Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Fast and robust camera-based motion tracking using FPGAs in microrobotics

Dissertation zur Erlangung des Grades eines
Doktors der Ingeneurswissenschaften (Dr.-Ing.)

von

Dipl.-Inform. Claas Diederichs

Gutachter:

Prof. Dr. Sergej Fatikow
Prof. Dr. Michaël Gauthier

Tag der Disputation: 28. November 2014

Abstract

In microhandling, the optical sensor is often the only way to obtain position information about tools and specimen. Optical microscopes are used to observe e.g. a manufacturing process. Image processing is extensively used if handling processes are fully automated. Based on the image sensor, objects are detected and classified. Additionally, the image sensor is often used as a motion tracking system for closed-loop positioning of tools (e.g. grippers or tips). There are several reliable algorithms available for motion tracking. Still, the speed and accuracy of a closed-loop positioning system is constrained by the quality of the sensor system used. In this case, the sensor system consists of the physical camera and the motion tracking system. From a control engineering point of view, the sensor system quality is defined by characteristics such as resolution and noise, but also by timing characteristics such as update-rate, latency and jitter. A closed-loop controller can perform significantly better if the sensor signal has a high update-rate and a low and predictable latency. A predictable and well known latency additionally allows for controller optimizations. High speed motion tracking is especially required in microrobotics because small objects have very low inertia and have high-speed dynamics.

Motion tracking using off-the-shelf computers and image-processing software has high latencies and high jitter. The achievable update-rates are up to 100 Hz. Experiments show that the update-rate has a high jitter, correlating with the used algorithm, the image content and the system load. Fast and robust motion tracking needs higher update-rates and a predictable runtime, thus a very low jitter. Therefore, for fast and robust motion tracking, different techniques are needed.

Different approaches to improving the timing performance of motion tracking are exploited. First, a real-time operating system was used to control the scheduling of tasks. This ensured that the motion tracking would not be preempted by other processes running on the computer. Secondly, a hardware-software co-design strategy was used: An FPGA-implementation of a tracking algorithm was utilized on an embedded system. Using an FPGA, a major part of the image processing can be done in parallel performing hardware blocks. Additionally, a stream mode implementation can be used. Stream processing does not operate on an image that is available in memory, but on single pixels that are sequentially streamed through the hardware. Therefore, the image can be directly processed while its data is acquired from the camera chip and, instead of written back to memory, directly

passed to the next operation. Such a pipelined design reduces the latency of image processing significantly.

A novel hardware implementation of the Binary Large Object (BLOB) detection approach was implemented to address the above mentioned timing characteristics. This implementation is superior to state of the art hardware implementations in terms of speed, controllable objects and resource utilization. Additionally, the implementation calculates novel BLOB features that allow for better object classification.

While the real-time operating system improved all characteristics slightly, the FPGA implementation showed huge improvements especially on latency, jitter and predictability. The achievable update-rate was improved to 1.8 Ghz. The latency was improved to exactly one camera frame (exposure time plus transfer time), which is the best theoretical value. Also, the jitter was improved by more than one order of magnitude, thus allowing for a high predictable update-rate and latency.

The developed FPGA-solution was integrated into a microhandling station to evaluate the performance. In this microhandling station, two mobile microrobots were cooperatively performing pick-and-place operations with 50 µm objects. The robots had no internal position sensors and needed to be controlled by optical motion tracking. Prior to the integration, the robots had been tracked by PC-based motion tracking. With the integration it was possible to reduce the time of a single pick-and-place operation from 15 seconds to less than two seconds. To show the flexibility of the developed algorithm, the prototype was also applied to a traffic-sign recognition system.

Zusammenfassung

In Mikromontagesystemen werden Positionen von Werkstücken und Manipulatoren oft durch Mikroskope und Kameras erfasst und ausgewertet. Dabei werden verschiedene berechnungsintensive Bildverarbeitungsalgorithmen genutzt, um Werkstücke zu erkennen und zu klassifizieren oder um Werkzeuge wie Greifer, Spitzen oder Kanülen zu positionieren. Wird die Bildverarbeitung als Sensor in die Positionsregelung eines Manipulators eingebunden, wird dies Visual Servoing genannt. Zu diesem Zweck sind verschiedene Algorithmen in den letzten Jahrzehnten entwickelt worden. Die Geschwindigkeit und die Positioniergenauigkeit einer Regelung sind abhängig von dem eingesetzten Sensorsystem, das bei Visual Servoing aus Kamera und Bildverarbeitungssystem besteht. Die Qualität eines Sensors wird über Werte wie Auflösung, Signalrauschen und Linearität definiert. Es sind aber auch zeitliche Merkmale wie Wiederholrate, Latenz und Jitter von Bedeutung. Hat ein Signal eine hohe Wiederholrate, eine geringe Latenz und ein vorhersagbares Latenzverhalten, kann eine Regelung erheblich schneller und, durch die Vorhersagbarkeit der Sensorlatenz, genauer arbeiten. Insbesondere in der Mikrorobotik werden Hochgeschwindigkeitssensoren benötigt, da Mikroaktoren kleine Massen und daher eine hohe Dynamik aufweisen.

In der Arbeit wird gezeigt, dass die oft eingesetzten Standard-PCs mit Bildverarbeitungssoftware eine hohe Latenz aufweisen und die Wiederholrate sowie die Latenz starken Schwankungen unterliegt. Die erreichbaren Wiederholraten sind, je nach Algorithmus, im Bereich von 100Hz und zudem stark abhängig von der Gesamtlast des Systems.

Es wurden verschiedene Ansätze untersucht, die drei Merkmale Wiederholrate, Latenz und Jitter zu verbessern. Als bester Ansatz hat sich die Benutzung eines Field Programmable Gate Arrays (FPGA) herausgestellt. Auf einem FPGA kann ein großer Teil der Bildverarbeitung in parallel arbeitenden Hardwareblöcken durchgeführt werden. Zudem kann eine Datenstromimplementierung genutzt werden, die bereits arbeitet während der Sensor ausgelesen wird, was zu einer verbesserten Latenz führt. Durch die Nutzung bekannter sowie neu entwickelten BLOB-Eigenschaften kann ein erkanntes Objekt klassifiziert werden.

Eine neuartige Hardware-Implementierung des Binary Large OBject (BLOB) Extraction Algorithmus wurde entwickelt, die alle drei Merkmale im Vergleich zu einer PC-Lösung signifikant verbessern. Die Klassifizierung wurde zunächst mit statischen Grenzen für die verschiedenen Eigenschaften implementiert. Der entwickelte Prototyp wurde in eine Mikrohandhabungsstation integriert und konnte

dort die Prozesszeit einer einzelnen Pick and Place Operation von 15 Sekunden auf unter 2 Sekunden verbessern. Der Prototyp wurde weiterhin in einem System zur Straßenschilderkennung eingesetzt.

Acknowledgements

The presented work has been carried out at the Division Microrobotics and Control Engineering (AMiR) at the University of Oldenburg, Germany, headed by Prof. Dr.-Ing. Sergej Fatikow. I would like to express my gratitude to Prof. Dr. Fatikow for supervising my Ph.D. thesis, the division's excellent laboratory equipment as well as the confidence he had in my work. I highly appreciate the freedom I had during the time of my doctorate. Furthermore, I would like to thank Prof. Dr. Michaël Gauthier for refereeing this dissertation.

I would like to thank all of my colleagues for the excellent team work. I especially thank Daniel Jasper for all the fruitful discussions, the joint work and the support he gave me even after he left the team.

I would like to thank my parents Dörte and Wolfgang Diederichs for their constant support throughout my life. Finally and most importantly, I would like to thank my family, my wife Lotte and my two children Emma and Marieke for their moral support throughout the years. Without your support, I would not be where I am today.

Contents

Abstract **iii**

Zusammenfassung **v**

Acknowledgements **vii**

List of Figures **xi**

1 Introduction **1**
1.1 Goals 2
1.2 Outline / Author's contribution 3

2 State of the art **5**
2.1 Object tracking in microrobotics 5
2.2 FPGA-based image processing 6
2.3 The binary large object extraction algorithms 14
2.3.1 Random-pixel access algorithms and memory-based algorithms 15
2.3.2 Features 17
2.3.3 Single pass algorithms 19

3 A novel FPGA implementation of the BLOB extraction algorithm **23**
3.1 Requirements and goals 25
3.2 Features 27
3.2.1 Weighted center of gravity 27
3.2.2 Bounding polygon 28
3.2.3 Principal component analysis based features 29
3.3 Equivalence handling 32
3.3.1 Implementation 36
3.4 Architecture 49
3.4.1 Parameters and RAM sizes 58
3.5 Further throughput improvement 62
3.6 Implementation and test strategy 64
3.6.1 Waveform simulation 64
3.6.2 Hardware in the Loop 66

3.6.3 Resource usage and throughput . . . 71
3.7 Conclusions . . . 75

4 Validation **77**
4.1 Test setup . . . 77
4.2 Prototype . . . 78
4.3 Update-rate jitter . . . 82
4.4 Latency and latency jitter . . . 93
4.5 Conclusions . . . 107

5 Applications **111**
5.1 Automated handling of microspheres . . . 111
5.1.1 Comparison of software-based and FPGA-based tracking . . . 112
5.1.2 Tracking of multiple robots . . . 118
5.1.3 Distortion correction . . . 120
5.1.4 Sensor accuracy evaluation . . . 122
5.1.5 Closed loop performance . . . 124
5.1.6 Conclusions . . . 127
5.2 Traffic sign recognition . . . 127
5.3 Conclusions . . . 131

6 Conclusions and outlook **133**
6.1 Conclusions . . . 134
6.2 Outlook . . . 135

Bibliography **137**

List of Figures

2.1 Logical layout of a hardware thresholding operation 9
2.2 Simplified logic layout of a hardware local filter 11
2.3 Control signals of stream-based image-processing 13
2.4 Control signals of stream-based image-processing (2) 13
2.5 Neighborhood for connected component labeling 15
2.6 Two-pass connected component labeling algorithm 17
2.7 The single-pass architecture used by Bailey and Johnston (2007). . 20

3.1 Example of a bounding octagon 28
3.2 Image of a micro-gripper 32
3.3 Example of a subset in the disjoint set 33
3.4 Union of two subsets in a disjoint set 34
3.5 Master exchange of a subset in the disjoint set 35
3.6 Shrinking of a subsection in a disjoint set 36
3.7 Union of two subsets that both have a finished master. 37
3.8 State machine of the equivalence handling hardware system 38
3.9 Union of two subsets consisting of only one element 40
3.10 Union of two subsets consisting of two elements each 41
3.11 Union of two subsets consisting of two elements each where both masters are finished 42
3.12 Switching of masters when a child is finished 43
3.13 Switching of masters in a larger subset 45
3.14 Architecture of the parallel approach algorithm 50
3.15 First-pass neighborhood of the parallel approach 52
3.16 Label assignments during the first pass 53
3.17 New label assignments during the first pass 53
3.18 Different join situations 54
3.19 Advanced first pass neighborhood for nine pixels 63
3.20 Architecture of the waveform simulation 66
3.21 Architecture of the hardware in the loop system 67
3.22 Different test images 69
3.23 Randomly generated test images 69

4.1 Image observed by the test setup 78

4.2 Setup for the validation of the system timing (FPGA) 79
4.3 Setup for the validation of the system timing(PC) 79
4.4 Overview of the hardware platform used 81
4.5 Overview of the FPGA design used 83
4.6 Update rate distribution of one object 84
4.7 Update rate histogram of one object 84
4.8 Update rate distribution of one object with the full processor load . 85
4.9 Update rate histogram of one object with full processor load 86
4.10 Update rate distribution of eight objects with a full processor load . 87
4.11 Update rate histogram of eight objects with a full processor load . . 87
4.12 Update rate distribution of one object with a real-time extension . 88
4.13 Update rate histogram of one object with a real-time extension . . 89
4.14 Update rate distribution of eight objects with a real-time extension and a full processor load 90
4.15 Update rate histogram of eight objects with a real-time extension and a full processor load 90
4.16 Update rate distribution of eight objects tracked with FPGA 91
4.17 Update rate histogram of eight objects tracked with FPGA 92
4.18 Latency distribution of one object 94
4.19 Latency histogram for one object 95
4.20 Latency distribution of eight object 95
4.21 Latency histogram for eight objects 96
4.22 Latency distribution of one object with a full processor load 97
4.23 Latency histogram for one object with a full processor load 97
4.24 Latency distribution of eight objects with a full processor load . . . 98
4.25 Latency histogram for eight objects with a full processor load . . . 98
4.26 Latency distribution of one object with a real-time extension 99
4.27 Latency histogram for one object with a real-time extension 99
4.28 Latency distribution of eight objects with a real-time extension . . 100
4.29 Latency histogram for eight objects with a real-time extension . . . 100
4.30 Latency distribution of one object with a real-time extension and a full processor load 101
4.31 Latency histogram for one object with a real-time extension and a full processor load 102
4.32 Latency distribution of eight objects with a real-time extension and a full processor load 102
4.33 Latency histogram for eight objects with a real-time extension and a full processor load 103
4.34 Latency distribution of one object tracked with FPGA 104
4.35 Latency histogram for one object tracked with FPGA 104
4.36 Latency distribution of eight objects tracked with FPGA 105

4.37 Latency histogram for eight objects tracked with FPGA 105

5.1 Mobile robot actuation unit 112
5.2 Bottom view of the tracked mobile robot 113
5.3 Architecture of the tracking system before the FPGA deployment . 113
5.4 Update rate deviation of software-based robot tracking 114
5.5 Latency of software-based robot tracking 115
5.6 Architecture of the tracking system using an FPGA 115
5.7 Update intervals of hardware-based led tracking 117
5.8 Mapping of two region centers to robot position and angle 119
5.9 Robot tracking performed in alternating ROIs 119
5.10 Architecture and dimensions of the multiple robot tracking system 120
5.11 Distorted image taken by the camera beneath the glass surface . . . 121
5.12 Noise standard deviation at the lens center 122
5.13 Noise maximum deviation at the lens center 123
5.14 Sensor linearity of one square pixel 124
5.15 Trajectory control algorithm for mobile robots 124
5.16 Error over time while driving along a linear trajectory 126
5.17 Measurements of a linear trajectory 126
5.18 Different German traffic signs 128
5.19 Traffic sign recognition architecture 128
5.20 Traffic sign recognition hardware chain example 130

1 Introduction

Image processing and motion tracking is a key concept in micro- and nanorobotics. In optical motion tracking, the camera is used as a sensor to determine the motion of a tool or object in the view of the camera. The information of motion tracking is often used as input for position controllers. The position controllers need to rely on the correctness of the information from motion tracking. However, not only the correctness of the position is of importance, but also the age of the sensor signal. If the sensor signal is too old, the motion controller works with outdated information. If the age of the sensor signal is deterministic and well-known, the controller can, with some effort, compensate for it. If the age of the sensor data is unknown or has a large jitter, the motion controller cannot compensate for it. Especially jitter in the position update-rate and the update latency can lead to incorrect behavior in the motion controller. A sensor signal with large jitter and unknown latency will lead to a decreased positioning speed. On the one hand, large unknown latencies will cause the controller to overshoot the target position, resulting in an oscillating position behavior. It can even be the case that the target position cannot be reached with the desired accuracy. This has to be compensated for; either by using less accurate target positions or a lower positioning speed.

Most camera systems used for motion tracking are consumer- or industry cameras that are used with off the shelf computers and software-based image-processing. In motion tracking for microrobotics, this constellation can be found in almost every manipulation setup.

Such a setup has the problematic characteristics described above. Firstly, it is unknown how long the sensor latency is. The sensor latency consists of the camera's exposure time, the time needed to transfer the data to the PC, the time needed by image-processing and the communication to the motion controller.

While some of the latency parts are deterministic and can be analyzed, others are not. The exposure time of the camera is dependent on the illumination and the auto-exposure behavior. For most cameras, the exposure time can be set to a fixed value. The transfer time from the camera to the PC is dependent on the interface used and its real-time capabilities.

