Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2006, Article ID 82564, Pages 1–8
DOI 10.1155/ES/2006/82564

# A Dynamic Reconfigurable Hardware/Software Architecture for Object Tracking in Video Streams

**Felix Mühlbauer and Christophe Bobda**

*Department of Computer Sciences, University of Kaiserslautern, Gottlieb-Daimler Street 48, 67653 Kaiserslautern, Germany*

Received 15 December 2005; Revised 8 June 2006; Accepted 11 June 2006

This paper presents the design and implementation of a feature tracker on an embedded reconfigurable hardware system. Contrary to other works, the focus here is on the efficient hardware/software partitioning of the feature tracker algorithm, a viable data flow management, as well as an efficient use of memory and processor features. The implementation is done on a Xilinx Spartan 3 evaluation board and the results provided show the superiority of our implementation compared to the other works.

## 1. INTRODUCTION/MOTIVATION

Pervasive and ubiquitous computing is gaining more and more in popularity. Boosted by advances in broadband communication and processing systems, *computer anytime and anywhere* is slowly but surely becoming a reality. Ubiquitous and pervasive computing usually involve a set of distributed sensing and computing nodes geographically located at different sites. Each node collects a given amount of raw data that is exchanged with other nodes in the system. One of the main requirements here is that raw data collected by sensors at a given geographic location by a given system or part of it should be processed by a corresponding module at that location. Therefore, the communication between different nodes is reduced. Only the results of computations at different sites—which are mostly sensor data interpretation with a reduced amount compare to the raw data—have to be sent to other nodes.

The constraints imposed on pervasive and ubiquitous computing systems—which are mostly untethered—lead to a very challenging design process. Large amount of data must be computed in real time whilst at the same time maintaining a very low power consumption for the whole system. Furthermore, the system must be able to adapt to changing environmental and operational conditions. None of the processors commonly used in embedded systems like DSPs, ASICs or general purpose processors can provide the features alone (performance, low power, and adaptivity) that are required in ubiquitous and pervasive systems.

The last decade has experienced an increasing interest in deployment of FPGAs in embedded systems. With the progress in manufacturing technology, FPGAs have become 40 times faster and consume 50 times less power with an increase of 200 fold in their capacity (number of available logic cells) whilst at the same time becoming 500 times cheaper in less than 15 years. According to several studies, this trend is going to be maintained at least in medium term. It is increasingly possible to implement a complete system on-chip solution using the lowest cost FPGA device. Furthermore, the partial reconfiguration capability of FPGAs allows for the realization of adaptivity, thus making FPGAs more and more attractive for pervasive and ubiquitous systems [1].

A main advantage of these programmable logic devices is the ability to realize parallel processing hardware. Especially image processing algorithms are inherently parallel and thus FPGAs can be used to develope highly efficient solutions. In many systems, for example, in surveillance systems, video data is captured by modules and sent to other modules for further processing. The processing task can be, for example, the detection of movement or the detection and tracking of suspect objects in a given environment that is monitored by a camera.

A system on chip is usually made up of a processor connected to a set of peripherals and dedicated hardware modules via a bus system. The bus system is mastered by the processor to access peripherals and collect data to be processed. In video streaming applications in which large amount of data must be computed in real time while streaming through different computational blocks, a traditional system on chip in which all the data transfer between different modules is done on the bus is no longer viable. The exclusive use of the bus at a given time by one master hinders simultaneous access to data by different modules.

In this work, we present a modular implementation of a feature tracker for video streams in an FPGA. The architecture is made up of a system on chip in which a processor and dedicated hardware accelerator modules cohabit. Contrary to traditional system on chip, we do not rely only on a bus for communication. A set of dedicated line and protocols allow for a real-time computation of data while they are streamed.

The implementation is done on a Xilinx evaluation board featuring a Spartan 3 FPGA and on a ML310 board with a Virtex2 Pro.

The remainder of this paper is organized as follows. Section 2 introduces the feature tracking and presents algorithms used. In Section 3, we present an overview of the related work. Section 4 presents the design of the feature tracking system on an FPGA. There we will discuss the main design decision. The adaptivity of the system is also discussed in this section. In Section 5, we present the implementation results for two platforms. Finally, Section 6 concludes the paper and gives some indication of future work.

## 2. OBJECT TRACKING IN VIDEO STREAMS

For object tracking purposes often feature trackers are used, which analyze image sequences and detect motion. For this purpose small windows, called features, with certain attributes are selected and then attempts are made to find them in the next frame. Such attributes can, for example, be some measure of texturedness or cornerness, like a high standard deviation in the spatial intensity profile, the presence of zero crossings of the Laplacian of the image, or a simple corner. Yet, apparently promising features can be useless or even harmful for tracking, if they do not correspond to a point in the real world. This happens with hotspots (a reflection of a highlight on a glossy surface), mirroring, or in the case of straddling a depth discontinuity. Conversely, useful features can be lost if they leave the field of vision by obstruction or by moving out of the image. The well-known KLT-tracker[1] is often used as a base for further development.

