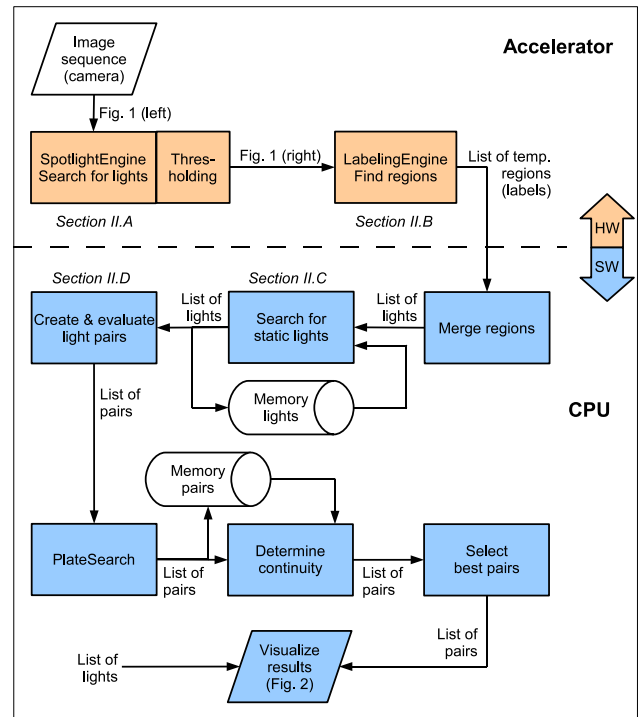


Fig. 3. Flowchart of the algorithm

A. Detecting spotlights

Spotlights are local, bright regions in the image that are ideally round and surrounded by relatively dark pixels. Taillights of cars typically appear as spotlights. A hardware module called SpotlightEngine scans the image and filters out the spotlights. It takes a grayscale image as input and outputs a binary (i.e. black and white) bitmap. As discussed in section I, previous authors used thresholding to find taillights. However, additional information, such as color or physical distance, was used to determine the threshold value. Also, thresholding does not consider the shape of an object, so it cannot reliably distinguish other bright objects, such as lane

markers, reflecting traffic signs, or the bright tunnel exit. The proposed algorithm applies a simple shape filter to the image and adapts dynamically to the ambient light. A mask defining two sets of pixels P_S and P_F is applied to each pixel of the image, as shown in fig. 4C and 4D. Two possible masks are shown in fig. 4A and 4B. They define the two pixel sets relative to the current pixel (CP). If all pixels in P_F are darker than all pixels in P_S , the CP is a spotlight pixel. That condition is expressed as follows:

$$\min(\text{lum}(P_S)) > \max(\text{lum}(P_F)) + \text{threshold} \quad (1)$$

With $\text{lum}()$ being the set of luminances of the pixels in the set. For a grayscale image, the luminance equals the pixel value. The value for *threshold* can be set constant and is relatively uncritical. Fig. 4C shows the mask applied to the center pixel of a taillight, causing eqn. (1) to evaluate true. In Fig. 4D, the mask is applied to a lane marker, which covers both sets defined by the mask. Therefore, even though all pixels in P_S are bright, eqn. (1) evaluates false here.

The shapes and sizes of P_S and P_F determine what kinds of spotlights will be found. A spotlight has to cover at least the area of P_S in order to be detectable. At the same time, it must fit into the frame given by P_F . Thus, if P_S is chosen to be almost as big as P_F , a very narrow search is done for lights of that specific size and shape (fig. 4B). On the other hand, if P_S is only one pixel, and P_F is relatively large, a very wide variety of lights will match the mask (fig. 4A). In this work, the latter variant is chosen as it requires only one search of the image. The resulting output image of the SpotlightEngine is shown on the right side of fig. 1. Alternatively, several searches for more specific sizes could be done and combined.

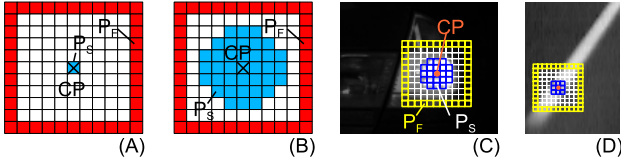


Fig. 4. Two possible masks, defining pixel sets P_F and P_S . In C and D, the mask is applied to different pixels of an input image.