However, with the common operating systems the latency that is caused by image processing is highly dependent on the hardware used and is influenced by other processes running on the PC.

Four problems can be identified for PC-based image processing:

- The latency of a PC-based motion tracking system is long and not deterministic
- The update-rate of a PC-based motion tracking system is limited and not deterministic
- The timing behavior of a PC-based motion tracking system is influenced by other processes running on the same PC.
- The nondeterministic behavior of PC-based motion tracking systems results in longer and less accurate positioning steps.

While some of these problems can be resolved using real-time operating systems, others remain. Additionally, the image processing throughput (i.e. update-rate) is often increased by using regions of interest (ROIs) in an image. The camera is configured in a way that only a small sub-image of the sensor is transferred to the PC and used for image processing. While the usage of ROIs increase the update-rate, the nondeterministic calculation behavior remains. Furthermore, using a ROI limits the area where other image-processing steps are performed and is often not feasible.

This work aims to improve the timing in positioning steps using image-based motion tracking.

As software-based image-processing running on PCs suffers from several problems, one goal is to develop an object detection system based on deterministic hardware. FPGAs are well suited for highly parallel, deterministic computations. Image processing on FPGAs is widely used for preprocessing and filtering steps. These operations are well understood and several deterministic, highly parallel FPGA implementations exist that can facilitate high speeds.

However, deterministic, robust high speed motion tracking systems are not yet state of the art.

1.1 Goals

The overall goal of this work is to improve the speed and robustness of positioning steps during camara-based positioning. To overcome the timing problems that exist with PC-based image processing, an FPGA system for robust high speed

motion tracking can be used. The goal of this work is to develop such a system with the following characteristics:

- High throughput: The tracking system must allow for high update-rates. As goal, the fastest speed of the CameraLink protocol (680 MB/s) should be possible.
- Low latency: The computational time for motion tracking should be as low as possible.
- Deterministic latency: The latency must be predictable and must not be influenced by the image content; an image with no objects shall have the same latency as an image with several objects. Furthermore, noise should have no impact on the latency.
- Robustness: The tracking system must continue to work with a deterministic timing behavior for unexpected images such as high and unexpected noise or images with unexpected highly numbers or sizes of objects.

A secondary goal is to test the tracking system and compare the results to PC-based image processing. Furthermore, a real-time operating system shall be evaluated with the same test setup to analyze whether the FPGA system has benefits over software-based tracking if a real-time operation system is used.

The last goal is to deploy the system in a microrobotic handling cell and evaluate a possible performance gain.

1.2 Outline / Author's contribution

The thesis contributes to the advancement of the closed-loop motion control for automated microhandling and microassembly and is structured in five chapters following this introduction.

chapter 2 describes the current state of the art for automated microhandling and FPGA-based image processing. For FPGA-based image processing, different implementation paradigms must be taken into consideration when deterministic behavior is the aim. The chapter gives an overview of the differences between software-based computations for image processing and hardware based approaches. An overview of FPGA-based object detection approaches is given. Special attention is given to the binary large object detection algorithm that is the base of the developed tracking system.

A novel implementation of the binary large object detection algorithm is introduced in chapter 3. The implementation choses a novel approach. This allows for a high throughput on the one hand, and robustness and deterministic timing

behavior on the other hand. The implementation can reach throughputs of up to 1.8 GB/s (depending on the FPGA used) while being robust and deterministic in the timing behavior. The system is tested with different test images, including images consisting only of noise and images consisting of several thousand objects.

In chapter 4, the tracking system is tested and compared against software-based image processing. A test architecture was developed that measures the response time of a tracking system by using a dedicated FPGA for the measurement. A real-time operating system was used to improve the performance of the PC-based tracking. Results show that a real-time system improves the timing behavior of PC-based image processing significantly, but the FPGA system is magnitudes better concerning the update-rate deviation, can handle higher data rates and has a lower latency than PC-based systems.

Two different applications that use the FPGA-tracking implementation are described in chapter 5. The focus is on the improvement of an existing microhandling cell. The microhandling cell was upgraded with the FPGA-based tracking algorithm. The closed-loop positioning speed could be improved by one magnitude. A second application is presented to underline the possibilities the FPGA-based object detection provides in other fields. The second application describes a low-cost traffic sign recognition system that aims at low power consumption.

The final chapter 6 contains a conclusion and an evaluation of this work's impact on the speed, throughput and industrial applicability of automated microhandling and microassembly. An outlook describes future steps extending this work.

2 State of the art

In micro- and nanohandling, the optical sensor is often the only device that can obtain position information about tools and specimen. In microhandling, optical microscopes are used to observe e.g. a manufacturing process. In contrary, a scanning electron microscope (SEM) is used for nanohandling operations, as the resolution of optical systems is not sufficient for these tasks. For both tasks, image processing is extensively used if handling processes are fully- or semi-automated. Based on the image sensor, objects are detected and classified. Additionally, the image sensor is often used as a motion tracking system for the closed-loop positioning of tools (e.g. grippers or tips); this is called visual servoing. Even if other position sensors are available for tools (e.g. internal position sensors of positioning systems), visual servoing of the end-effector is often required, e.g. to perform positioning relative to specimen. This chapter gives an overview of the state of the art. First, an overview of object-detection and motion-tracking in microrobotics is given in section 2.1. In section 2.2, an overview of FPGA-based image processing is given. FPGA-based image-processing differs from conventional image-processing; the differences are described in this section. One special object-extraction approach, the Binary Large OBject (BLOB) extraction, and its FPGA-implementations are described in section 2.3.

2.1 Object tracking in microrobotics

Visual object detection and tracking are key concepts in microrobotics. Several works exist that report individual solutions to specific microrobotic problems (Papanikolopoulos et al., 1992; Pappas and Codourey, 1996; Fatikow et al., 1999, 2000; Sun and Nelson, 2002; Krupa et al., 2003; Ferreira et al., 2004; Dechev et al., 2004; Amavasai et al., 2005; Sievers and Fatikow, 2006; Hasegawa et al., 2008; Wortmann et al., 2009; Wang et al., 2009; Huang et al., 2010; Tamadazte et al., 2010, 2011; Ni et al., 2012; Tamadazte et al., 2012a; Diederichs et al., 2013).

However, all of these approaches use software-based image processing. While the throughput, and therefore the achievable update-rate, increases with each new generation of general purpose CPUs, problems like jitter, displacement and non-deterministic behavior were rarely addressed.

Ogawa et al. (2005) was one of the first to use a real-time operating system for motion tracking in microrobotics. More recent is the work of Kharboutly and Gauthier (2013). In their research, a dielectrophoresis-based system is controlled using visual servoing and a real-time operating system. The systems operated with an update-rate of 1 KHz. However, to achieve this update-rate only a small region of interest (256×256) pixels is used for motion tracking. Additionally, the algorithm used is a very simple one and not robust against noise.

Hardware-based image processing has been used used inside a small sensor smart camera system for eye in hand micro-manipulation (Tamadazte et al., 2012b). Here, simple pre-processing steps can be performed inside the smart camera.

Hardware implementations for microrobotic motion tracking have only been reported by the author of this thesis (Diederichs, 2010, 2011; Diederichs and Fatikow, 2013).

There is no known previous work that addresses and analyzes the timing issues of motion tracking in microrobotics.

2.2 FPGA-based image processing

Image-processing in general purpose CPUs is based on computational operations that read the input information from the memory and write the output information to the memory. The throughput of CPU-based image processing is therefore highly dependent on the memory read- and write speed. Furthermore, every image processing step needs the input to be fully available in the memory. If several image processing steps need to be performed, each processing step has to wait until the previous image processing step has finished and written the result to the memory. Each of the image processing steps therefore adds to the overall computation time for e.g. a motion tracking operation. Furthermore, CPU-based image processing has to wait until the image is fully transferred from the camera and available in the memory.

For FPGA-based image processing, several different approaches are possible. Firstly, it is possible to transfer the memory-based approach from software-based image processing to FGPAs. This may have small benefits, as it is possible to have a very high parallelism with FPGAs. However, it is also required to have a large amount of memory available. Furthermore, the read and write lanes of such a memory can become a limiting factor due to high parallelism. This kind of image processing is often referred to as *offline* image processing (Gribbon et al., 2006).

A second approach for FPGA based image processing is to use *stream-based processing*. Instead of reading and writing image data from and to the memory, the

image data is streamed line-wise, pixel after pixel, through hardware components. In stream processing, the hardware system is clocked with a fixed pixel clock. With every clock cycle, a new pixel is streamed into a hardware component. The hardware component usually has an output of one pixel value per clock cycle. For example, a thresholding system would compare the incoming pixel to a threshold and give either a black or a white pixel as output (see equation 2.1). This can easily be done in one clock cycle.

This kind of image processing has several implications that need to be taken into account when designing stream-processing algorithms. The most severe implication is that it is not possible to pause the input. In software-based image processing, it is possible to read a pixel from the memory and perform several operations before writing the result back to the memory. Using stream-based processing, with every cycle a new pixel is pushed into the hardware component.

The second implication is that random pixel access is not possible. When working on an image in the memory, all pixels of the image can be accessed. In the streaming mode, pixels that follow the current pixel are not available yet, and pixels that preceded the current one have to be stored in order to secure further access. For stream-based image processing, algorithms that require full random pixel access (e.g. working in two different areas of the image) are not feasible. However, it is possible to buffer small portions of the image to have limited access to surrounding pixels. An example is the case of line buffers, that store one or more rows of the image. With this technique local filters, also referred to as neighborhood filters, can be implemented. Examples of local filters are morphology operations or noise reduction filters such as the Gaussian filter, median filter or mean filter.

The most simple case that can be implemented with a stream processing algorithm is a point operation. The result of a point operation is only generated from the current pixel value and static inputs. Examples of point operations are thresholding, brightness changes or contrast changes. The equation for a thresholding operation that produces only black or white values is shown in equation 2.1.

$$v_o = \begin{cases} 255 & \text{if } v_i >= t \\ 0 & \text{else} \end{cases} \tag{2.1}$$

with v_i being the incoming pixel value, t the threshold value and v_o the outgoing pixel value.

In a software implementation, the operation would go through all image positions and write the values to a new matrix (see listing 2.1).

Listing 2.1: Pseudocode of a thresholding operation.

```
foreach y in source:
    foreach x in source:
        if source[x][y] >= threshold:
            target[x][y] = 255;
        else:
            target[x][y] = 0;
        end if;
    end foreach;
end foreach;
```

In a hardware module, the operation can be written in VHDL or verilog and will be synthesized in hardware units. The VHDL code for a thresholding module is shown in listing 2.2.

The corresponding logical layout is shown in figure 2.1. While the basic concepts are the same, the implementation of the filters in an FPGA hardware system differs from the software approach. This becomes even more clear when writing local filters.

As an example, the binary morphological operation *dilatation* will be used (Serra, 1982). The dilatation filter, also referred to as the Minkowsky add operation, fills the whole local filter area (e.g. a 3×3 window) to one if the value of the center is one. In a software implementation, this could be written as shown in listing 2.3 (for simplicity, border handling is not taken into account).

This implementation cannot be transferred to hardware because a hardware module cannot update several pixels in one cycle. However, the same results as the

Listing 2.2: VHDL code of a thresholding operation.

```
process(clk)
begin
    if rising_edge(clk) then
        if data_in >= threshold then
            data_out <= (others => '1');
        else:
            data_out <= (others => '0');
        end if;
    end if;
end process;
```

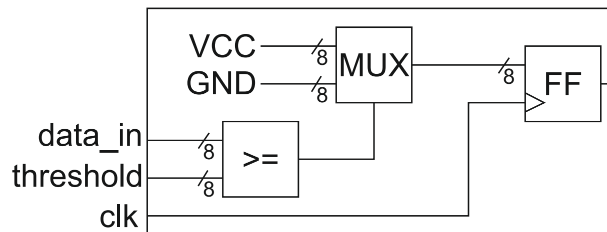


Figure 2.1: Logical layout of a hardware thresholding operation. The incoming data is compared with the threshold. If is is greater or equal, the comparator unit outputs a '1'. Otherwise, it outputs a '0'. Based on this, a mulitplexer (MUX) unit decides whether eight VCC values (i.e. binary '11111111' representing 255) or eight GND values (i.e. binary '00000000', representing 0) are outputted. The flip-flop (FF) at the end of the logic latches the output of the multiplexer at each rising clock edge.

Listing 2.3: Pseudocode of a dilatation operation.

```
target = allzero;
foreach y in source:
   foreach x in source:
      if source[x][y] == 1:
         target[x-1][y-1] = 1;
         target[x-1][y1] = 1;
         target[x+1][y+1] = 1;
         target[x][y-1] = 1;
         target[x][y] = 1;
         target[x][y+1] = 1;
         target[x+1][y-1] = 1;
         target[x+1][y] = 1;
         target[x+1][y+1] = 1;
      end if;
   end foreach;
end foreach;
```

Listing 2.4: Pseudocode of a maximum operation.

```
foreach y in source:
    foreach x in source:
        row1 = max(source[x-1][y-1],
                       source[x][y-1], source[x+1][y-1]);
        row2 = max(source[x-1][y],
                       source[x][y], source[x+1][y]);
        row3 = max(source[x-1][y+1],
                       source[x][y+1], source[x+1][y+1]);
        result[x][y] = max(row1, row2, row3);
    end foreach;
end foreach;
```

dilatation can be achieved when applying a maximum filter. The maximum sets the center pixel of a window to one if one of the surrounding pixels is one. This implementation only updates one pixel, but needs to read several pixels. A software implementation for the maximum filter could be written as shown in listing 2.4.

A hardware implementation of this algorithm is possible. However, again it is not directly transferable from the software approach, as the image is not stored in the memory and pixels other than the current pixel cannot be addressed directly. Therefore, a hardware implementation has to facilitate line buffers that buffer one or more rows of the image. The line buffer then outputs one pixel of each buffer in addition to the current pixel. Using this technique, a column of several pixels can be read in one clock cycle. If a local filter with a 3×3 window is used, two line buffers are needed. Additionally, the column that is sent by the line buffers has to be processed through a shift register to be able to calculate the output from a 3×3 window. figure 2.2 shows the architecture of such a local filter. Such line buffers require only limited memory. For example, a line buffer for a 3×3 window would require two lines to be buffered. For a maximum image width of 2048 and a pixel data depth of eight bit, a memory buffer would be needed that can hold 2048 elements that are 16 bit wide. FPGAs have dedicated memory blocks for this kind of buffers, referred to as BlockRam or tightly coupled memory (TCM). Furthermore, very small buffers can also be synthesized using only FPGA-slices (e.g. flip-flops). If FPGA slices are used to implement memory, this is referred to as distributed RAM.

If several image processing steps need to be performed, they need to operate one after another. In software, one step has to wait until the previous step is finished and has written its result to the memory. In FPGA-based stream-processing, operations can be pipelined. As every operation works on a pixel stream, the

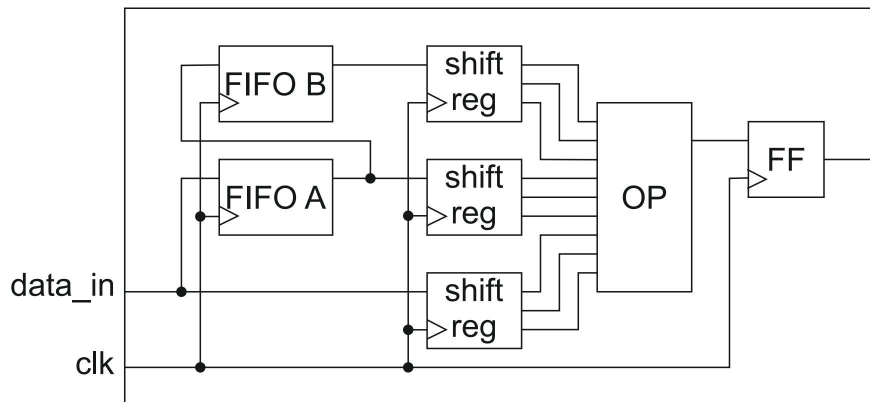


Figure 2.2: Simplified logic layout of a hardware local filter. The incoming pixel data is shifted info a FIFO (FIFO = first in first out. a small memory buffer. Elements are sent out in the same order as they ware pushed in.). The data is shifted into FIFO A as well as transfered to the shift register. The output of FIFO A is transferred into the shift register and into FIFO B. This simplified overview lacks the logic to control the FIFOs based on the columns. In theory, FIFO A contains the last line and FIFO B the line before that. When the third line is pushed into the logic, the first line is available in FIFO B and the second line in FIFO A. Thus, a three pixel column is streamed into the shift registers. The shift register shifts each incoming pixel by one, keeping the last two pixels. Together, the shift registers utilize the 3×3 window. The value of each shift register cell is pushed into the operation logic (OP) which can be any local filter implementation.

output stream of one operation can be used as the input stream for the next operation. Therefore, the delay of hardware-based image processing is very low. The delay introduced by each operation depends on the number of buffered elements during the operation. A point operation usually introduces a delay of one clock cycle and is in the range of 5 to 50 ns, depending on the pixel clock. For a window operation, the delay is the number of buffered lines multiplied with the image width and the clock time. For a 3×3 window operation on an image with a width of 2048 with 100 Mhz, the introduced delay would be 41 µs.