The same case is with the following algorithm which is the approach of Shi and Tomasi [4]. According to them, the pure translation model is not an adequate model for image motion when measuring dissimilarity between the features. They provide experimental evidence for this and introduce an extended model considering affine image changes.

Image motion can be regarded as a change in image intensity $I$ of a given point $(x, y)$ at time $t + \tau$:

$$I(x, y, t + \tau) = I(x - \xi(x, y, t, \tau), y - \eta(x, y, t, \tau), t). \quad (1)$$

[1] Kanade, Lucas, and Tomasi [2, 3].

The time-dependent functions $\xi$ and $\eta$ provide the displacement in $x$ and $y$ directions. So, $\delta = (\xi, \eta)$ defines the amount of motion and, respectively, the displacement of the point at $\mathbf{x} = (x, y)$. Even within the small windows used for feature tracking, $\delta$ varies and a certain displacement vector does not exist. A more efficient way is to consider the affine motion model:

$$\delta = D\mathbf{x} + \mathbf{d}, \quad (2)$$

where

$$D = \begin{bmatrix} d_{xx} & d_{xy} \\ d_{yx} & d_{yy} \end{bmatrix} \quad (3)$$

is a deformation matrix and $\mathbf{d}$ is the translation of the feature window's center. Applying this to the intensity relation leads to

$$J(A\mathbf{x} + \mathbf{d}) = I(\mathbf{x}), \quad \text{where } A = \mathbf{Id} + D. \quad (4)$$

This means that for any two given images $I$ and $J$ six parameters must be calculated. The quality of the results depends on the size of the feature window, the texture of the image within it, and the amount of motion (camera or object) between frames. Smaller features result in less reliable values for $D$, because only few image changes are considered, but are generally preferable because they are less likely to straddle a depth discontinuity.

Because of image noise and because the affine motion model is not perfect, (4) is in general not satisfyingly exact enough. To solve this problem the following equation is used to reduce the minimal error to a sensible value for $A$ and $\mathbf{d}$:

$$\epsilon = \iint_W \left[J(A\mathbf{x} + \mathbf{d}) - I(\mathbf{x})\right]^2 w(\mathbf{x})d\mathbf{x}, \quad (5)$$

where $W$ is the feature window and $w(\mathbf{x})$ a weighting function, which comes to 1 in the simplest case. Alternatively, a Gaussian-like function can be used to emphasize the center area of the window.

The problem can be converted to a linear $6 \times 6$ system and $D$ and $\mathbf{d}$ can be found with an iterative Newton-Raphson style minimization (see [5]).

Shi and Tomasi use the pure translation model for tracking and affine motion for comparing features between the first and the current frames in order to monitor quality.

This algorithm has its advantages and drawbacks. First, it works at subpixel precision. Feature windows in frames will never be identical because of image noise, intensity changes, and other interfering factors. Thus translation estimation cannot be absolutely accurate, the errors accumulate and feature windows drift from their actual positions. In [6] Zinßer et al. take care of this problem and also deal with illumination compensation. Another advantage of their concept is the detection of distorted and rotated features, which is achieved by the affine motion model.

The algorithm excessively uses floating point operations causing high resource costs. Also, only small translations can be estimated which requires slow moving objects in the observed scene or high frame rates of the incoming video stream, which results in high resource consumption, too. We want to introduce another procedure for tracking features which is much more suitable for an implementation on FPGAs.

The following algorithm refers to [7]. In contrast to the KLT-tracker, features (in this case: Harris corners[2]) are detected in *each* frame and only comparisons between *features* are permitted. For that purpose each position in the current frame, or to be more precise a feature window with this position (feature point) in the middle, is assessed: the derivatives $I_x$ and $I_y$ are computed by horizontal and vertical filters in the form $[-1\ \ 0\ \ 1]$. Next, the products $I_xI_x$, $I_xI_y$ and $I_yI_y$ are separately convolved with the binomial filter $[1\ \ 4\ \ 6\ \ 4\ \ 1]$, again horizontally and vertically, to produce the values $G_{xx}$, $G_{xy}$, and $G_{yy}$. Now determinant $d = G_{xx}G_{yy} - {G_{xy}}^2$, trace $t = G_{xx} + G_{yy}$, and finally the strength $s = d - kt^2$ with $k = 0.06$ of the corner response are calculated (see Figure 1(a), white areas represent high values of $s$).