B. Labeling regions

The operation described so far creates a binary bitmap of 0- (black) and 1-pixels (white) in which a light consists of one or more pixels, depending on its size. Subsequent modules require a list of lights, which is extracted from the binary image by the LabelingEngine module. It scans the bitmap for white pixels and identifies separated regions of connected 1-pixels, each identified by a distinct label. A label needs to be assigned to each 1-pixel, whereas 0-pixels are ignored, because they are the background. All 1-pixels connected to each other need to be assigned the same label. Different labels must be assigned to 1-pixels without such a connection. Each pixel is directly connected to eight neighboring pixels at its corners and edges. One region or label corresponds to one light,

regardless of how many pixels the light covers. This labeling process is very common in computer vision and described, for instance, in [7].

C. Determination of static lights

At this point, a list of spotlights has been extracted from the input image. For each light, the following properties are available:

- Bounding rectangle
- Position in the image (coordinates of the center of the bounding rectangle)
- Total brightness (defined as $\sum \text{lum}(P_S)$)
- Number of pixels (size of region)

These lights stem from different sources, such as vehicles, tunnel lighting, lit road signs and reflections. Some light sources are static, i.e. they do not move relative to the road. For the detection of moving vehicles, these lights are not relevant, so they should be filtered out. Due to the movement of the on-board camera, static lights do move in the two-dimensional video - depending on their position in the three-dimensional space and on the velocity of the own car. As shown here, the direction of the light's motion vector can be used to determine static lights, whereas the motion speed is irrelevant.

The two-dimensional camera image is a projection of the three-dimensional environment. In that projection, parallel straight lines seem to intersect in an infinite distance, at the so-called vanishing point. A straight road or tunnel can be described by such parallel lines. While the own car is driving straight ahead, every static object along the road seems to appear at the vanishing point and travel to the outside of the image, along straight lines. Therefore, the motion vector of a static object always points away from the vanishing point. This point is fixed as long as the road is fairly straight, which is generally the case for a freeway. Objects that are moving relative to the road exhibit a different motion vector, depending on their speed relative to the own car.

In order to find motion vectors for each spotlight, light tracking is used. For each light in the current frame, a close-by light is searched in the previous frame. The motion over several frames is monitored in order to obtain smooth motion vectors. If the motion vector points away from the vanishing point, the corresponding light is marked static and no longer considered by the subsequent routines.

D. Finding light pairs

The previous routine has identified a list of spotlights moving relative to the road. Next, pairs of spotlights are identified. A pair might correspond to the two taillights of a car or truck. The labeling module ensures that list entries are already sorted by the y-position of one arbitrary pixel within each light. Even though this sortation is not perfect, the following processing modules can rely on a sorted list, simplifying future steps.

In order to find pairs of lights, all their possible two-combinations are considered. If there are N lights in the image, initially $\frac{N!}{2! \cdot (N-2)!} = \binom{N}{2}$ potential pairs are formed.

However, pairs that are too far apart in y -direction can immediately be removed, leaving less than $2N$ pairs in all practical cases studied. For each potential pair of lights, several properties are considered and scored separately:

- Distance of lights, y -component
Both taillights of a car are expected to be on the same height on a flat road. Consequently, they should appear at the same y -coordinate in the image. If that is the case, full score is given.
- Distance of lights, x -component
A vehicle that is far away from the camera appears more in the top of the image than a closer vehicle. Thus, based on the y -position (height) of a pair, its expected width can be calculated. For observed distances close to the expected value, full score is given. A tolerance is allowed to account for small changes in road inclination as well as for tolerances in the actual vehicle width.
- Ratio of brightness
Both lights in the pair are expected to exhibit a similar brightness. This property is rated with full score if the brightness ratio is close to 1.
- Existence of additional light in between the pair
Two cars driving next to each other are seen as four similar spotlights in a row. In that case, the two outer lights of the four spotlights could be considered as one candidate pair, while there are really two smaller pairs. A zero score is given for this property if there are additional spotlights in between the current pair, suggesting that the pair should be subdivided.
- License plate visible
Most license plates are located exactly between the taillights and are dimly lit. The module PlateSearch performs a search for a luminance pattern as shown in fig. 5 (right) on the pixels between the two taillights. That pattern was constructed from an idealized rear view of a car, shown in fig. 5 (left). It accounts for two bright regions caused by the taillights at both ends and another one in the middle, caused by the license plate illumination (L). In between, dark pixels (D) are expected. The transition between D and L can be within a certain area and will be determined dynamically from the actual image. For better performance, a simplified one-dimensional pattern was chosen and applied at several different horizontal lines in the proximity of the pair. The grayscale pixels are converted to a binary format before matching against the pattern is attempted, using a threshold based on the ambient brightness.