The delay of a pipeline step in stream-processing is fixed and predictable. In offline processing and software-based image processing, the delay depends on the computation time of the image-processing algorithm. Often, the computation time is depending on the image content. For example, the first dilatation algorithm needs more steps for an image with many white pixels than for an image with only black pixels because operations are only performed based on comparisons with the image content. Stream processing performs the operation regardless of the image-content; a pixel has to be produced in every clock cycle. In most cases, the delay of stream-processing is significantly lower than the delay of offline processing. However, some cases exist where the delay of offline processing might be lower. Nevertheless, the predictability of stream-processing is always better (Bailey, 2011; Diederichs and Fatikow, 2013).

The above descriptions of stream-based image processing gave a general overview about how to of point operation and local filter implementations. To achieve correct timing for the line buffers in the local filters, control signals are used in stream processing.

Stream based protocol

In stream-based image processing, with every clock cycle a pixel value is pushed into the logic. To be able to determine the position of the pixel value in the image, several control signals are used. These control signals follow the low-level interface of CMOS and CCD camera chips. The same control signals are also used in the CameraLink interface. In the simplest version of the protocol, one eight-bit pixel value is used. The partitioning into frames is controlled by a *frame valid* signal. The signal is high during the whole image frame. If the frame valid signal switches to low, the frame is finished. Pixels that arrive when the frame valid signal is low are not valid and must be ignored. Another signal partitions the frame into different rows. The *line valid* signal is high during the transition of one row. It changes to low on the end of each row. If the line valid signal is low, the pixel values are not valid and must be ignored. Figure 2.3 shows the interface control lines.

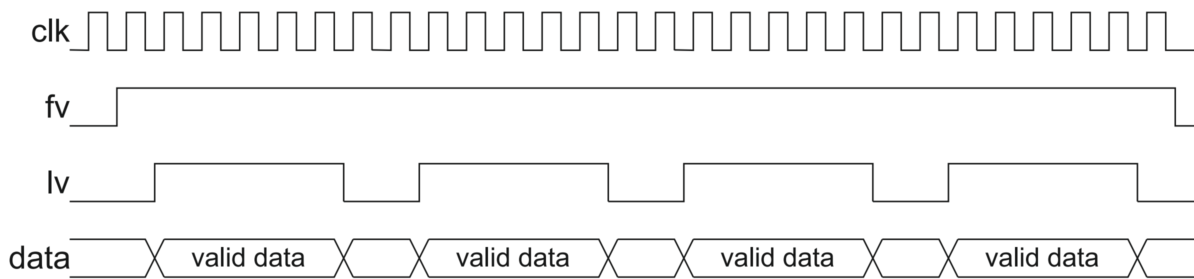


Figure 2.3: Control signals of stream-based image-processing. The shown protocol transmits a 5×4 image. After the frame valid (fv) signal changes to high, the line valid (lv) signal changes to high and stays high for five clock cycles. During this time, the five pixels of the first row are transferred from left to right, one in each clock cycle. This is repeated for each of the following rows. After the frame is transferred, the frame valid signal changes to low.

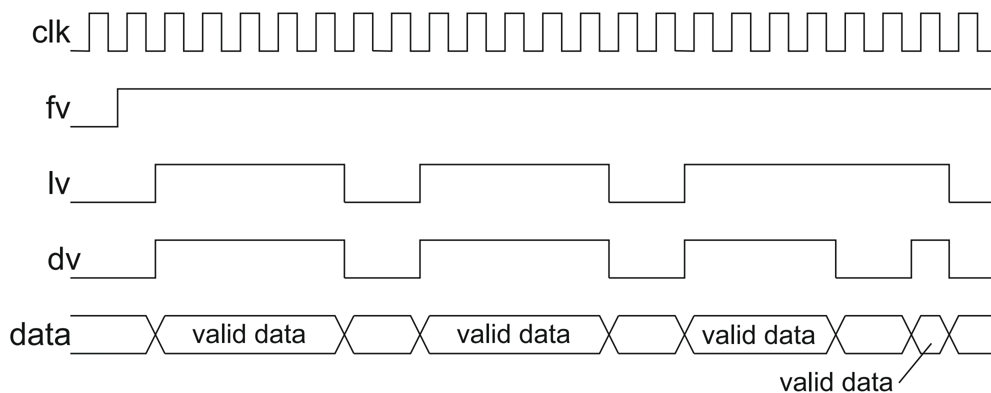


Figure 2.4: Control signals of stream-based image-processing. The general protocol remains the same. The transfer of image data can be interrupted during the transfer of a row. If this happens, the line valid (lv) signal remains high while the data valid (dv) signal changes to low. The pixel value is only valid if the line valid signal and the data valid signal are both high.

The extended interface introduces an additional *data valid* line. The data valid line can be used to pause the image transmission during a row. If this happens, the line valid signal remains high while the data valid signal changes to low. The pixel value is only valid if the line valid signal and the data valid signal are both high. If data valid changes to low, the number of cycles, where line valid stays high, are extended. An example of the protocol is shown in figure 2.4. This technique is useful when transferring image data with a faster clock than the image acquisition clock.

The camera link protocol extends this protocol further. In this protocol, several pixels are transferred in one clock cycle. The protocol specifies that one to eight consecutive pixels of one row can be transferred during one clock cycle. The camera link specification allows pixel clock frequencies of up to 85 MHz, allowing for transfer speeds of up to 680 MP/s.

While point operations and local filters are well known and have been used as hardware filters in cameras and other devices for a long time, in recent years several more complex image processing operations where transferred to FPGAs, mainly to exploit the high parallelism that is possible on FPGAs.

One of the major fields of complex image processing algorithms that benefit from faster processing times, lower latency and predictable behavior are object detection algorithms.

2.3 The binary large object extraction algorithms

One object-extraction approach that is well suited for the stream-based FGPA-implementation described in section 2.2 is the binary large object (BLOB) extraction algorithm. “Technically, image objects are formed out of components that in turn are made of connected pixels. It is thus most equitable to first detect components from images. When objects have been successfully extracted from their backgrounds, they also need to be specifically identified” (Chang et al., 2004). A variety of descriptors can be used to identify objects. One possibility is to use the contour either identified by chain codes (Freeman, 1961), Fourier descriptors (Persoon and Fu, 1977), or stochastic models (He and Kundu, 1991).

To detect a component, a method called connected component labeling can be used. Using this approach, each set of connected pixels is assigned a unique label. Every pixel in the image has a label that describes its affiliation with a component. Eventually, features can be calculated based on the labels rather than based on contours.

Different approaches exist for connected component labeling. They can be classified according to the classification introduced in section 2.2.

2.3.1 Random-pixel access algorithms and memory-based algorithms

A variety of different algorithms based on random pixel access exist; several were transferred to an FPGA. However, these are not suited for a deterministic low latency, high speed implementation (see section 2.2).

Chang et al. (2004) used an approach based on contour boundary scans. After detecting the boundaries, all pixels inside the boundary are assigned the same label. This approach makes use of existing fast algorithms for contour detection. Hedberg et al. (2007) introduced an FPGA implementation of the algorithm.

Another approach was introduced by AbuBaker et al. (2007), using flood-filling instead of boundary scans. Here, a flood-filling is performed as soon as a raster-scan finds an unlabeled foreground pixel.

Other approaches for connected-component labeling do not need random pixel access. These approaches use a local filter to assign the label of a pixel based on the pixels in the direct neighborhood. Figure 2.5 shows the basic principle. The pixel left and the pixels above the current one are used for label assignment. If one of these pixels is already labeled (i.e. has a label ID higher than 0), the current pixel belongs to the same component and therefore is assigned the same label ID. If none of these pixels have a valid label ID, the current pixel is assigned a new label ID.

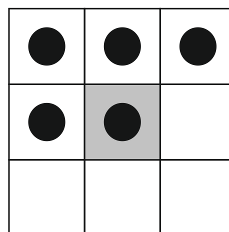


Figure 2.5: The neighborhood for connected component labeling. For the current pixel in the center, the labels of the pixels above and left are taken into consideration for the label assignment.

While the concept is fast and straightforward, several problems arise. The algorithm only knows the direct neighborhood of the current pixel. However, it is possible

that a pixel belongs to an already established component, even if the surrounding pixels belong to the background. The simplest example is a ∪-shaped object. The algorithm will encounter two vertical lines during the raster scan, and assign two different labels. Later during the raster scan, the algorithm will encounter the connection between the two regions. However, more complex objects will result in even more complex connections between assigned labels. Most implementations solve this problem by scanning the label matrix once or multiple times to correct the labels.

The different connected component labeling algorithms can be classified by the number of correction scans. An implementation that does not need additional intelligence is the multi-pass algorithm. After the first pass through the image, the label matrix is scanned backwards, correcting distinct labels to a single one. After that, the image is scanned forward again and so forth. The algorithm is finished when no correction was performed during a scan. This algorithm was implemented for an FPGA by Crookes and Benkrid (1999) and Benkrid et al. (2003). This approach needs no additional memory capacity besides the memory that stores the result label matrix. The circuit is simple and small, but the computational time depends on the complexity of the image. Additionally, the result image needs to be stored in the memory.

The most common approach for software implementations is the two-pass algorithm, proposed by Rosenfeld and Pfaltz (1966). This approach stores equivalent labels in an *equivalence table*. Instead of doing multiple iterations, equivalent labels are stored. After the first pass, all equivalences are resolved. A lookup table is created, which maps the initial labels to the final label. In the second pass, the lookup table is used to correct all connection labels into a single label ID. Figure 2.6 shows the process of the two-pass algorithm.

An FPGA implementation of the two-pass algorithm was introduced by Rachakonda et al. (1995). However, the two-pass algorithm requires storage of the result matrix containing the labels for each pixel. Additionally, the effort needed to transform the equivalence table to a lookup table can be complex and time consuming. The main difference between two-pass implementations is the handling of the equivalence table (Wu et al., 2005; He et al., 2007). From the two-pass algorithm, different single pass algorithms have evolved that perform an on-line handling of equivalences and performing feature detection during a single pass. These will be described in section 2.3.3.

While the approaches explained above describe the segmentation, the goal of binary large object extraction is not the exact segmentation, but the detection of robust and repeatable features (Forssén and Granlund, 2003). In all of the approaches described above, the feature detection is performed as an extra step after the

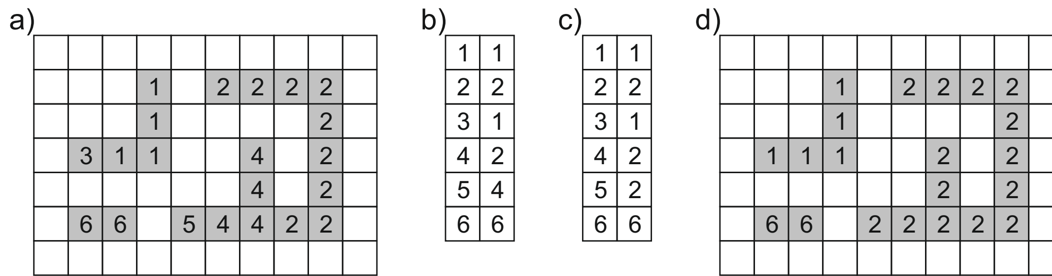


Figure 2.6: The two-pass connected component labeling algorithm. a) The labels for each foreground pixel after the first pass. During the first pass, the equivalences are stored in the equivalence table for each label. b) shows the equivalence table. From this, a lookup table (c) is generated, that contains the minimal equivalent label for each pixel. d) during the second pass, each label is replaced with its minimal equivalent label.

segmentation. The feature detection is often performed based on contours or based on the final label matrix.

2.3.2 Features

Features characterize the region they were extracted from. Each feature contains significant information about the region. The features can be used to distinguish regions from each other (Bailey, 2011). They can also be used for classification (Schomaker and Bulacu, 2004). Examples of state of the art features for BLOB extraction are:

- Number of pixels: The number of pixels that belong to the found region.
- Bounding box: A rectangle that encloses the region.
- Center of gravity: the gravital center of the region, based on all x and y coordinates on the pixels inside the region.
- Contour: The contour that encloses the region.
- Convex hull: a minimal convex contour around the object.
- Perimeter: The length of the object's contour.
- Color information: The average color or grayscale.

Based on these features, found regions can be classified and grouped, e.g. for motion tracking. The features of the found regions can be compared to pre-defined data to detect whether a region belongs to the desired group and should be analyzed further.

The feature calculation can be performed based on a segmentation matrix (e.g. after connected-component labeling) or based on contours (e.g. after contour detection).

If the features need to be calculated during the segmentation instead of after the segmentation, several conditions apply. The features have to be calculated incrementally during the segmentation process. Furthermore, especially if connected component labeling is used, situations occur where previous independent segments become one segment. In this case, it must be possible to combine the calculated feature data for both segments.

Especially for hardware (i.e. FPGA) implementations of feature detection during segmentation, the incremental calculation and the combination of feature data must be achievable in one clock cycle. Feasible operations for both are sums of values or comparisons. In most cases, the operation is the same during incremental calculation and merging. The least complex feature is the number of pixels in a segmented region:

$$np = \sum_{i=1}^{n}(i) \tag{2.2}$$

during segmentation, the sum can be calculated step by step. During merging, two sums can simply be added. For the bounding box, the minimum and the maximum x and y values of the region's pixels are calculated:

$$\begin{aligned} x_{min} &= \min(x(v_1), x(v_2), \ldots, x(p_n)) \\ x_{max} &= \max(x(p_1), x(v_2), \ldots, x(p_n)) \\ y_{min} &= \min(y(p_1), y(p_2), \ldots, y(p_n)) \\ y_{max} &= \max(y(p_1), y(p_2), \ldots, y(p_n)) \end{aligned} \tag{2.3}$$

During segmentation, the maximum and minimum values can also be calculated step by step, by comparing the stored value with the new value and storing the minimum or maximum, respectively:

$$x'_{max} = \begin{cases} x(p_i), & \text{if } x(p_i) > x_{max}, \\ x_{max}, & \text{otherwise} \end{cases} \tag{2.4}$$

The merging of two regions can be performed in a similar way.

For more complex features, computation is not directly possible. For example, the equation for the center of gravity is:

$$\begin{aligned} x_{cog} &= \frac{\sum_{i=0}^{n} x(p_i)}{n} \\ y_{cog} &= \frac{\sum_{i=0}^{n} y(p_i)}{n} \end{aligned} \tag{2.5}$$

To calculate these features incrementally, the numerator and the denominator are stored and updated individually. Then, once the segmentation of a region is finished, the result is calculated once. This only has to be done once for valid objects in the image.