To define the actual feature points, nonmax suppression is used: each pixel for which the corner response is strongest, considering a $5\times5$ neighborhood, is declared a feature point. This method is an alternative to using a global threshold for the strength of the corner response (see Figure 1(b), where features are marked).

For matching, each feature is compared with all features of the next frame which reside within a certain distance of the original window. To achieve this, normalized correlation is used. The distance can be adjusted to performance requirements. Because Harris corners happen to be in the corner of their feature window, which impedes correct matching, a bigger window of $11 \times 11$ pixels ($n = 121$) is used instead. Many comparisons have to be made but fortunately some values can be precomputed:

$$\begin{aligned} A &= \sum I, \\ B &= \sum I^2, \\ C &= \frac{1}{\sqrt{nB - A^2}}. \end{aligned} \tag{6}$$

With the scalar product

$$D = \sum I_1 I_2 \tag{7}$$

of the two features to be compared, the normalized correlation is

$$\epsilon = (nD - A_1A_2)C_1C_2. \tag{8}$$

[2] The Harris corner detector computes the locally averaged moment matrix computed from the image gradients, and then combines the eigenvalues of the moment matrix to compute a corner *strength*, of which maximum values indicate the corner positions.

To decide which matches to accept, a mutual consistency check is used: all features are compared under several different aspects. For each feature, the preferred counter part, which produces the highest value of $\epsilon$, is saved. Finally only features which mutually fit each other are valid matches.

This algorithm is not equipped with a drift detection. In addition, because of the irregular input data, as mentioned above, the features, which are detected for every frame, will vary and matching is partly impeded. On the other hand, translations over larger distances can be estimated while only low frame rates are needed and calculations are simple, compared to the first algorithm which excessively uses floating point operations. Further, the complete feature detection and selection process is highly parallelizable and additionally can be computed using integer operations only. These are very good preconditions for hardware/software co-design implementation on an FPGA.

## 3. RELATED WORK

Feature tracking is usually implemented in the context of autonomous navigation where objects have been detected and tracked by a given entity. Most of the available systems are implemented as a pure software solution [4–7]. Usually a personal computer is mounted on a robot to perform the required computation. Acceleration of feature tracking on parallel computers is considered in [8–10]. The MIMD is considered in [9] while [10] implements the SIMD paradigm. The target platform for the MIMD implementation is a MasPar MP-1 with a $128\times128$ mesh of processing elements while the SIMD targets the Intel Paragon and the IBM SP2 platforms. The implementation of [8] is used more often for simulation purposes and is done on an adaptive grid machine called GrACE.

While such solutions can be useful for experimental purposes and for proof of concept, it is not applicable to real autonomous systems. Parallel machines, for example, cannot be used in an embedded environment because the power consumption of workstations mounted on a robot will allow the robot only to drive a few meters. Moreover, the size of the robot must be sufficiently large to carry the PC, thus leading to a very large system.

Some effort to tackle the aforementioned problem has been done in [11–13]. In [12] the goal is to have a real-time implementation of the feature tracking using a hardware platform. The target system is a C4x board featuring eight Texas instrument processors C4x running at 50 MHz. Each processor is assigned a specific task. One processor grabs the frame, two processors perform feature selection, and one processor performs motion estimation. Feature tracking is done by three processors and the rendering is done by one processor. The system is able to process 0.8 frames per second for feature detection (100 features) and 4 frames per second for feature tracking, leading to an overall performance of 0.8 frames per second. Although the size of this system as well as its power consumption remains low compared to a software solution, it is still far from being suitable to be used in a mobile system.