A maximum score of 1 is given if a property value is within a certain range. In case the value is outside that range, its score is reduced gradually, but not below 0. The ideal range of each value is determined from an ideal pattern of a car back side and by inspection from vehicles seen in test videos. An overall score is calculated for each light pair from the individual scores by averaging. The overall score is a measure for the probability that a potential pair actually corresponds

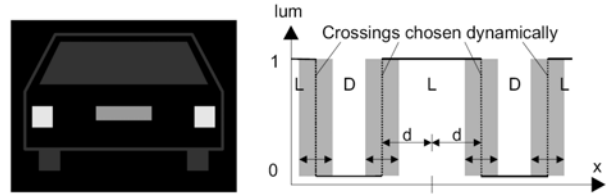


Fig. 5. Idealized rear view of a car (left) and 1D brightness pattern used to search for license plates (right)

to a vehicle. Each property discussed above delivers an equal fraction to the overall score. Potential pairs ranked above an adjustable threshold are selected as actual pairs that most likely represent a vehicle. Each light might have been associated with several potential pairs, but it may only belong to one actual pair.

Real vehicles are expected to remain at a similar position between two consecutive frames. This property can be used to differentiate between pairs representing real objects and pairs resulting only from random light reflections. Therefore, pairs are tracked over several frames, using their position and size. If a pair was found in several previous frames, it exhibits a high temporal continuity, as expected from a real car.

III. SYSTEM IMPLEMENTATION

The system was implemented on an ML310 evaluation board with a Xilinx Virtex-II Pro FPGA, suitable for HW/SW co-designs. That FPGA also features two embedded 300 MHz PowerPC CPU cores, one of which was used to run an embedded Linux operating system. The software modules described above run within a user mode process. A kernel driver was developed to provide access to the SpotlightEngine and LabelingEngine. Table I shows the hardware resources required by the two hardware modules, clocked at 100 MHz. Additional hardware resources are required to create a system capable to boot a Linux operating system. A Processor Local Bus (PLB) connects the CPU and all hardware components.

RESOURCE	USED	AVAILABLE	PORTION OF FPGA
Slices	2816	13696	20%
Slice Flip Flops	2344	27392	8%
4 input LUTs	5057	27392	18%
BRAMs	19	136	13%

Maximum Frequency: 103.2MHz (Minimum period: 9.689ns)

TABLE I
HARDWARE REQUIREMENTS FOR THE COPROCESSORS
(SPOTLIGHTENGINE & LABELINGENGINE)

A. HW/SW partitioning

One of the main goals of the presented work was to design a system that is well suitable for hardware/software partitioning and thus can take advantage of a hardware acceleration module. As discussed in section II, the system consists of two distinct types of modules - those operating on a complete bitmap, and those working on a list of features. The bitmap

modules apply the same relatively simple and fixed pixel operation to a large amount of data, namely to each individual pixel of the image. These characteristics are ideal for hardware implementation, because simple operations translate to a low number of logic elements. At the same time, there is a great benefit in execution time over a software solution because of the large input data set. Thus, the first modules in the data flow are implemented as hardware modules in the proposed system. (Spotlight- & LabelingEngine, the first row in fig. 3.)

The remaining modules work on a list of data originally assembled by the hardware, while the input image is not accessed at all. Operations performed by these modules are more complex and highly dependent on the input data. Consequently, a software solution was chosen here. Due to the small size of input data, the execution time remains low. In the design of the presented algorithm a clear interface in the data flow between hardware and software can be defined: Partitioning happens at the point where the large input bitmap is transformed into a small list of regions. The module PlateSearch is a slight exception, as it also needs to access a small portion of the original input bitmap directly. A hardware implementation of this module might be desirable, however this will only result in a small execution time benefit (some 100 μ s) compared to the software solution.

B. Coprocessor implementation

The graphic hardware coprocessors, SpotlightEngine and LabelingEngine, take advantage of a flexible image processing framework which was introduced previously in [4]. It utilizes direct memory access to read the input image from memory and write back the output image. Each pixel is individually fed to a processing module, beginning at the top-left pixel and continuing horizontally. The processing module only performs the main pixel operation and does not have to take care of border effects, memory address calculations or bus stalls. Part of the image is cached in a ring buffer, providing easy access to pixels in proximity to the currently processed pixel (neighborhood). One column of pixels can be read out in parallel from the cache within one clock cycle.