A second limitation for hardware implementations is that knowledge about the memory sizes is needed. While the required memory for features like number of pixels, bounding box or center of gravity is known, several features cannot be calculated without variable memory usage. The contour and the convex hull of a segment are the most prominent examples. The number of points in a contour highly depends on the object's shape and can reach from three to several thousand. However, the contour is rarely used directly for classification. Instead, the contour is used as an intermediate result to represent a segment.

In summary, several features can be derived directly during segmentation. For hardware implementations, some features are not feasible, either because they cannot be calculated incrementally, or they need variable memory.

2.3.3 Single pass algorithms

The basic principle behind all single-pass approaches is to extract the data required to calculate the features during the first pass (Bailey, 1991, 2011) instead of storing the segmentation results. Different approaches exist, handling the first pass, the equivalence transformation and the feature data calculation without storing the image or intermediate results matrices.

Bailey and Johnston (2007) uses the architecture shown in figure 2.7. The equivalence transformation is done on-line. The merger-control selects the label for the current pixel based on the surrounding labels similar to the first pass in the above described algorithms, but the labels from the previous line are already resolved, as far as possible, by the equivalence transformation.

The merge information is pushed onto a stack during the labeling and resolved after each line. The approach needs a line blanking time of at least 25 % of the line width to be able to resolve the stacked merge information at the end of each row. Furthermore, the approach needs 262144 label IDs for the worst case in an 1024×1024 image. The authors propose an area-optimized algorithm that needs

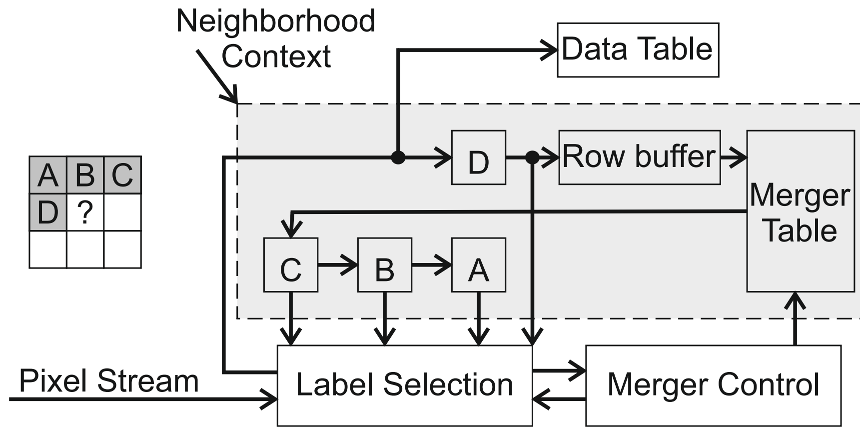


Figure 2.7: The single-pass architecture used by Bailey and Johnston (2007).

less labels and therefore less RAM. However, the optimized algorithm is not robust and requires the image to be noiseless and have only few objects.

The algorithm was demonstrated by Johnston and Bailey (2008) and further optimized by Ma et al. (2008). Ma et al. (2008) introduced label reuse to further lower the resource usage of the algorithm. Additionally, the merge algorithm was extended. However, the paper states that complicated structures cannot be handled by the algorithm and must be avoided for real-time applications. Additionally, the preconditions for the line blanking time are the same as in Bailey and Johnston (2007).

The approach of Bailey and Johnston (2007) was used as a basis for the work by Klaiber et al. (2012). The paper focuses on the throughput of single-pass connected component labeling. The main idea is to divide the image into several vertical slices and compute the connected component labeling in parallel for each slice. In order to do this, the label merging needs to be communicated between the different slices. Klaiber et al. (2013) optimizes the approach in terms of resource utilization by introducing label reuse. The hardware algorithm reaches throughputs of 3.2 GB/s for images with a width of 8192 pixels. However, the approach has a very high resource utilization. Also, only one very simple feature (the bounding box) was computed by the algorithm. Furthermore, no statement for the robustness of the algorithm against noise, a large number of objects and complex structures is given.

Another approach was developed by Trein et al. (2007) (Trein et al., 2007, 2008). This approach aims for a high throughput. The main idea is to use run-length coding for adjacent foreground pixels in one line. Blocks of 32 pixels in one line are transformed into run length encoded packets (called runs). The equivalence handling differs from the approach used by Bailey and Johnston (2007): The runs are pushed into a FIFO. Then, a merging algorithm pulls the runs from the FIFO and starts the labeling and merging. With this approach, throughputs of

680 MB/s can be achieved. The algorithm already uses label re-usage. However, the implementation has several drawbacks. During tests, the authors calculated that the processing of one run takes six cycles in average. Therefore, the compression of the run-length coding must be less than 1:6. Therefore, the algorithm is not robust against noise and a large number of small objects. Furthermore, it is not possible to calculate some features, because information is lost during run length encoding, e.g. the grayscale value of each individual pixel. Additionally, other features will be complex to compute, because the information about the x position of a pixel must be extracted from the run-length coding.

Most of the reported implementations have limitations, either in the throughput, the number of extractable BLOBs or the type of features they can detect. The throughput is limited by the number of concurrent processed pixels and the maximum clock frequency of the hardware implementation. Concerning the number of extractable BLOBs, image noise as well as size and structure of objects, most algorithms have preconditions to the image content.

Therefore, a novel algorithm is proposed in this thesis that allows for robust, high speed BLOB detection and is able to calculate multiple features with low resource utilization.

3 A novel FPGA implementation of the BLOB extraction algorithm

As described in section 2.3, several FPGA implementations of the Binary Large OBject (BLOB) extraction algorithms have been reported. Most of these have limitations, either in the throughput, the number of extractable BLOBs or the type of features they can detect.

The throughput is limited by the number of concurrent processed pixels and the maximum clock frequency of the hardware implementation. Concerning the number of extractable BLOBs, most algorithms have preconditions for the image content.

Either they require the foreground-background ratio to be below a certain limit, or they cannot handle large numbers of small BLOBs. It is crucial that if the preconditions in a hardware-implementation are not met, the algorithm fails. In a software implementation, not meeting such preconditions would result in an increase of processing time, thus not meeting the timing requirements but still calculating correct results. The image processing is still possible, but with lower quality.

In hardware implementations, these preconditions are set on the base of the hardware-component's sizes. In particular, the size of buffers between processing elements are the main limitation and the base of those preconditions. In consequence, if the precondition is not met, a buffer overflow occurs, resulting in wrong results for the BLOB features. If the buffers are monitored for overflows, the system can detect this failure, but not correct or overcome it. Summarizing, a hardware implementation will either give wrong results or no results for the BLOB extraction if the preconditions are not met.

Besides these preconditions for the image content, most algorithms have severe preconditions for the image processing pipeline. Almost every reported algorithm relies on considerably long line blanking and frame blanking times. Line blanking is the time between the readout of two lines by the image sensor. Traditionally, these line blanking times are several tenths or hundreds of clock cycles long, due to the physical readout procedure in the camera. Frame blanking is the time between the end of one frame (i.e. image) and the start of the subsequent one. Frame blanking

times, just as line blanking times, are traditionally several tenths or hundreds of clock cycles long.

With the advancements made in camera-chips, this precondition is no longer feasible. The readout speed of modern vision sensors exceeds the transfer speed of the transfer interfaces. To obtain the maximum readout-speed, modern high-speed cameras facilitate pixel-buffers between the vision sensor and the transfer interface. As a consequence, the readout clock of the camera is set higher than the clock of the transfer interface. As a result of this, the physical line-blanking and frame-blanking times are no longer carried over to the transfer interface; the transfer interface will read the buffer content during these times. This results in line-blanking and frame-blanking times of only a single clock cycle for the BLOB algorithm.

A novel algorithm was developed aiming to overcome the identified shortcomings of the existing implementations. After defining the goals for the algorithm, a pipeline-based implementation based on the current state-of the art was developed (Diederichs, 2010, 2011). While this implementation was sufficient for simple BLOB detection tasks, it had the same limitations concerning preconditions on the image content and the line-blanking times. Experiments trying to remove those preconditions with the existing design were not successful. Consequently, a new approach without these preconditions was developed. This novel implementation is described in the following chapter.

The new algorithm has to be independent of frame-blanking and line-blanking times. It should use as few resources as possible and work with all kinds of images.

The developed approach differs from any other reported approach in several ways.

In PC-based connected component labeling, the labeling and the feature calculation is separated and performed independently. Also, the first label detection and the merging of labels is performed independently. The single-pass algorithms became possible by changing this approach. Single-pass algorithms perform the first pass and pause the first pass calculation to obtain the merging of labels. Additionally, the features are calculated on the fly and need to be merged as well. Depending on the complexity of the image, this pause can lead to a major backlog, as the image data coming from the camera is not paused. Most algorithms rely on long line-blanking times or large-background-areas to process this backlog.

The algorithm described below overcomes this paradigm by further separating the three steps: first pass, equivalence handling and feature calculation. Additionally, feature calculation is separated into different steps. The main idea is to perform feature calculation based on the regions detected by the first pass. The first pass and feature calculation is pipelined without buffers. This way, both can run at full speed with the pixel-clock, even if every pixel needs to be handled and the

line-blanking and frame-blanking times are one clock cycle, resulting in a higher throughput.

The algorithm includes a novel hardware equivalence handling. The equivalence handling is a major contribution to the robustness of the algorithm. It is performed in parallel to the first pass and feature calculation and allows for dynamic growing and shrinking of equivalence tables and the reuse of labels.

Furthermore, the novel approach performs the calculation on several pixels in one clock cycle. By doing this, the throughput and therefore the update-rate was significantly increased compared to the state of the art. First, the algorithm was developed to calculate three pixels in one clock cycle. Later, this was extended to nine pixels in one clock cycle, allowing for throughputs of up to 1.93 GB/s.

To allow for better classification, three novel BLOBs are introduced in this work. These extend the state of the art for hardware-based BLOB detection and were first reported by the author in Diederichs (2010); Diederichs et al. (2012); Diederichs and Fatikow (2013).

After describing the goals and requirements for the new algorithm, the novel implementation details of the algorithm are described. All of the following chapters are part of the original work of this thesis. First, the novel BLOB features are described. The novel equivalence handling is explained in section 3.3. In section 3.4, the hardware architecture of the novel approach is revealed. The following section explains the changes that were made to the architecture to allow for an even higher throughput, by changing the number of concurrently calculated pixels from three to nine. In section 3.6 the implementation- and test strategy is described. The chapter concludes in section 3.7.

3.1 Requirements and goals

The overall goal was to develop a motion tracking algorithm for microhandling tasks. Image processing is performed in order to do motion tracking for closed-loop robot control. For closed-loop control, the timing characteristics of the image processing are of great interest.

One main characteristic of a sensor is its update-rate. A higher update-rate leads to more frequent information updates for the closed-loop controller and thus to better and faster positioning. Therefore, the motion tracking algorithm should allow for high throughput.

For an FPGA-implementation this either means fast clock frequencies (i.e. above 100 Mhz) or the processing of multiple pixels in one clock cycle. It is not possible to formulate a requirement for the update-frequency, as it does not only depend on

the image-processing algorithm, but also on the camera-resolution and the camera readout-speed.

However, a requirement for the throughput can be formulated. The throughput is defined by the number of pixels that can be processed in one second (P/s). The motion tracking algorithm shall have a throughput of at least 680 MP/s. This update-rate corresponds to the update-rate of the CameraLink standard in its full configuration. With this throughput, the motion tracking algorithm can operate on a 1 MP image with an update-rate of 680 Hz.

As stated in section 1, one major issue for closed-loop controllers with image-based sensors is the update-rate deviation (update-rate jitter). Therefore, the motion tracking algorithm is required to provide a very stable update-rate. The standard deviation of the sensor signal is aimed to be below 1 % of the update-rate. An update-rate of 100 Hz corresponds to an update interval of 10 ms. For an update-rate of 100 Hz, the required jitter must be below 0.1 ms.

The last timing constraint, which is derived from the requirements of the closed-loop controller, is a predictable signal latency. The latency should be as short as possible, as it describes the age of the information obtained by the sensor. However, predictability is even more important. If the closed-loop controller knows exactly about the latency, it can compensate for it. Therefore, the latency jitter should be as low as possible, but at least below 1 % of the overall latency. The latency itself should be below 110 % of the time needed for exposure and image transfer.

The update-rate of a camera is defined by the readout time of the sensor. While motion tracking is performed on the whole image, it cannot be faster than the readout time of the sensor.

Table 3.1 summarizes all timing requirements for the image-processing algorithm.

Requirement	Value	Unit
Throughput	680	Mb/s
Update rate jitter	1	% of update-rate
Latency	110	% exposure and transfer time
Latency jitter	1	% of latency

Table 3.1: Requirements for the image-processing algorithm.

Furthermore, the algorithm should extend the state of the art in terms of it's classification features. For highly accurate position detection based on LEDs,

a method that is more exact than the center of gravity should be available. Furthermore, it should be possible to detect the main direction of an object that is elongated as well as the aspect ration of such objects.

3.2 Features

To allow for a more accurate classification, additional features are used by the algorithm. The features extend the state of the art for hardware-based BLOB detection and are part of the authors contribution.

Three new features are introduced. First, The weighted center of gravity can be used to calculate the center of an object more precisely. This feature was introduced by the author in Diederichs (2010).

Hardware implementations cannot easily calculate an objects contour or the convex hull, because it needs a dynamic memory size for it's features (see section 2.3.2). However, it is possible to calculate several bounding rectangles, rotated by different angles. By creating the intersections of these rectangles, a rough approximation of the convex hull can be calculated.

Lastly, current features are not able to give general information about the main direction of the object or the aspect ratio. Knowing the aspect ratio and the main direction of an object is a huge benefit when objects need to be classified and the angle of the object must be tracked. The last two features were already introduced by the author in Diederichs et al. (2012); Diederichs and Fatikow (2013).

3.2.1 Weighted center of gravity

The weighted center of gravity is an extension of the center of gravity calculation. Instead of only using the x and y coordinates, the pixel intensity value is added as a weight (Dahmen et al., 2008).

$$\begin{aligned} x_{wcog} &= \frac{\sum_{i=0}^{n} w(p_i) \cdot x(p_i)}{\sum_{i=0}^{n} w(p_i)} \\ y_{wcog} &= \frac{\sum_{i=0}^{n} w(p_i) \cdot y(p_i)}{\sum_{i=0}^{n} w(p_i)} \end{aligned} \tag{3.1}$$

The pixel intensity can be used as a weight $w(p_i)$. If the image is thresholded before the object detection, a different approach would be to use the intensity minus the threshold. The two approaches lead to different results. Which approach is the

best depends on the application. For binary images, the weighted center of gravity produces the same results as the center of gravity.

The weighted center of gravity can be used to retrieve a more precise center of an object if the object's intensity in higher in the center than on the edges. This is i.e. the case for LEDs (see figure 5.2, page 113).

3.2.2 Bounding polygon

For a bounding octagon, the bounding box as well as a rectangle rotated by 45° is calculated. The rotated rectangle consists of four edges:

- $\min(x + y)$ (top left edge)
- $\max(x + y)$ (bottom right edge)
- $\min(x - y)$ (bottom left edge)
- $\max(x - y)$ (top right edge)

The minimum and maximum values of $x + y$ and $x - y$ can be updated and merged in the same way as the bounding box.

In a post processing step, the rotated rectangle can be intersected with the bounding box. The bounding octagon is a very rough approximation of the convex hull.

The approximation can be improved by calculating finer structures, using more and differently rotated rectangles (e.g. hexadecagon).