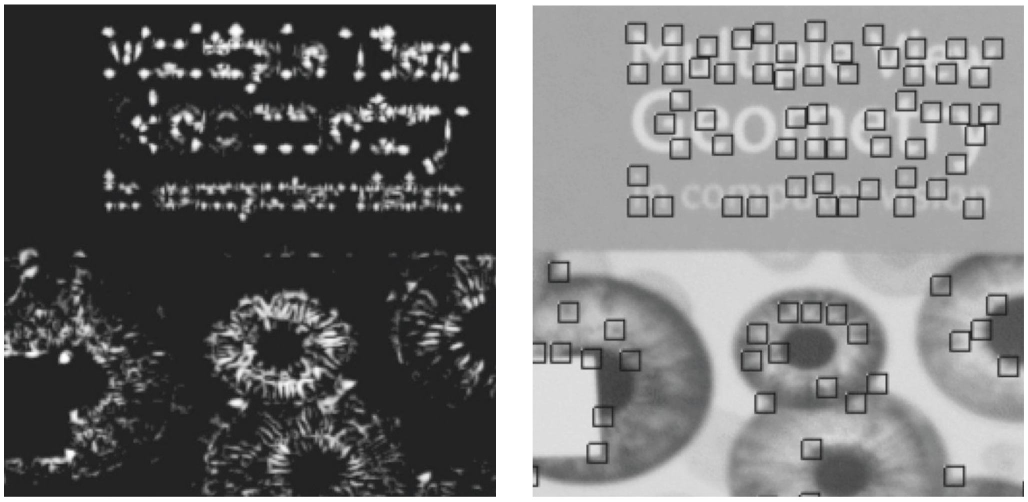

(a) (b)

Figure 1: Feature detection and selection.

Some recent works [11, 13] have targeted FPGAs for an efficient implementation of feature tracking. In [11] features are selected in an FPGA mounted on a PCI-Board, which is embedded in a workstation. Not only cannot this solution be used in small mobile environments, it also presents the drawback that the processor, a 3 GHz Pentium, must be used all the time for data transfer between the FPGA and the processor.

The system in [13] is a more compact hardware/software system. The software part is implemented on a PowerPC processor that is attached to an FPGA in which the hardware part is implemented. The FPGA is in charge of the image enhancement that is done using a high pass filtering process. The implementation uses a sliding window that is used to capture the neighborhood of an incoming pixel. The latter is then used to compute the enhanced value of the pixel that is stored in a memory shared by the PowerPC and the FPGA. The adaptive process is done via the modification of the filter parameters as well as the threshold parameter for the number of features to be selected. Because the single available memory can be accessed only by one module (processor or FPGA) at a time, the streaming computation process will be delayed leading to a decrease in the number of frames that may be processed in a second. The FPGA is used in this system as an ASIC, since no reconfiguration is done. Because the structure of filters varies according to the algorithms used, a simple change in the filter coefficient is not sufficient to replace a filter in the FPGA. The Sobel operator, for example, is two dimensional while the Laplace is only one dimensional. Therefore, a Laplace filter cannot become a Sobel just by replacing the coefficients. The configuration of the FPGA can be used to replace the complete filter structure.

This work presents a better use of a single chip to implement an embedded adaptive hardware/software solution to feature tracking. The system is optimized to perform computations on all the streamed frames without delay. We also exploit the possibilities to dynamically extend the instruction set of the embedded processor by binding an accelerator directly to the processor. Adaptivity is done by means of configuration rather than by parameter modifications as is the case in [13].

## 4. DESIGN OF A HARDWARE/SOFTWARE SOLUTION FOR FPGA

This section describes our implementation of the chosen feature tracking algorithm. The data flow from the incoming images to the positions of image movements looks like this: video in → feature detection → feature selection → feature tracking → further processing.

The feature detection is highly parallelizable and can be implemented completely in hardware (see Figure 2). A compilation of five convolve filters (for $I_x$, $I_y$, $G_{xx}$, $G_{xy}$, and $G_{yy}$) and simple arithmetic operations is used. Usually convolving is realized using a sliding window, whose size can be $3 \times 3$, $5 \times 5$, and so on. In case of a $3 \times 3$ window, incoming pixel data must fill up two line buffers before the first calculation is possible. The latency results in $2\ lines + 2\ pixels + 1$ clock cycles. The corner strength can be computed by add, subtract, and multiply units. Down shifting provisional results prevent arithmetical overflows. The factor $k = 0.06$ in $s = d - kt^2$ can be approximated by shifting by 4 ($\widehat{=} k = 0.0625$). The FPGA used is equipped with single clock cycle multipliers. Thus, the corner strength calculation needs only three further clock cycles to be completely computed while using only integer numbers.

For feature selection nonmax suppression is used, which is realized by a sliding window, too. Each value within the