The SpotlightEngine evaluates for each pixel:

$$lum(CP) = \begin{cases} \min(lum(P_S)) - & \text{if result} > 0 \\ \max(lum(P_F)) & \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

By comparing that value with a threshold, the condition given earlier in eqn. (1) is evaluated. The frame defining P_F (see fig. 4) was selected to be 11×11 pixels wide, so $|P_F| = 40$. Two clock cycles are required to read the left and the right pixel columns from the neighborhood cache, which consequently needs to store at least 11 lines around the current pixel. A shift register holds the values of the top and bottom row of P_F . Finding $\max(lum(P_F))$ requires the comparison of 40 values, each 8 bits wide. Multiple comparators are connected in a pipeline structure, each featuring two 8 bit wide inputs. One stage of the pipeline reduces the number of values by a

factor of two, so there are $\lceil \log_2(|P_F|) \rceil$ stages needed to find the maximum. The latency of the comparator is equal to the number of stages, but during each clock, new values can be fed in. The set of pixels P_S is only 2 pixels big, so there is only one comparator needed to find $\min(lum(P_S))$.

The LabelingEngine receives a binary image from memory and does not create an output bitmap. Instead, it creates a list of pixel regions saved in internal FPGA memory (BRAM). That list contains already the properties described in II-A and can later be read out by the CPU. In some cases, the list describes trees of equivalent regions, which are resolved in software efficiently. This means, hardware/software partitioning happens right within the labeling algorithm.

IV. EXPERIMENTAL RESULTS

It is crucial for the presented system to work in real-time. This means that processing for each image of the video (frame) needs to finish before the next one becomes available. Dropping frames is not acceptable, as this would delay the reaction time and corrupt motion data. Given a framerate of 25 frames/s, processing time may not exceed 40 ms. Runtime analysis and worst-case tests were performed to show that the presented system can meet this requirement. For all tests, an image size of 384 by 288 pixels was used.

For each of the two hardware modules, there is an easy way to calculate the expected runtime theoretically. The design runs at 100 MHz and requires in general 2 clock cycles per pixel. A pixel cache has to be filled with default values before processing a new image line. Also, at the end of each line, about 20 clock cycles are required for initialization. This yields the following expression for the runtime of a hardware module:

$$t_{HW} = \frac{[r \cdot (w + 2r) + h \cdot (2w + 20)]}{f_{HW}} \quad (3)$$

With h , w representing the image size and r being the number of preloaded lines. For the given image size and $r = 5$, as required for an 11×11 frame, a runtime of $t_{HW} = 2.29$ ms is obtained. Practical measurements showed times only slightly above that value. However, due to inefficiencies in the bus transport, the runtime increases by about 50% if an output image is written to the memory, as done by SpotlightEngine.

MODULE	MINIMUM	MAXIMUM
SpotlightEngine (FPGA)	3.44 ms	3.52 ms
LabelingEngine (FPGA)	2.37 ms	2.40 ms
Spotlight (PowerPC)	-	607 ms
Spotlight (Pentium 4)	-	26 ms

TABLE II
MEASURED RUNTIME FOR HARDWARE MODULES AND THEIR SOFTWARE IMPLEMENTATIONS

Table II lists the results of runtime measurements for a typical input image. As a comparison, the runtime of a software implementation of SpotlightEngine is also given. It was tested with the 300 Mhz PowerPC processor on the Xilinx ML310 board as well as on a 3 GHz Intel Pentium 4 system. The execution time of the software parts in the

system is highly dependent on the number of spotlights found in the image. A theoretical analysis is impractical, so several different images with up to 50 spotlights were used to measure the actual runtime. Altogether, the software modules take between 0.7 *ms* and 6 *ms* to execute on the embedded PowerPC on the ML310. For comparison, the runtime on a 3 GHz Pentium 4 is between 0.3 *ms* and 0.6 *ms*. Real-world images are in general less complex than the worst image tested, so an upper boundary of 12 *ms* for the total execution time (hardware + software) can be given. This means that there are resources remaining in the system that can, for instance, be used to process a larger input image. As an example, a bigger input image of 640 × 480 pixels would scale the hardware execution time by a factor of 2.8, whereas the runtime for the software would not be affected. With a calculated hardware runtime of 16.7 *ms*, the overall runtime of the system would be 23 *ms* in the worst case. Even more resources can be made available by having coprocessor and main CPU run in parallel. Also, both hardware modules can be combined in one, if the output of the SpotlightEngine is not needed. This effectively halves the total coprocessor runtime.