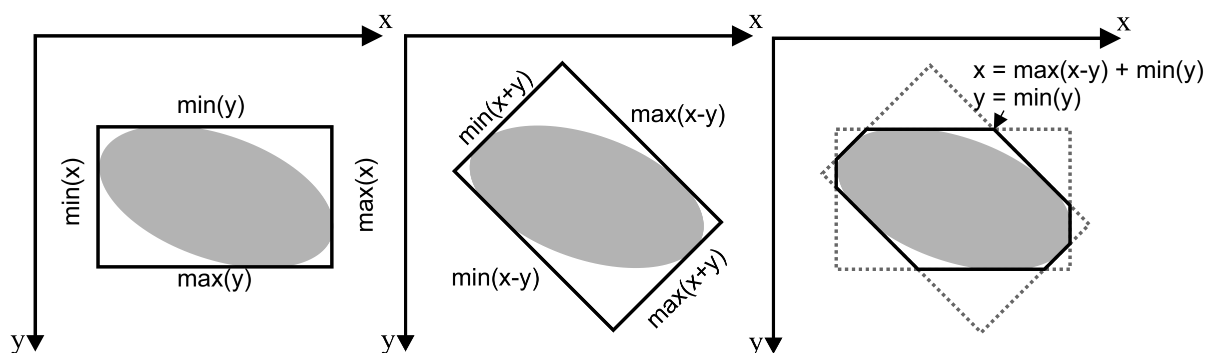


Figure 3.1: Example of a bounding octagon. The bounding box and the rotated rectangle are intersected. The resulting octagon is a rough estimation of the convex hull.

3.2.3 Principal component analysis based features

The principal component analysis (PCA) is a method for data reduction and is well known in image processing. However, it can also be used to detect the main axis of an object (Lee and Koo (2006)). A factor PC_e which describes the relationship between the expansions of the main axis and its orthogonal axis can also be derived from the PCA (Wortman and Fatikow (2009); Rodrigo et al. (2006)); this factor can be used to detect aspect ratios. For object classification, both of these features are of great interest. The computation of the PCA is a complex task and can not easily be separated into simple computations.

The PCA is computed from the covariance matrix of all points in one component:

$$C = \begin{pmatrix} \operatorname{cov}(x,x) & \operatorname{cov}(x,y) \\ \operatorname{cov}(y,x) & \operatorname{cov}(y,y) \end{pmatrix} \tag{3.2}$$

The covariance is defined as follows:

$$\operatorname{cov}(x,y) = \operatorname{cov}(y,x) = \frac{\sum_{i=0}^{n} (x_i - \bar{x}) \cdot (y_i - \bar{y})}{n-1} \tag{3.3}$$

As stated in section 2.3.2, the incremental computation of features must be possible and achievable in one clock cycle. The main goal is to calculate the covariance during the update and merge process. The final PCA calculation based on the covariance matrix can be done in the post-processing step. In equation 3.3, the local value of a pixel as well as the average value are inside the sum.

This can not be calculated incrementally, as $\bar{x}$ and $\bar{y}$ are unknown until the entire component has been processed. To be able to calculate the covariance matrix the desired way, the $\bar{x}$ and $\bar{y}$ are not allowed inside the sum. Starting from equation 3.3, a sequence of transformation steps will convert the equation into the desired form. The denominator of equation 3.3 is the number of pixels, which is a feature itself, minus one. In the following, only the numerator will be transformed.

$$
\begin{aligned}
N &= \sum_{i=0}^{n}((x_i - \bar{x}) \cdot (y_i - \bar{y})) \\
&= \sum_{i=0}^{n}(x_i y_i - \bar{x} y_i - x_i \bar{y} + \bar{x}\bar{y}) \\
&= \sum_{i=0}^{n}(x_i y_i) - \sum_{i=0}^{n}(\bar{x} y_i) - \sum_{i=0}^{n}(x_i \bar{y}) + \sum_{i=0}^{n}(\bar{x}\bar{y}) \\
&= \sum_{i=0}^{n}(x_i y_i) - \bar{x}\sum_{i=0}^{n}(y_i) - \bar{y}\sum_{i=0}^{n}(x_i) + n\bar{x}\bar{y} \qquad (3.4) \\
&= a - b - c + d \qquad (3.5)
\end{aligned}
$$

After the transformation, the numerator consists of four parts. In equation 3.4, $\bar{x}$ and $\bar{y}$ can be substituted with their actual computation:

$$
\begin{aligned}
\bar{x} &= \frac{\sum_{i=0}^{n} x_i}{n} \\
\bar{y} &= \frac{\sum_{i=0}^{n} y_i}{n} \qquad (3.6)
\end{aligned}
$$

Substituting $\bar{x}$ in the second term (term b) of equation 3.4 will result in equation 3.7:

$$
\begin{aligned}
b &= \frac{\sum_{i=0}^{n} x_i}{n} \sum_{i=0}^{n}(y_i) \\
b &= \frac{\sum_{i=0}^{n} x_i \sum_{i=0}^{n} y_i}{n} \qquad (3.7)
\end{aligned}
$$

Substituting $\bar{y}$ in the third term of equation 3.4 produces the same result (equation 3.8).

$$
\begin{aligned}
c &= \frac{\sum_{i=0}^{n} y_i}{n} \sum_{i=0}^{n}(x_i) \\
c &= \frac{\sum_{i=0}^{n} x_i \sum_{i=0}^{n} y_i}{n} \qquad (3.8)
\end{aligned}
$$

Substituting $\bar{x}$ and $\bar{y}$ in the fourth term of Eq. 3.4 results in a similar equation:

$$\begin{aligned} d &= n\frac{\sum_{i=0}^{n} x_i}{n}\frac{\sum_{i=0}^{n} y_i}{n} \\ d &= \frac{\sum_{i=0}^{n} x_i \sum_{i=0}^{n} y_i}{n} \end{aligned} \tag{3.9}$$

Using the equations 3.7, 3.8 and 3.9, the final equation for the transformed covariance computation can be created:

$$\begin{aligned} N &= a - b - c + d \\ & \quad b = c = d \\ N &= a - b \\ N &= \sum_{i=0}^{n} (x_i y_i) - \frac{1}{n} \sum_{i=0}^{n} x_i \sum_{i=0}^{n} y_i \\ \mathrm{cov}(x,y) &= \frac{\sum_{i=0}^{n} (x_i y_i) - \frac{1}{n} \sum_{i=0}^{n} (x_i) \sum_{i=0}^{n} (y_i)}{n-1} \end{aligned} \tag{3.10}$$

In equation 3.10, there are three sums that can be updated and merged in the desired way. The divisions and subtractions of the sums are done in the post-processing step.

For all elements of the covariance matrix, five different sums have to be computed during connected component labeling. In addition to the three sums shown in equation 3.10, two more sums have to be computed for $\mathrm{cov}(x,x)$ and $\mathrm{cov}(y,y)$. However, only the first term in the numerator of equation 3.10 introduces a new sum, the other sums can be reused for all covariance equations; they are also used for the center of gravity calculation.

Therefore, to calculate the basic elements for the covariance matrix during the hardware BLOB detection process, only three additional sums have to be calculated and stored.

With the above described novel features, objects can be classified more precisely. The weighted center of gravity allows for a more precise position detection of objects where the grayscale characterizes an intensity. The bounding polygon can be used to get an approximation of the convex hull. This can be used together with the number of pixels to deduce information about the objects shape. The PCA-based features allow for the detection of a main direction of objects as well as aspect ratios. The aspect ratio is a valuable indicator for whether a specific

shape (e.g. a gripper jaw as shown in figure 3.2) needs to be detected while the main direction gives additional information about the object's alignment.

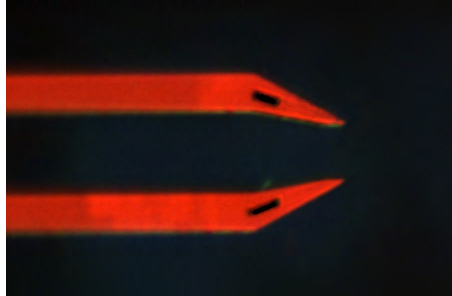


Figure 3.2: Micro gripper in a microhandling setup (Diederichs and Fatikow, 2012). If the gripper jaw is tracked, PCA based features are of great help.

3.3 Equivalence handling

One main challenge for hardware-based connected component labeling is the equivalence handling (see section 2.3).

In the presented implementation, the equivalence handling is performed in a process parallel to the actual labeling. Several optimizations allow this process to perform with a very small buffer even on complicated images. Furthermore, an additional system keeps track of the label appearance and detects if a component-label is not assigned anymore. This information is used to shrink the equivalence tables and to perform merging of feature data. Additionally, these labels can be used again for new components, allowing for smaller RAMs at several points in the design. This implementation was developed by the author during this thesis and is part of the contribution.

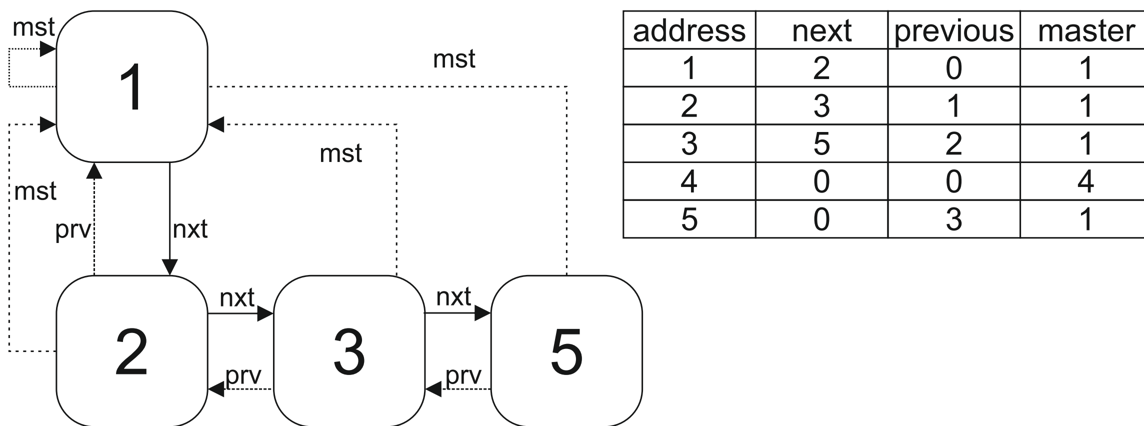
In this section, the original idea and the operations of the novel equivalence handling are described. The actual implementation details are given later in this chapter.

The equivalence information is kept in a disjoint set. A disjoint set is a data structure that organizes its content into non-overlapping subsets (Galler and Fisher, 1964; Galil and Italiano, 1991). Every subset has a *representative*, called *master* hereafter. Several operations have to be performed on the disjoint set.

The disjoint set is implemented as a mixture of a vector and a linked list. The pointers of an element are stored at its corresponding RAM address. In this

algorithm, the labels are represented as positive integers, starting with 1. The number 0 is used as a null, meaning a *not used* or *not valid*. Therefore, storing the information about label 3 in a memory at address 3 is straightforward and allows for fast lookups. Consequently, the pointers of a linked list represent the memory address and the ID of the label at this address, again allowing for fast lookup and optimizations.

figure 3.3 shows a subsection consisting of four elements. The linked list has next and previous pointers, allowing for the iteration of the list in either direction. Additionally, every element in the linked list has a pointer to the master element. Therefore, it is possible to find the master of each element with only a single lookup.



address	next	previous	master
1	2	0	1
2	3	1	1
3	5	2	1
4	0	0	4
5	0	3	1

Figure 3.3: Example of a subset in the disjoint set. Every element has three pointers: to the next element, to the previous element and to the master element. The master pointer is not allowed to be 0. If the next pointer is zero, there are no more elements in that list. The master always has a previous pointer of 0, every other element has a valid previous pointer. On the right side the memory content for this subsection is shown.

Several operations need to be performed on the disjoint-set. If a new label is assigned, it is created as a new subset. The next and previous pointer of the element are set to 0 and the master element is set to itself.

If two labels a and b need to be joined, it must first be checked if they already belong to the same subset. To check this, the masters of a and b need to be found. In a normal linked list, the list would have to be iterated until the first element of the list has been found. The extra pointer to the master of each subset makes

it possible to derive the master directly. If the masters of a and b are equal, no additional step has to be performed. If they are distinct, the two subsections have to be united.

During the union process, the master pointers of one of the subsets have to be updated to the new master. The union is performed by inserting the linked list of the subset B between the master of A and the first element of that subset (see figure 3.4). The usual way would be to append one of the lists to the other. However, by inserting one list directly after the master of the other, the number of steps that need to be performed is reduced. When one subset is appended, the last element of subset A has to be found. Then, the next pointer of that element needs to be changed. Following, the master and previous pointers of subset A need to be updated. In figure 3.4, this would be three steps to get from element 1 to element 5. Afterwards, the next pointer of element 5 needs to be changed. Lastly, the previous pointer and the master pointer of element 4 need to be changed and the master of element 6, resulting in 6 steps. During this process, every element of each subset has been read once. If subset B is inserted after the master of A, only two elements of A have to be read and updated. Especially for larger subsets, this strategy is faster. The decision which master is kept is dependent on several factors that are discussed later in this section.

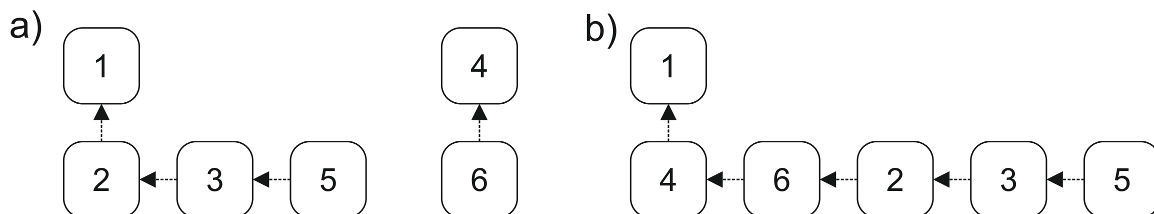


Figure 3.4: Union of two subsets in the disjoint set. a) The two disjoint subsets with the masters 1 and 4. b) The new subset after the union. The second subset was inserted after the master of the first one. The master pointers of the elements 4 and 6 where updated to the new master.

As stated earlier in this section, the subsections can be shrinked when a label is discontinued. If it is discontinued, its feature data must be combined with the feature data of a different label. It is only possible to merge feature data if both labels are discontinued, the reason for this will be discussed in the next section. Therefore, a subset may only be shrinked if at least one other element of the subset is already discontinued. To keep track of this easily, only the master of a subset can be a discontinued element. Therefore, it must be possible to exchange the master of a disjoint set.

If a discontinued element is found, it is first checked to see whether the element is already a master. If this is the case, the element is flagged as finished and nothing more happens. If it is not the master of the subset, it is checked for whether the master is already flagged as finished. If this is not the case, the master will be swapped with the finished element. To do this, the element is removed from the linked list by altering the pointers of the sibling elements in the list. Then, the pointers of the removed element are updated to be a single independent subset and flagged as finished. After this step, the subset has been divided into two distinct subsets. As a next step, the newly created subset will be united with the old subset. During this operation, the united subset will contain the same elements as before, but the master will have changed (see figure 3.5).

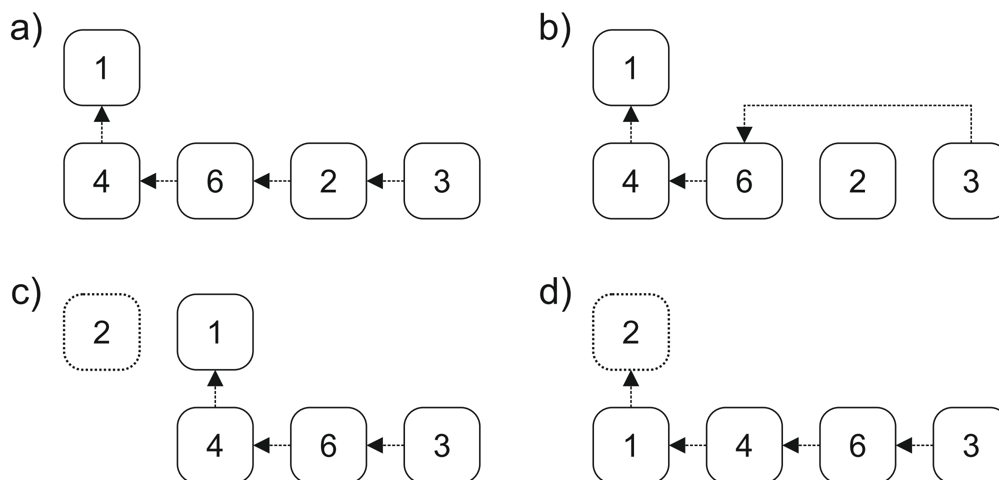


Figure 3.5: Master exchange of a subset in the disjoint set. a) The disjoint set before the transformation. Element 2 has been found discontinued. b) The previous and next pointers of elements 6 and 3 are changed to omit element 2 in the linked list. c) Element 2 is updated to a single subset of only one element. It is also flagged as finished. d) The two subsets are united to one subset. The new master of this subset is the finished element 2.