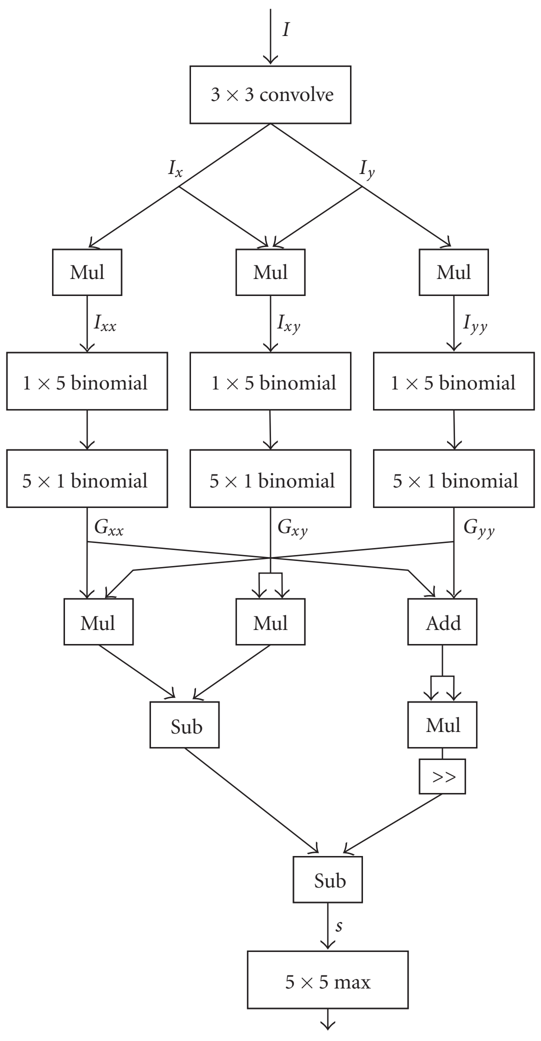


Figure 2: Logic for feature detection.

window is compared with the center value. If it is the maximum the window position is declared a feature point.

For a system on-chip layout, these preliminary ideas allow a hardware/software partitioning as shown in Figure 3(a). The feature detection and selection is completely implemented in a dedicated hardware module (FT) and feature tracking is done in software by the Xilinx MicroBlaze processor. The video frames are captured by the videoin module and stored in the SRAM. The RS232 module is used for debugging purposes.

Considering the data transfer, there is communication between video input module and feature module, between feature module and memory in order to access the current frame and store the selected features, and between processor and memory in order to continue with tracking these features. All transactions simultaneously utilize the bus, which is in general only designed for low peripheral load. The amount of data produced by video streams is very high and clutters up the bus. That means that this solution is not fit to process data in real time.

### 4.1. *Efficient hardware/software partitioning*

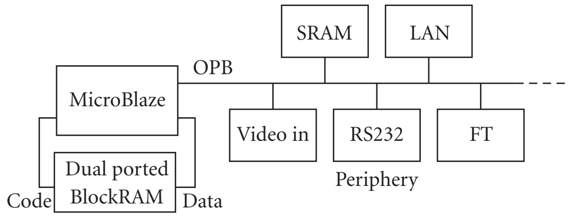
By rearranging the components while accounting for the data flow, performance can be improved. Our architecture is shown in Figure 3(b). The videoin module sends image sequences to the feature module, which stores image data and selected features in the memory. A dual ported memory is used that can be accessed from two different clients and clock domains. The feature module is, furthermore, connected to the bus to exchange information with the processor. This information consists of controlling instructions like "start" and "stop," but also of parameters like the base address of image data in the memory or parameters which influence the processing. The BlockRAM is the main memory of the processor and holds data like variables and heap of the application running on it. A certain area of the memory is reserved for image and feature data by the application. To notify the feature module about the location of this area the base address is transmitted by the processor to the module over the bus.

### 4.2. *Dynamic reconfiguration increases flexibility*

The Xilinx FPGAs allow partial reconfiguration of each individual column. The Erlangen slot machine (ESM) [14] is an architecture that exploits this feature. Its new concept allows an unrestricted relocation of modules on the device. The FPGA is logically divided into *slots* which can be reprogrammed independently. Via a programmable crossbar each module can, regardless of placement, communicate with its peripherals and also with other modules. Memory banks are vertically attached to each slot, providing enough memory space to store temporary data. In streaming applications, this memory can also be used for shared memory communication between neighboring modules. Smaller data chunks can be transferred either by placing (dual ported) BlockRAM between them or via a reconfigurable multiple bus (RMB).

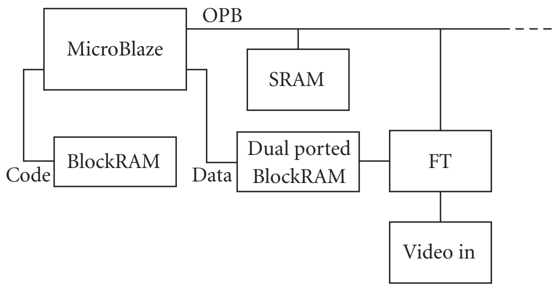
Figure 4 shows the possible placement of our feature tracker on the ESM platform. The data flow was already described above. Using multiple memory banks, a technique called double buffering can further increase performance: image frames are filled alternately into two banks by the videoin module. The feature module reads out data but always from the respectively other bank. Hence, in contrast to a single memory architecture, no bottleneck will occur while accessing the memory.