The TaillightEngine was evaluated using 20 second, unpreprocessed video sequences recorded during tunnel drivings. Table III shows the detection statistics for one of those videos, normalized to the total number of vehicles present in the video (N_{TOT}). The system also assigns a certainty value to each detected object, based on its temporal continuity. All those objects marked with a high certainty were in fact cars and thus detected correctly. Altogether, 73% of all vehicles N_{TOT} were detected. Some objects, such as reflections, were erroneously detected as cars, but their number is relatively low compared to N_{TOT} (1:20). The number of non-detected vehicles seems pretty high at 27%. Part of it can be attributed to objects that were far away from the camera, only producing an unclear image. However, about 10% of all vehicles N_{TOT} were not detected because of an activated indicator at the corresponding vehicle. This event was not considered in the design of the TaillightEngine, but it deranges the search for light pairs severely.

RESULT	CERTAINTY	PERCENTAGE
Vehicle detected	high	57%
	low	16%
Vehicle not detected	-	27%

TABLE III
VEHICLE DETECTION STATISTICS

V. CONCLUSION AND FURTHER WORK

In this paper a flexible HW/SW co-design architecture to detect cars based on their taillights in real-time was presented. Computationally intensive image processing operations were performed on a custom image coprocessor. The remaining software part could thus run on an embedded processor with only limited computational resources. An FPGA with two embedded PowerPC cores was used to implement the

system, enabling the possibility to dynamically reconfigure the hardware coprocessor for different driving environments.

The presented work should be regarded as a first demonstration system, which does not cover all situations that can be encountered on the road. For instance, vehicles with solely one taillight, such as motorbikes, cannot be detected. Vehicles that are standing still or using their indicators are not detected in some cases. Non-moving cars could be detected by first searching for light pairs instead of discarding the corresponding lights due to their motion vectors. Certain changes in the environment, such as curved or hilly roads, will break some of the assumptions made and degrade the detection performance. In an improved version a dynamic adaption to all of these changes could be implemented. Also additional types of surroundings such as urban traffic need to be considered in order to design a more robust system. For different environments, such as daylight driving, the hardware accelerator could be exchanged by another, more suitable one. This can be accomplished by the dynamic partial reconfiguration capabilities of Xilinx Virtex devices as described in [4]. Finally the system should be ported to a real car to demonstrate the benefits of the proposed system.

VI. ACKNOWLEDGEMENTS

This work is supported by the German Research Foundation DFG (Deutsche Forschungsgemeinschaft) in the focus program No. SPP1148. We also want to thank Xilinx for providing development boards and BMW for providing test video sequences.

REFERENCES

- [1] M. Betke, E. Haritaoglu, and L. S. Davis, "Real-time multiple vehicle detection and tracking from a moving vehicle," *Machine Vision and Applications*, vol. 12, no. 2, pp. 69–83, August 2000.
- [2] I. Cabani, G. Toulminet, and A. Bensrhair, "Color-based detection of vehicle lights," in *IEEE Proceedings of Intelligent Vehicles Symposium 2005, Las Vegas, USA*. IEEE Computer Society, 2005, pp. 278–283.
- [3] M.-Y. Chern and P.-C. Hou, "The lane recognition and vehicle detection at night for a camera-assisted car on highway," in *Proceedings of the 2003 IEEE International Conference on Robotics and Automation, ICRA 2003, Taipei, Taiwan*.
- [4] C. Claus, J. Zeppenfeld, F. Müller, and W. Stechele, "Using partial-runtime reconfigurable hardware to accelerate video processing in driver assistance systems," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe, Nice, France, 16th-20th April 2007*, pp. 498–503.
- [5] R. Cucchiara and M. Piccardi, "Vehicle detection under day and night illumination," in *Proceedings of ISCS-IIA99, Special Session on Vehicle Traffic and Surveillance, Genoa, Italy, 1999*. [Online]. Available: <http://citeseer.ist.psu.edu/cucchiara99vehicle.html>
- [6] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.
- [7] R. M. Haralick and L. G. Shapiro, *Computer and Robot Vision*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1992, vol. 1.
- [8] S. Kim, S.-Y. Oh, J. Kang, Y. Ryu, K. Kim, S.-C. Park, and K. Park, "Front and rear vehicle detection and tracking in the day and night times using vision and sonar sensor fusion," August 2007, pp. 2173–2178.
- [9] W. Kubinger, S. Borbely, H. Hemetsberger, and R. Isaacs, "Platform for evaluation of embedded computer vision algorithms for automotive applications," in *Proceedings of 13th European Signal Processing Conference (EUSIPCO 2005), September 4-8, 2005, Antalya, Turkey*, pp. 130–133, September 2005.