If a discontinued element is found and its master is already flagged as finished, the discontinued element will be removed from the subset. This is done by adjusting the next and previous pointers of the element's siblings (see figure 3.6). If only the master remains after the removal, all elements of this subset are finished. In other words, all labels that belong to the same region are discontinued and the region is finished and cannot grow anymore. Thus, the subset will be entirely removed. How the feature information is handled is discussed in the next section.

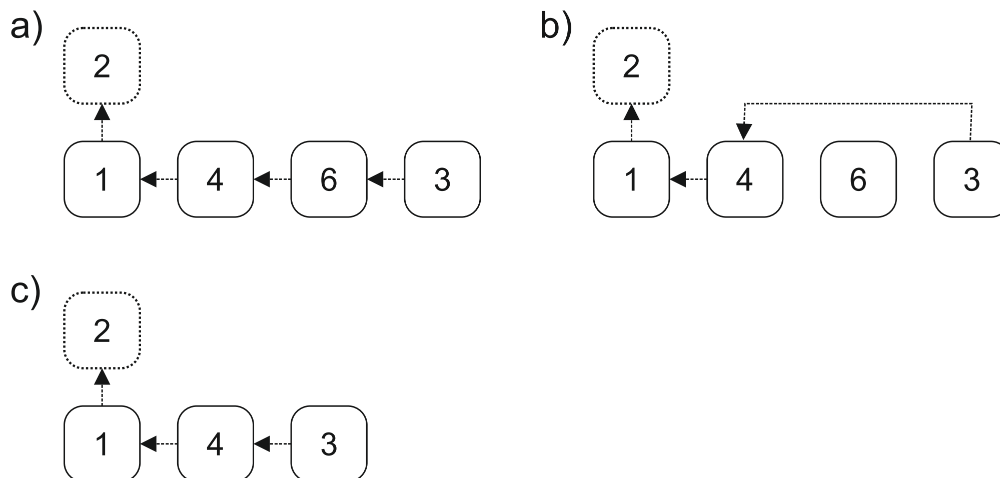


Figure 3.6: Shrinking of a subsection in a disjoint set. a) The subsection with a finished master. Element 6 has been found discontinued. b) The next and previous pointers of elements 4 and 3 are updated. c) The element is removed. Its feature data is joined to the feature data of element 2.

The information about a finished master can be used to make decisions when uniting two subsets. If one of the subsets has a finished master, this will be the new master for the united subset. This prevents changing the master at a later state and, more importantly, it prevents having finished elements in the subset at a different position than the master position. However, if both subsets have a finished master, a special treatment must be performed. One of the masters will be removed entirely from the new subset, while the other master will be the master of the new subset (see figure 3.7). The features data of the removed master will be joined to the feature data of the new master; this will be discussed in the next section.

3.3.1 Implementation

The equivalence disjoint set is stored in a dual ported RAM. This RAM contains the next, previous and master pointers of each element. Additionally, a small RAM exists to store the finished flags of masters. The different operations performed on the disjoint set are programmed in a Mealy state machine. This state machine is optimized to perform every operation in as few cycles as possible. Furthermore, it is designed in such a way that it uses both ports of the dual-ported RAM whenever possible. The state-machine is shown in figure 3.8. The state-machine begins in a *startup* state. During this startup, a buffer containing all unused labels is filled with all possible labels. This is done only once, later the labels will be reused all the time. When the startup is finished, the main sequence is started. The main

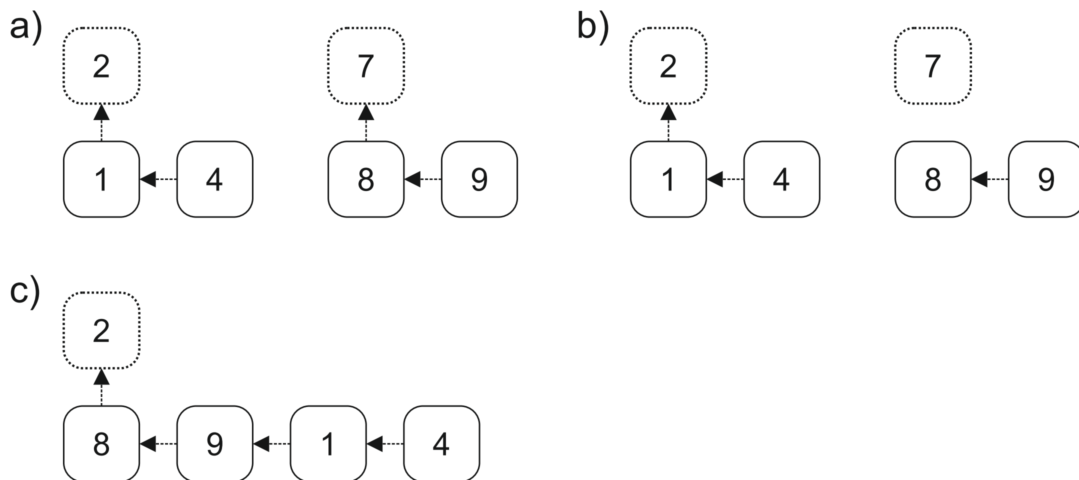


Figure 3.7: Union of two subsets that both have a finished master. a) The subsets that will be united. b) Element 8 and 9 are detached from their current master. c) The elements 8 and 9 are merged into the other list. The former master 7 has been removed. Its feature data is joined to the feature data of element 2.

sequence always starts with a check of the incoming data: if there is information about joining regions, the *checkJoin* state is entered. If no joining information is available, but information about a discontinued label, the *getFinished* state is entered. If no new information is available, the *idle* state is entered. When in the idle state, the system will continue with the check in the next cycle.

The right side of the state-machine consists of states handling the union of elements and subsets. The left part of the state-machine consists of states handling the shrinking of subsections. The full state machine is shown in figure 3.8. In the following, the different operations that are performed on the disjoint-set are described together with the states that are involved in the operation. Later, all the states will be described one by one to give a better understanding of the single steps of the whole state-machine.

Creating and merging subsets

To create a subset or merge a subset, a queue containing two labels is checked by the state machine. If one of the labels is valid and the second one is zero, a new subset is created for the valid label. If both labels are valid and distinct, the subsets of those labels need to be united.

The creation of new subsets is handled in the *checkJoin* state. The state checks whether the incoming information is a join operation or a new label. If it is a new

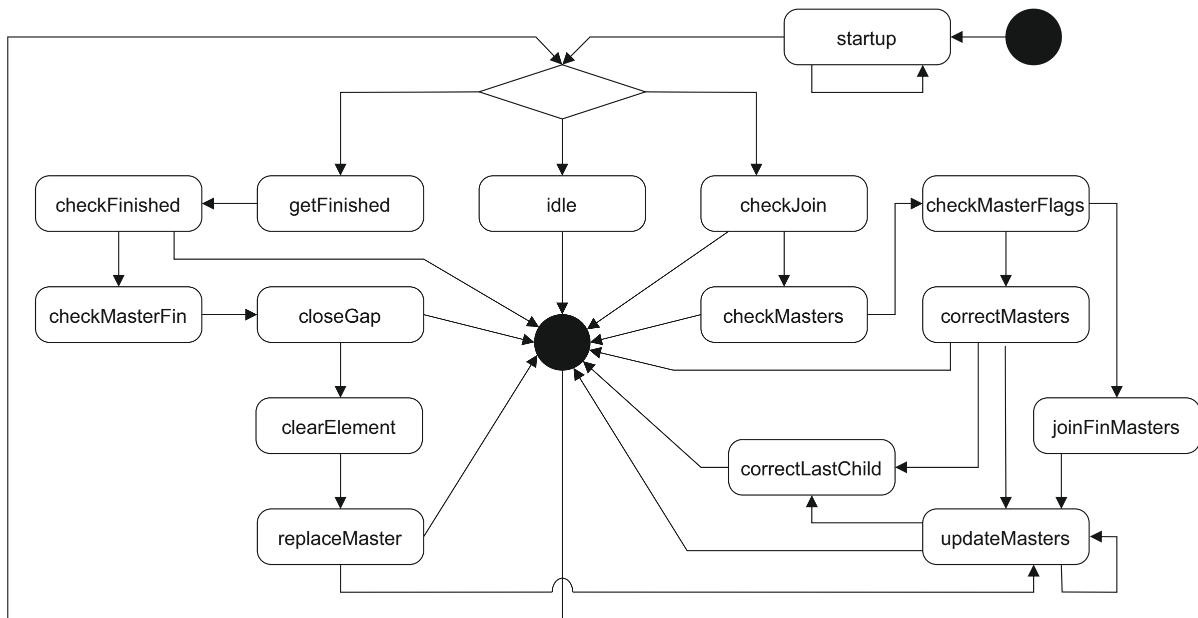


Figure 3.8: State machine of the equivalence handling hardware system. The state machine is optimized to perform every operation in as few cycles as possible. Every operation starts with the incoming data check. If a new label or a label join needs to be performed, the *checkJoin* state is entered. The right part of the state-machine is designed to handle the union of subsets. The right part is designed to handle the shrinking of subsets. If a finished label is found, the *getFinished* state is entered. In every other case, no operation needs to be performed and the idle state is entered.

label, the information is written to the RAM and the next state will be decided by the incoming data check. Additionally, the finished flag of this label will be set to false.

If both incoming labels are valid, the address lines of the RAM are set to the label IDs, to receive their next, previous and master pointers for the next state. The next state will be the *checkMasters* state.

If the *checkMasters* state is entered, the output lines of the dual ported RAM contain the pointer information for the two labels that need to be joined. For each label, the information about the label's master, next and previous element can be used to decide the next step.

As a first step, it must be checked if both labels belong to the same subset. This can be checked by comparing the master pointers of both labels. If the master pointers are the same, the labels belong to the same subset and were joined earlier. The next state will be decided by the incoming data check.

If the masters are different, the labels belong to different subsets. In this case, these subsets must be united. Several different situations can occur for the subsets. When a subset is created for the first time, it consists of only one element. The most simple but very common case is that two subsets consisting of one element each need to be joined. This special case is shown in figure 3.9. One of the elements becomes the new master of the united subset. This special case can be detected by analyzing the master and next pointers of the labels. A label is a single element subset if the master pointer points to itself and the next pointer is invalid. This can be checked during the *checkMasters* state. If this situation occurs, one RAM address line is used to update the next pointer of the element that will be the master of the new subset. The second address line is used to update the master and previous pointers of the other element. In this simple case, no other steps need to be performed before the next operation can be performed. Therefore, the next state will be decided by the incoming data check.

In every other situation, the union of the subsets is dependent on more factors. More information about the masters of both subsets are needed. Therefore, the address lines of the disjoint-set RAM as well as the address lines of the finished flag RAM are set to the master pointer values. By doing this, the pointer and finished information of the masters are available during the next state.

figure 3.10 shows the default operation for the union of two subsets that consist of more than one element. As stated before, after the *checkJoin* state the information about the labels, that should be joined, is available. Now, during the next state (*checkMasters*), the information about the master element of each subset is available. To join the two subsets, the pointers of the elements must be adjusted. After the adjustment, the resulting subset must consist of a linked list containing all

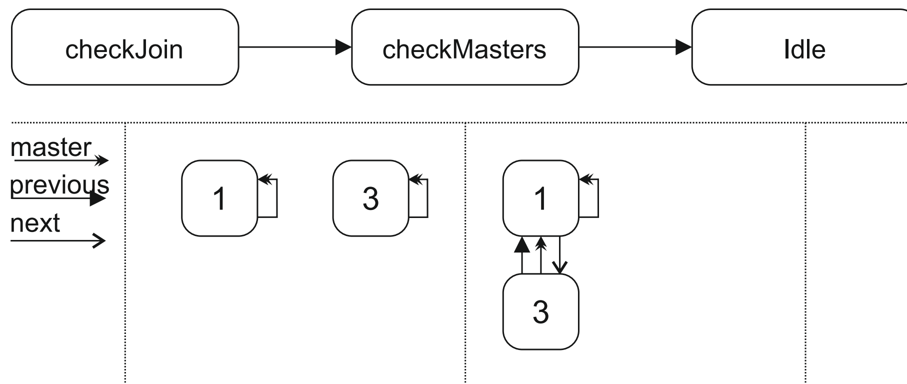


Figure 3.9: The union of two subsets consisting of only one element. After the *checkJoin* state, the information about elements 1 and 3 is known. In the *checkMasters* state, it is evaluated that both elements are single element subsets by checking their next pointers. Then, the pointers of both elements are altered by using both ports of the disjoint-set RAM.

elements of both subsets. One element must be selected as master. The master element has to be the first element in the linked list and the master pointers of all elements in the list must point to this master. The process is to insert one of the subsets between the master of the other and its first child. The whole process and the states involved are described by an example in figure 3.10. If the subsets are consisting of more than two elements, the sequences repeat only the *updateMasters* state and goes to the *correctLastChildState* once the last element of the list has been corrected. This can be done by checking the next pointer of the elements.

The first discontinued label of a subset will become the new master element of the subset. This information must be used when uniting two subsets. It must be avoided that a master that is already flagged as finished becomes a child element. Therefore, the merging process described above should keep a finished master as the new master of the united subset.

If both masters are finished, the union process works differently. In this case, one of the finished masters will be removed from the united subset while the other one is kept as the new master. This process is shown in figure 3.11. The different situation is reflected by entering a different state after the *checkMasterFlags* state. However, this is the only difference. As before, if the second subset consists of more than two elements, the process needs more iterations of the *updateMasters* state.

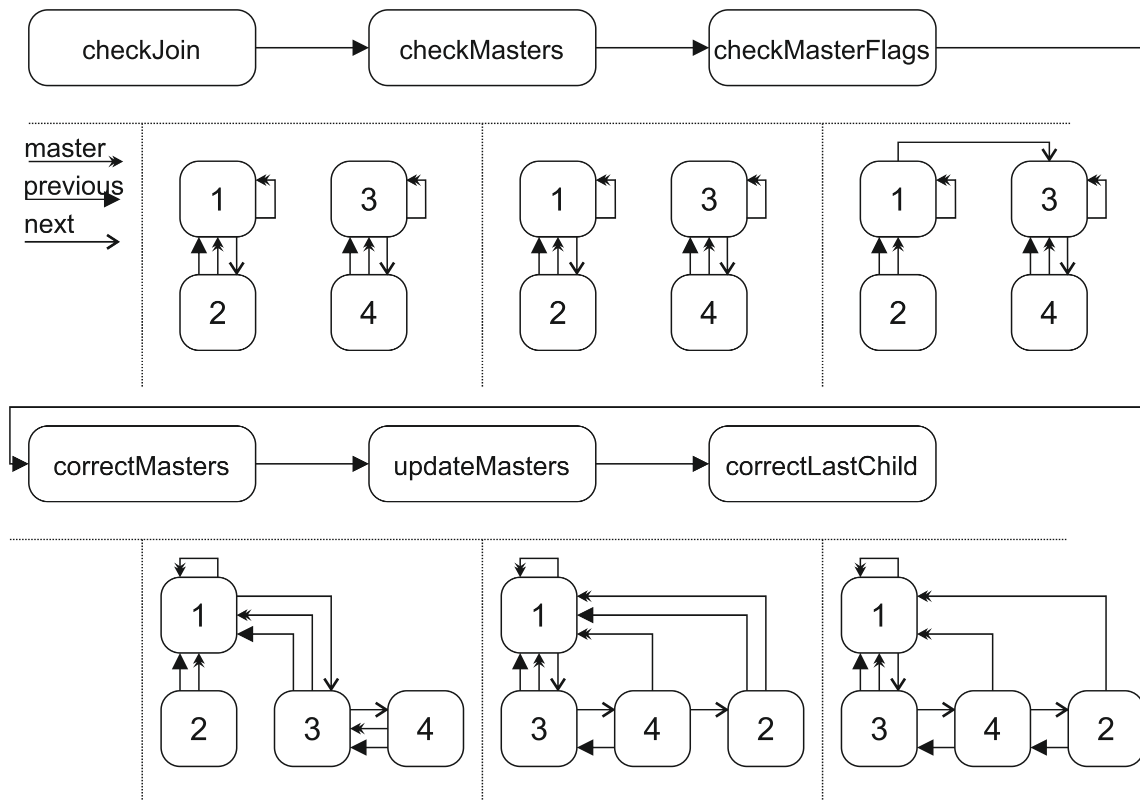


Figure 3.10: The union of two subsets consisting of two elements each. After the *checkJoin* state, the elements that have to be joined are known (e.g. 2 and 4). These are read from the disjoint-set RAM during the state change. In the *checkMasters* state it is checked whether the masters are equal and whether these are one-element subsets. If this is not the case, the master elements (in this case 1 and 3) and their finished flags are read during the state change. In the *checkMasterFlags* it is checked if any of the masters is finished. If one or none is finished, the next pointer of the master that will be kept is set to the other master during the state change. The *correctMasters* state updates the previous and master pointers of the element 3, and the *updateMasters* state updates the master and next pointers of element 4. In the last state, the previous pointer of element 2 is corrected.