Reconfiguration highly increases flexibility of the feature tracker. This means that, for example, the source of incoming image stream can simply be an analog video input as well as a firewire or LAN connection. By reconfiguration the input module can be replaced by an appropriate one. Image prefiltering, like illumination compensation or the Gaussian function to smooth image noise, increases the quality of the tracking results. In addition, feature detection and selection processes can be altered or exchanged to adapt to the system environment. The tracker unit can use dedicated helper hardware, fore example, to speed up the comparison of features (see next subsection). In contrast to works like [13] we



OPB = on-chip periphery bus
FT = feature detection and selection

(a) Suboptimal system on chip



OPB = on-chip periphery bus
FT = feature detection and selection

(b) Efficient hardware/software partitioning

Figure 3: System architecture.

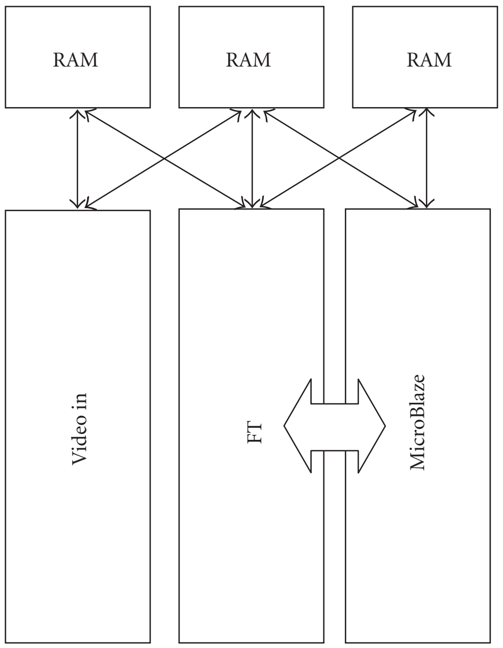


Figure 4: Mapping of the feature tracker on the column-based reconfigarable device.

rely on reconfiguration rather than on just exchanging module parameters to increase flexibility.

### 4.3. Instruction set metamorphosis

By analyzing the remaining part of the algorithm, namely, feature tracking, which is done in software, some more improvements are possible. The features of the MicroBlaze processor can be upgraded. Eight fast communication channels, called fast simplex links (FSL), are available to connect dedicated hardware accelerators, which are linked through FIFO buffers. The instruction set offers special put and gets commands to access these pipelines by software (shown in Figure 5).

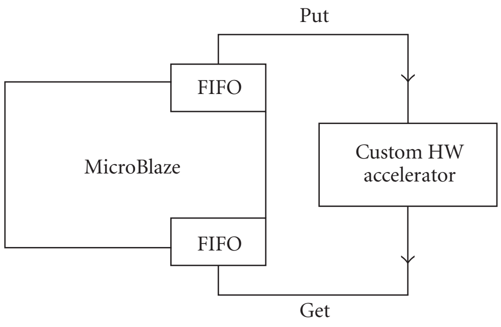


Figure 5: Instruction set extension through fast simplex link.

The tracking code reads each image point from the memory to calculate the parameters for the normalized correlation (8). A software implementation only allows a sequential computation of each individual pixel. We take advantage of the FSL and the dedicated hardware attached to it to increase the throughput as well as the speed of feature comparison. Since one pixel is represented by 8 bits and the FSL and memory bus width are both 32 bits, we are able to transfer and process four pixels at once. The hardware accelerator simultaneously calculates the sum $A$ and the sum of squares $B$ while the processor only pushs data into the FIFO. A similar procedure is used to compute scalar products $D$ while feature comparison phase.

## 5. RESULTS

Our design was implemented on a Spartan 3 (xc3s400) FPGA. This FPGA is to small to host all hardware accelerators together with the microBlaze processor. Thus our first implementation is a software only version which runs completely on the microBlaze.

Table 1: Performance of software only solution.

| | Spartan 3 board | ML310 board (Virtex2 Pro) |
|---|---|---|
| Feature detection | 5.01 s | 153 ms |
| Feature selection (30 features) | 0.89 s | 78 ms |
| Tracking | 1.6 s | 28 ms |

Table 2: Pipeline stages for hardware feature detection and selection and their latencies. Examples with different image formats and system clocks.

| Stage | Logic | Purpose | Latency |
|---|---|---|---|
| 1 | $3 \times 3$ convolve | $I_x$ and $I_y$ | $2l + 2 + 1$ |
| 2 | 3 multipliers | $I_*$ products | 1 |
| 3 | $1 \times 5$ convolve | Binomial horiz. | $4 + 1$ |
| 4 | $5 \times 1$ convolve | Binomial vert. | $4l + 1 + 1$ |
| 5 | Arithmetic | $d, t, s$ | 3 |
| 6 | $5 \times 5$ convolve | Feature selection | $4l + 4 + 1$ |
| Total | | | $10l + 19$ |
| QCIF ($176 \times 144$ pixels), 50 MHz | | | 35.58 $\mu$s |
| QCIF ($176 \times 144$ pixels), 100 MHz | | | 17.79 $\mu$s |
| VGA ($640 \times 480$ pixels), 100 MHz | | | 64.19 $\mu$s |

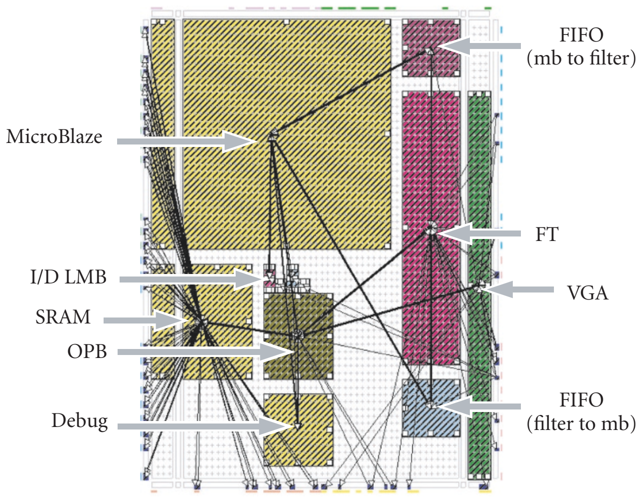


Figure 6: Floorplan of the feature tracker on a Xilinx Spartan 3 FPGA.

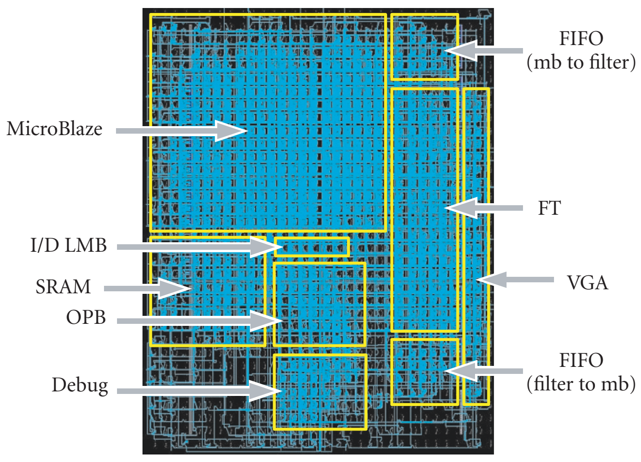


Figure 7: Placed and routed design of the feature tracker.

Figure 6 shows the placement of the modules in the floorplanner tool. The resulting design in its placed and routed form can be seen in Figure 7.

Considering the timings of a software only solution (which takes no advantage of hardware accelerators) feature detection takes 5.01 s executed on the introduced design. The latency for feature selection is proportional to the amount of features found, for example, the latency for 80 features is 2.39 s. As we will see in the following this is much slower than the hardware solution. The tracking part takes 1.6 s per frame.

Porting to a Xilinx ML310 board equipped with a Virtex2 Pro FPGA and using a newer version of the development tools further increased the performance and allowed timings in the range of milliseconds (see Table 1).

Table 2 summarizes the pipeline stages of the hardware feature detection and selection module (independent from the complete design) and their latencies. The underlying algorithm was already described in Section 2 so only some remarks follow: Stage 2 computes the values $I_xI_x$, $I_xI_y$, and $I_yI_y$ and stage 3 and 4 produce the values $G_{xx}$, $G_{xy}$, and $G_{yy}$. Stage 5 uses arithmetic units to calculate the corner response strength $s$. Finally stage 6 selects features using a modified convolve filter. The total latency results in $10l + 19$ clock cycles, where $l$ is the image width. The table also shows examples for different image formats and system clocks.

Because this design processes one pixel per clock, very high frame rates are achieved. The frame rate can be calculated as system clock divided by image size, for example, 1972 fps for a QCIF format or 162 fps for a VGA ($640 \times 480$ pixels) resolution. Of course the tracking part that runs in software on the MicroBlaze cannot achieve this high performance and thus is the bottleneck in this case. The tracking delay of 28 ms allows about 3-4 frames per second with a video stream in QCIF format.