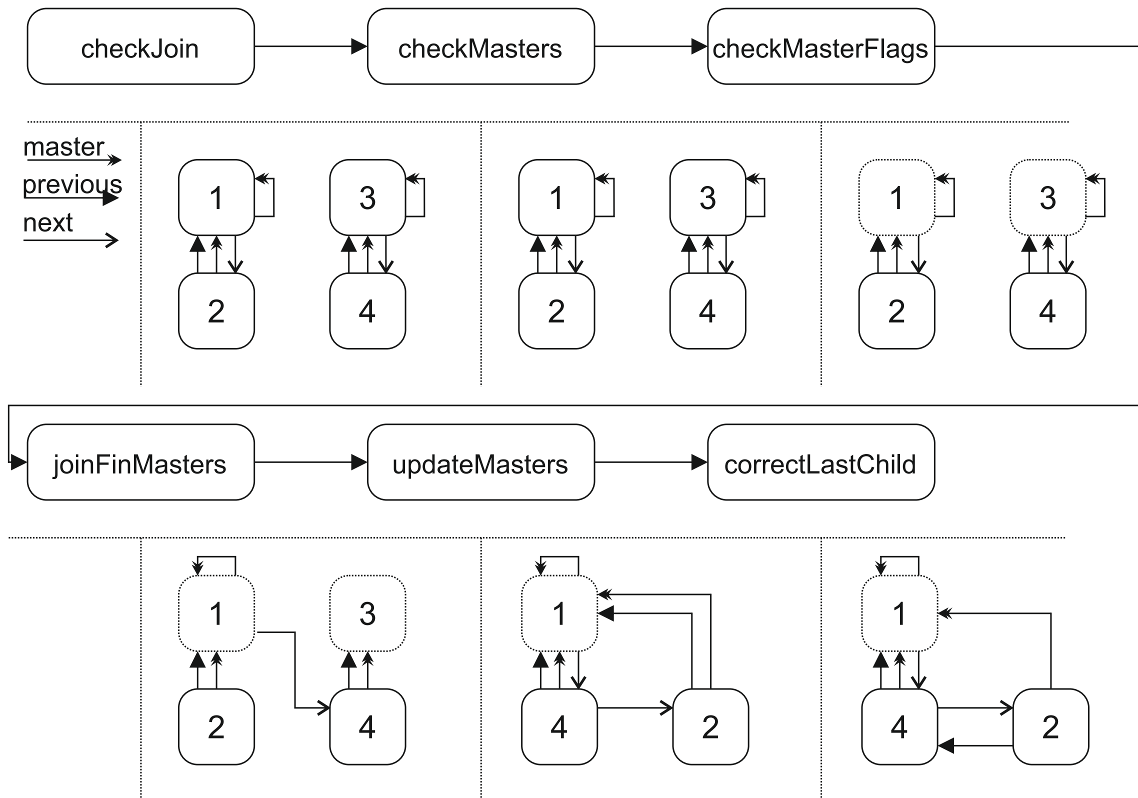


Figure 3.11: The union of two subsets consisting of two elements each where both masters are finished. The first steps are analogous to the steps in figure 3.10. In the *checkMasterFlags* it is checked if any of the masters is finished. If both are finished, a different state is entered. The *joinFinMasters* state updates the next pointer of the master that is kept. The *updateMasters* state updates the master, next and previous pointers of element 4. In the last state, the previous pointer of element 2 is corrected. Element 3 is not used anymore.

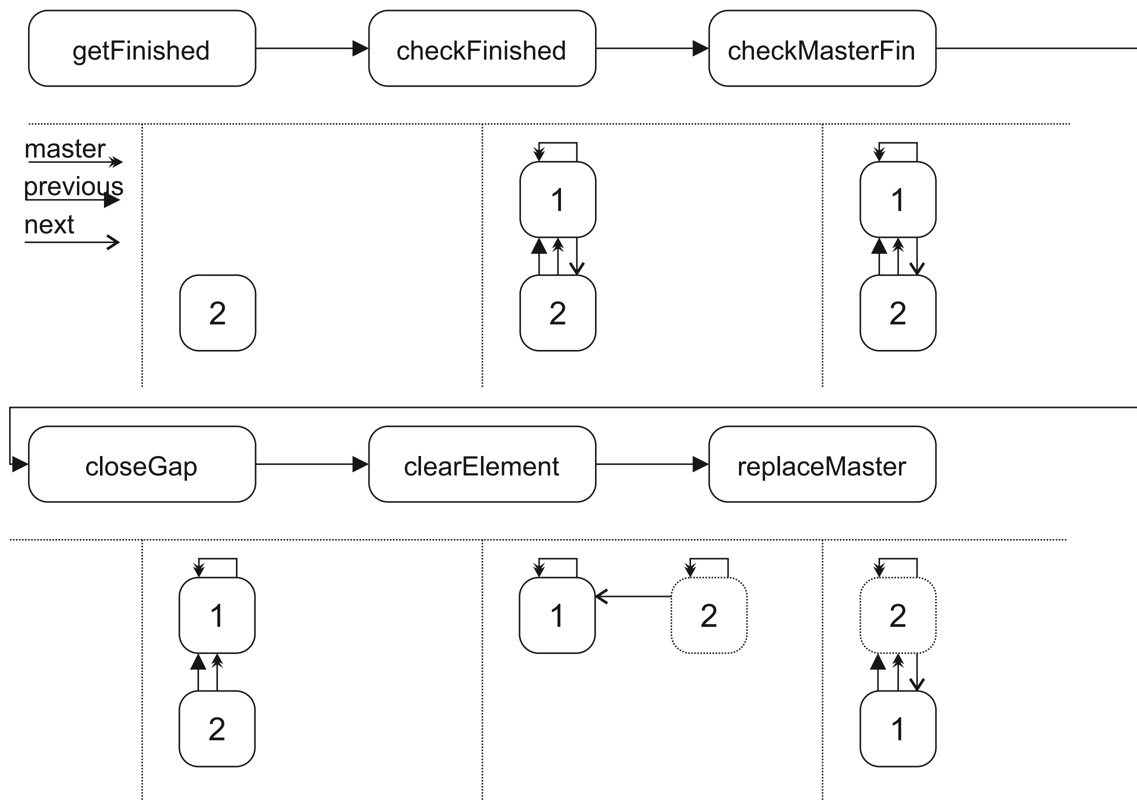


Figure 3.12: Switching of masters when a child is finished. After the *checkFinished* state, the master of the finished element 2 is known. After the *checkMasterFin* state, the finished flag of the master is available. The *closeGap* state removes the next pointer of element 1, removing element 2 from the subset. The *clearElement* state changes all pointers of element 2, making it a master and changing the next pointer to element 1. Additionally, its finished flag is set. The *replaceMaster* state replaces the master and previous pointer of element 1.

Shrinking and removing subsets

The first discontinued label of a subset will become the new master element of the subset. For this reason, a switch master process is supported by the state machine. Figure 3.12 shows the exchange of a master and the involved states in a subset consisting of two elements. Here, the pointers of the elements are completely reversed, as the old master becomes the first child and the first child becomes the new master.

If the subsection consists of more than two elements, the discontinued element is removed from the linked list. Then, it is inserted at the beginning of the list. Afterwards, all master pointers are adjusted to the new master. As this procedure is identical to the last when uniting two subsections, the same state is used. Figure 3.13 shows the procedure and the involved states.

Once a subset has a finished master, every other discontinued label of the subset is removed from the subset. The state-machine sends out flags for merging the feature data of the labels, this will be described later. To remove the element, discovering whether the master of the subset is already finished is necessary.

Once the last child of a subsection is removed, the whole region is finished. Then, the subset is entirely removed from the disjoint set. During this process, flags to send the cumulated feature data of the region are set.

Individual state description

The state-machine begins in the *startup* state. During this startup, a buffer containing all unused labels is filled with all possible labels. This is done only once, later the labels will be continually reused. The startup state starts with label ID 1 and increments the label ID that is pushed into the buffer in every cycle until the maximum label ID is reached. When the startup is finished, the main sequence is started. The main sequence always starts with a check of the incoming data: if there is information about region joining, the *checkJoin* state is entered. If no joining information is available, but information about a discontinued label, the *getFinished* state is entered. If no new information is available, the *idle* state is entered. When in the idle state, the system will continue with the check in the next cycle.

A newly used label is represented as a join of the new label with the label 0. Therefore, the creation of new subsets is handled in the *checkJoin* state. The state checks whether the incoming information is a join or a new label. If it is a new label, the information is written to the RAM and the next state will be decided by the incoming data check. Additionally, the finished flag of this label will be set to false. If it is a real join, the address lines of the RAM are set to the labels that

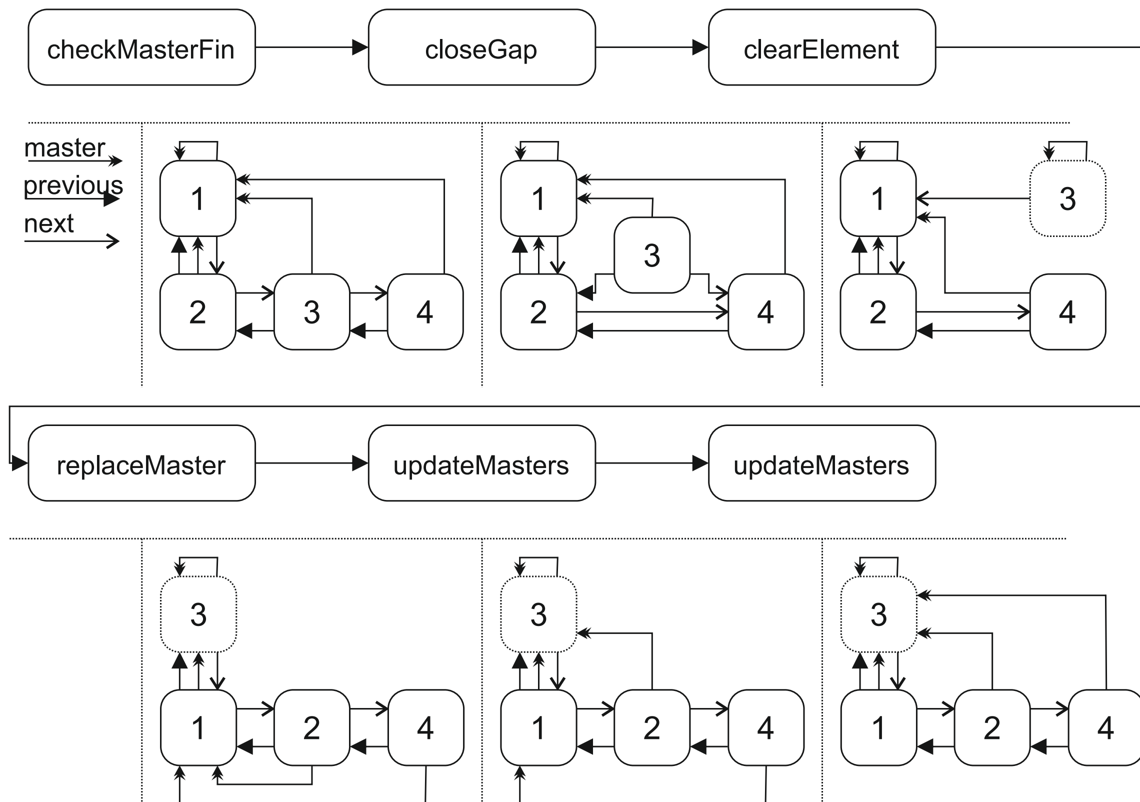


Figure 3.13: Switching of masters in a larger subset. For the finished element 3, the first steps are analogous to figure 3.12. After the *checkMasterFin* state, the finished flag of the subset's master is known. The *closeGap* state alters the pointers of element 2 and 4, removing element 3 from the subset. The *clearElement* state updates all pointers of element 3 and sets its finished flag. The *replaceMaster* state replaces all pointers of element 1, making it the first child of the new master element. In the next two steps, the *updateMasters* state updates the master pointers of the remaining elements in the subset.

should be joined, to receive their next, previous and master pointers for the next state. The next state will be the *checkMasters* state.

If the *checkMasters* state is entered, the output lines of the dual ported RAM contain the pointer information of the two labels that need to be joined. The state compares the label IDs of both masters with each other. If they are equal, the labels belong to the same subset and were joined earlier. If the masters are equal, the next state will be decided by the incoming data check. If the masters are different, it is necessary to check if any of the masters is finished already. To do this, both master ids are assigned to the address lines of the finished flag RAM. The finished flags of both masters will be available in the next state, which is the *checkMasterFlags* state. One special case may occur when both labels are masters and have no next pointer, being the masters of two subsets consisting of only one element. In this case, there is no need to read the finished flags, as subsets consisting only of a finished element are not possible. Furthermore, it is the most simple case of a union. The pointers of those two elements can be updated in one cycle, each on one channel of the disjoint-set RAM. If this case occurs, the next pointers of both masters are zero. This can be easily checked. In this special case, the union of both subsets is finished and the next state will be decided by the incoming data check. This special case is also shown in figure 3.9.

Based on the finished flags of both masters, the *checkMasterFlags* state decides what to do next. If both masters are finished, the next state will be the *joinFinMasters* state. Otherwise, the next state will be the *correctMaster* state. Regardless of the next state, the address lines of the disjoint-set RAM will be set to the ids of the master. However, which id will be assigned to which line differs depending on the finished flags of the masters. If only one master is finished, its id will be assigned to the first address line. If none or both are finished, the master ids will be assigned based on the lines that their members were assigned to in the previous steps. The element that will be the new master of the united subsets (referred to as master A of subset A) is on the first line of the RAM. There, the master will already be updated, if the next state is *correctMaster*. Its next pointer is changed to the other master. The master that will be joined into the other subset is on the second line (referred to as master B and subset B).

The *joinFinMasters* state is only entered when two subsets with finished masters were detected by the *checkMasterFlags* state. This implies that both subsets consist of more than one element, because a subset consisting of only a finished master would have been released. In this state, the *bufferedChild* register is set to the first sibling of master A (the next pointer of master A). On the first port of the disjoint-set RAM, the next pointer of master A is set to the first sibling of master B. The master B label is released after this state. The second address line of the disjoint-set RAM is set to the next pointer of master B. Thus, in the

next cycle the information about this element will be present at the output of the RAM. This data is needed by the next state, *updateMasters*, to update the master pointers of subset B.