## 6. CONCLUSIONS

In this paper, we have designed and implemented an efficient and flexible feature tracker on a reconfigurable device. The efficiency is obtained by using a viable hardware/software partitioning, by communication between modules, as well as by using an efficient memory access. Furthermore, the exploitation of the MicroBlaze features, like the fast simplex link, improves the performance further. Contrary to other works that modify the parameters of some filter to increase flexibility, we use reconfiguration to exchange hardware modules with different structures. The progress made in the last decade have affected the power consumption and the size of FPGAs, their costs have dropped rapidly while their capacity continues to increase. This trend is expected to continue, allowing the use of FPGAs in mobile autonomous environments. Our future work is to further improve the tracking part of our solution and the deployment in a system of cooperative intelligent robots using FPGAs as computing platform.

## REFERENCES

[1] P. Lysaght, “FPGAs in the decade after the von Neumann century,” in *Friday workshop at Design, Automation and Test European (DATE ’06)*, Munich, Germany, March 2006.

[2] B. D. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” in *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI ’81)*, pp. 674–679, Vancouver, BC, Canada, August 1981.

[3] C. Tomasi and T. Kanade, “Detection and tracking of point features,” Tech. Rep. CMUCS-91-131, Carnegie Mellon University, Pittsburgh, Pa, USA, 1991.

[4] J. Shi and C. Tomasi, “Good features to track,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR ’94)*, pp. 593–600, Seattle, Wash, USA, June 1994.

[5] J. Shi and C. Tomasi, “Good features to track,” Tech. Rep. TR-93-1399, Department of Computer Science, Cornell University, Ithaca, NY, USA, 1993.

[6] T. Zinßer, C. Gräßl, and H. Niemann, “Efficient feature tracking for long video sequences,” in *Proceedings of the 26th DAGM Symposium*, pp. 326–333, Tübingen, Germany, August-September 2004.

[7] D. Nister, O. Naroditsky, and J. Bergen, “Visual odometry,” Tech. Rep. CN5300, Sarnoff Corporation, Princeton, NJ, USA, 2004.

[8] J. Chen, D. Silver, and M. Parashar, “Real time feature extraction and tracking in a computational steering environment,” in *Proceedings of the High Performance Computing Symposium (HPC ’03)*, pp. 155–160, Society for Modeling and Simulation International, San Diego, Calif, USA, March-April 2003.

[9] M. B. Kulaczewski and H. J. Siegel, “Implementations of a feature-based visual tracking algorithm on two MIMD machines,” in *Proceedings of the International Conference on Parallel Processing*, pp. 422–430, Bloomington, Ill, USA, August 1997.

[10] M. B. Kulaczewski and H. J. Siegel, “SIMD and mixed-mode implementations of a visual tracking algorithm,” in *Proceedings of the International Parallel Processing Symposium (IPPS/SPDP ’98)*, pp. 716–720, Orlando, Fla, USA, March-April 1998.

[11] A. Bissacco, J. Meltzer, S. Ghiasi, S. Soatto, and M. Sarrafzadeh, “Fast visual feature selection and tracking in a hybrid reconfigurable architecture,” Tech. Rep., UCLA, Los Angeles, Calif, USA, 2004. http://www.cs.ucla.edu/~bissacco/hybridfeatrack.html.

[12] X. Feng and P. Perona, “Real time motion detection system and scene segmentation,” Tech. Rep. CIT-CDS-98-004, California Institute of Technology, Pasadena, Calif, USA, 1998.

[13] S. Ghiasi, A. Nahapetian, H. J. Moon, and M. Sarrafzadeh, “Reconfiguration in network of embedded systems: challenges and adaptive tracking case study,” *Journal of Embedded Computing*, vol. 1, no. 1, pp. 147–166, 2005.

[14] C. Bobda, M. Majer, A. Ahmadinia, et al., “The erlangen slot machine: a highly flexible fpga-based reconfigurable platform,” in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM ’05)*, pp. 319–320, Napa, Calif, USA, April 2005.

**Felix Mühlbauer** completed his degree in the diploma course of studies in computer science at the University of Erlangen-Nuremberg, in 2005. Since November 2005 he works as a Scientific Assistant in the Self-Organizing Embedded Systems group in the Department of Computer Sciences at the University of Kaiserslautern .

**Christophe Bobda** is the leader of the new created working group “Self-Organizing Embedded Systems” in the Department of Computer Sciences at the University of Kaiserslautern. He 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, respectively. In June 2003, he joined the Department of Computer Science at the University of Erlangen-Nuremberg in Germany as a Postdoctoral. He 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. He is a Member of the IEEE Computer Society, the ACM, and the GI. He is also in the program committee of several conferences (FPT, RAW, RSP, ERSA, DRS), the DATE executive committee as proceedings chair (2004, 2005, 2006). He served as a 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).