The *correctMasters* state is entered when one or none of the masters is finished. In this state, the two different situations are handled differently. The first situation is when the subset B has several elements. In this case, the siblings of the master B must be retrieved and their master pointers must be adjusted to master A. For this case, the first port of the RAM is used to write the new pointers of master B. The next pointer will be kept, while the previous and master pointer are changed to master A. On the second line of the RAM, the next element of subset B will be retrieved. The next state for this situation is the *updateMasters* state. The id of master A is stored in the *newMaster* register, while the id of the next sibling of master A will be stored in the register *bufferedChild*.
The second situation that can occur is that the subsection B only consists of one element, while the subsection A consists of more than one element. In this case, correcting the pointers of subset B is finished after this state. On the first port or the disjoint-set RAM the pointers of master B is updated: the previous pointer and the master pointer are set to master A while the next pointer is set to the value of the next pointer of master A. The next state for this situation will be the *correctLastChild* state, where the current sibling of master A is corrected. This state needs the information about the next sibling of master A on the second RAM port. Fir this reason, the second address line is set to the next pointer of master A.

The *updateMasters* state updates the master pointers of a subset to reflect its union with a different subset. When this state is entered, there must already be a valid element at the second output line of the disjoint-set RAM, which is referred to as current element. The master of the current element is updated to the element id stored in the *newMaster* register by a previous state. For this, the first RAM port is used. If the previous state was the *joinFinMasters* state, the previous pointer will be set to the *newMaster* register as well. This needs to be done because its previous element, which is also its old master, is removed from the disjoint-set. Figure 3.11 shows this special case. If the current element's next pointer is valid ($neq0$), the next element id is assigned to the second address line of the disjoint-set RAM. In this case, the state is not changed. The *updateMasters* state is only left if the next pointer of the current element is not valid ($= 0$). In this case, the last corrected master is written on the first RAM port and the id of the stored *bufferedChild* will be assigned to the second address line of the RAM. The information about the *bufferedChild* is needed by the next state, the *correctLastChild* state. If the *bufferedChild* pointer is invalid ($= 0$), there is no detached child and the union process is finished. In this case, the next state will be decided by the incoming data check.

When the *correctLastChild* state is entered, the first steps of a union have already been performed: all pointers of the subset B have been changed, as well as the pointers of master A. In this state, the detached elements of subset A will be attached to the end of subset B. For this, the previous pointer of the former first sibling of master A needs to be changed to the last element of subset B. If this state is entered, the information about the former first sibling of master A is valid on the second output line of the disjoint-set RAM. This is necessary because the next pointer of that element must be known before writing the information back to the RAM. The previous pointer will be set to the last element of subset B, which is still present at the first port of the disjoint-set RAM. After this operation, the union of two subsets is finished and the next state is decided by the incoming data check.

The states described above are used to unite subsections of the disjoint set in different ways. Several figures show the different union types and the involved states. Figure 3.9 shows the union of two subsections consisting of only one element, figure 3.10 shows the union of two subsections with more than one element and figure 3.11 shows the union of two subsections with finished masters. While the right side of the state-machine in figure 3.8 is responsible for uniting subsections in different ways, the left part of the state-machine handles the shrinking and changing of subsections to reflect discontinued labels.

When no joining information is available, but information about a discontinued label is, the *getFinished* state is entered. The id of the finished element is assigned to the first address line of the RAM. In the next state, the *checkFinished* state the information about this element is available at the RAM output. Three different situations are handled by this state. First, if the finished element is a subset of only one element, the subset is finished and cannot grow anymore, therefore the subset is removed. The second situation is that the finished element is a master of a subset containing more than one element. In this case, the master is flagged as finished in the corresponding RAM. In both cases, the handling is finished and the next state will be decided by the incoming data check. The third situation is that the finished element is not a master and therefore belongs to a subset with more than one element. In this case, the id of the element's master is assigned to the finished flag RAM and the next state will be the *checkMasterFin* state where the finished flag of the subsection's master is stored in a register. The previous and next pointers are assigned to the address lines of the disjoint-set RAM. The next state will be the *closeGap* state.

Independent of its finished state, the discontinued element needs be removed from the linked list. This is done by the *closeGap* state. If the discontinued element is the only child of a finished master, the master will be removed as well. This is the case if the previous pointer of the discontinued element points to the master and the next pointer is invalid ($= 0$). In this case, the pointers do not need to be

updated and the next state will be decided by the incoming data check. In all other cases, the element needs to be removed from the linked list. The information about the previous and the next element of the discontinued element are available at the RAM output. To remove the element from the list, the pointers of those elements need to be changed. The element will always have a previous element, as it is not the master. The next pointer of the previous element will be updated to the next pointer of the discontinued element. If the discontinued element was not the last element in the list, it has a valid next pointer. In this case, the previous pointer of the next element will be updated to point to the previous element. If the master is already finished, the process of removing the element is finished and the next state will be decided by the incoming data check. If the master is not finished, the discontinued element will be reinserted into the subset as the new master. In this case, the next state is the *clearElement* state.

The *clearElement* state updates the removed element to be a master on one channel of the RAM. Its master pointer is set to itself, its previous pointer to 0 and its next pointer to the current master of the subset. Also, it is flagged as finished in the corresponding RAM. On the second channel, the current master id is assigned. to read its pointers in the next state. The next state is the *replaceMaster* state. In this state, the pointers of the current master are changed at the first line of the RAM. It will become the first child of the formerly removed element, which will be the new master. For this, the previous and master pointers are updated to point to the new master. If there are no other elements in this subset (the subset consists of only two elements), the process of master switching is finished and the next state will be decided by the incoming data check. If more elements are in the subset, their master pointers need to be updated. To do so, the content of the former master's next pointer is assigned to the second address line of the RAM. As the state of the subset is equal to the state of a subset that was united, the next steps will be handled by the *updateMasters* state, which is also used during the union of subsections. The master switching is described in figure 3.12 for a subset with two elements and in figure 3.13 for a subset with more elements.

3.4 Architecture

Figure 3.14 shows the block-diagram of the novel algorithm. Each of the modules perform a different task in the overall algorithm. The different modules and the connected queues and RAMs are explained in detail in the following paragraphs. The input to the whole system is of the image data and configuration inputs. The image data consist of the basic signals described in section 2.2, namely frame valid, line valid, data valid and the pixel data. Unlike all other systems reported, the novel system is designed to handle three pixels in one clock cycle. The configuration

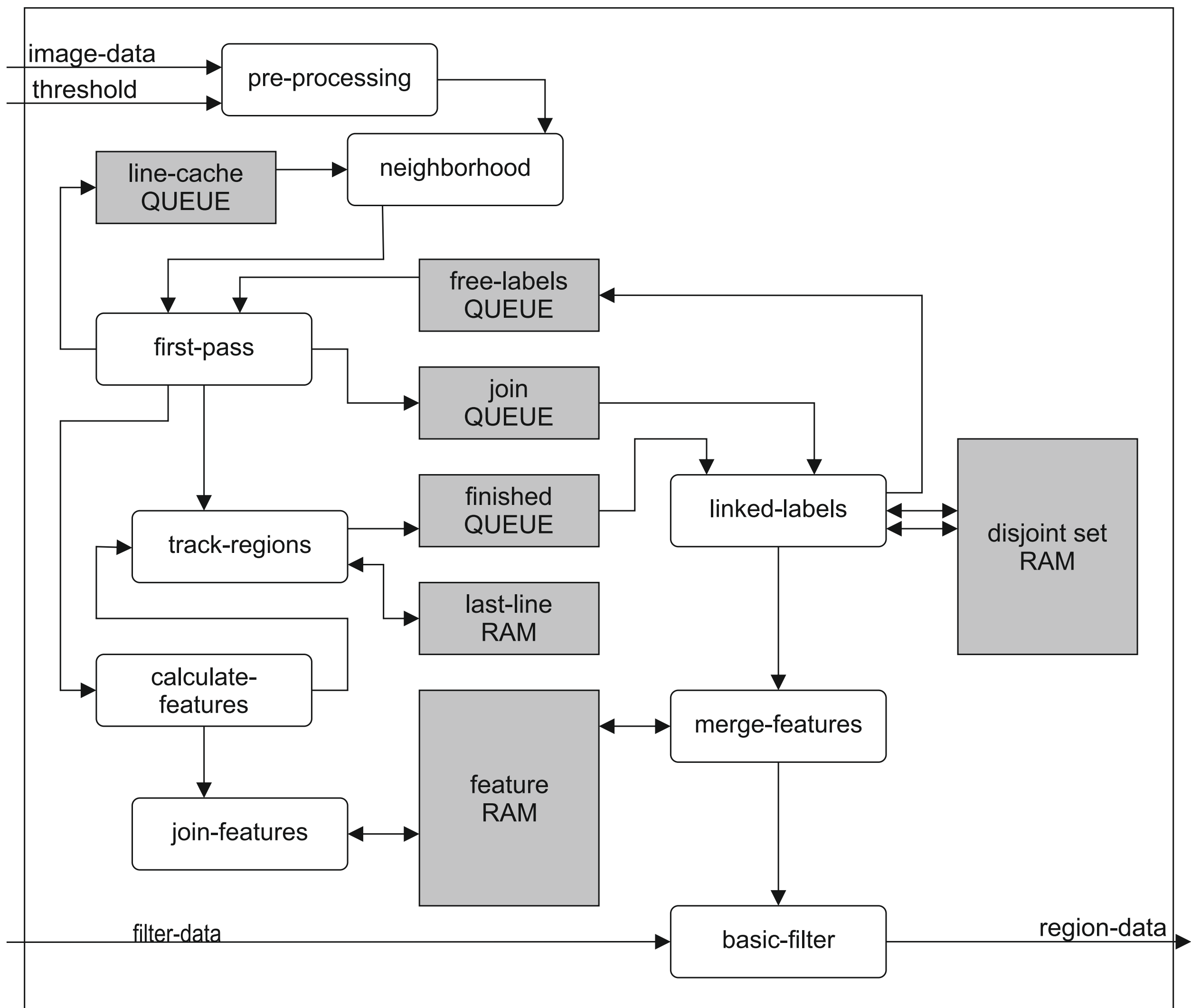


Figure 3.14: Architecture of the parallel approach algorithm. Each of the modules perform a different task in the overall algorithm. The input to the whole system is of the image data and configuration inputs. The left part of the system performs connected component labeling without considering the equivalences; information about equivalent and finished labels are stored to queues. The right part of the system contains modules to perform the equivalence handling and the merging of feature data.

inputs consist of the threshold and filter information. The filter can be used to omit regions that do not fulfill the filter bounds. Currently, the region size, based on the number of pixels, can be filtered with a minimum and a maximum bound. The output of the system is the information about found regions. The information consists of the feature-data (see section 2.3.2) before the post-processing step. The information is accompanied by a valid flag. The information stays valid for one clock cycle.

First-pass modules

The pre-processing module alters the image data stream to prepare it for the actual connected-component labeling. It has three tasks. Firstly, it applies the threshold to the image data. Pixel values that are above the threshold are kept, values below the threshold are set to zero (see equation 2.1).

Secondly, the module assigns a valid flag to each pixel. This valid flag indicates whether the pixel belongs to the foreground and therefore to a component, or to the background. The following modules can use this flag for their decision, resulting in faster processing due to less computations.

The third task of the pre-processing module is to apply a specialized version of the morphological closing operation (Serra, 1982). The closing operation removes small holes in an image, in this case one pixel background between two foreground pixels. The closing performed here is adapted to the special needs of this algorithm. The closing is performed with a mask of 2×1, thus removing holes of one pixel in the x direction. Additionally, the pixel value is not changed; only the valid flag is changed. The effect is that, if a pixel is changed by the closing operation, it is counted as part of the component, but its value is not taken into account during the calculation of the component's features. The closing operation is necessary to be able to calculate the component membership of three pixels in one cycle.

The neighborhood module is coupled with the line cache queue. The line cache queue contains the assigned labels of the previous line. It is designed as queue/FIFO. Based on the actual image control flags the neighborhood module decides if data needs to be pulled from the queue. The neighborhood then passes the image data together with the information about the previously assigned regions to the next module. This information consists of the five label IDs: the three label IDs that were assigned to the pixels above the current three pixels and the label IDs left and right of those labels (see figure 3.15).

The main task of the first pass module is to decide which label ID is assigned to any of the foreground pixels of the current pixel data. Here, the closing operation of pre-processing is used as a precondition: It is not possible for a one-pixel gap to be between two valid pixels. Therefore, all of the valid pixels of the current image

tl	t_1	t_2	t_3	tr
pl	p_1	p_2	p_3	

Figure 3.15: First-pass neighborhood of the parallel approach. The pixels p_1, p_2 and p_3 need labels assigned. The left pixel *pl* and the five pixels above p_1, p_2 and p_3 are checked: the top-left pixel *tl*, the pixels directly above (t_1, t_2, t_3) and the top-right pixel *tr*.

data belong to the same region. The decision process is as follows: If there is no current foreground pixel, no region is assigned. If the leftmost of the current pixels (p_1) is a foreground pixel and the pixel left of this (pl) has a valid label, this label is assigned. If the pixel left of the current one was a background pixel and therefore has no valid label, the labels of the five pixels above the current ones are checked. Here, the leftmost valid label that touches a foreground pixel is used. If none of the surrounding pixels were foreground pixels, a new label ID is assigned to the pixels. The new label is taken from the *free labels* queue. The assigned region id is pushed into the line cache queue for the foreground pixels. For background pixels a zero, indicating no label, is assigned. figure 3.16 shows different situations where a previous region was assigned to the valid pixels. Figure 3.17 shows different situations where a new label ID was assigned to the valid pixels. Additionally, the modules checks whether two previous distinct labels need to be joined. That is the case if the valid foreground pixels connect two different previously assigned labels. This can be the case if the region left of the current window is distinct from one of the regions above the current window. It is also possible that there are two different labels in the five pixels above the current window figure 3.18 shows the algorithm that calculates if and which regions have to be joined.

However, it is not possible that three distinct regions need to be joined. Because of the closing process in the pre-processing module it is not possible for a one pixel gap to be between two valid regions. Therefore, it is not possible for three different label IDs to be in the top row of five pixels. Also, if there are two different labels in the top row, one of the labels must be assigned to a pixel touching *pl*, either *tl* or t_1. If this is the case, the label of *pl* is either equal to the label of *tl* or t_1, or, if this its not the case, the labels were scheduled for joining in the previous clock cycle. The different possible situations that lead to a join operation are shown in figure 3.18. If a join needs to be performed, the information about the two regions are pushed into the join queue. The region information is forwarded into two other modules: the assigned label ID, with the information whether this label is a newly assigned one, and the image data flags are forwarded into the track-regions module. This module keeps track of the region occurrences in respect to their position in

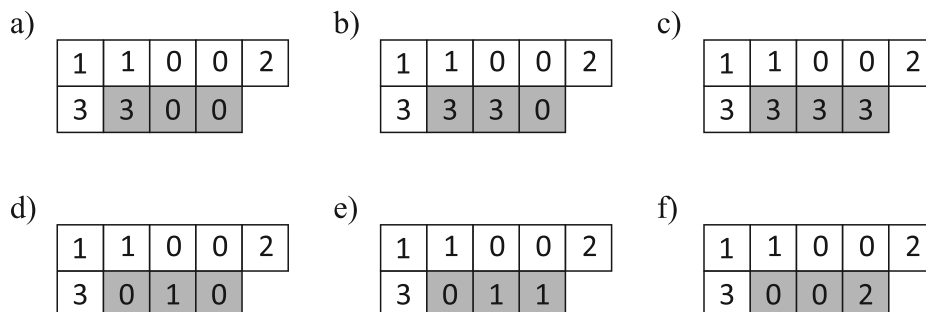


Figure 3.16: Label assignments during the first pass. If *pl* is valid and p_1 is a foreground pixel, the label of *pl* is used (a and b, see Fig. 3.15). Otherwise, a label from the line above is used. Here, the leftmost label that touches a foreground pixel is used. The areas with a gray background show the label IDs that are assigned. A zero in the assignment represents a background pixel.

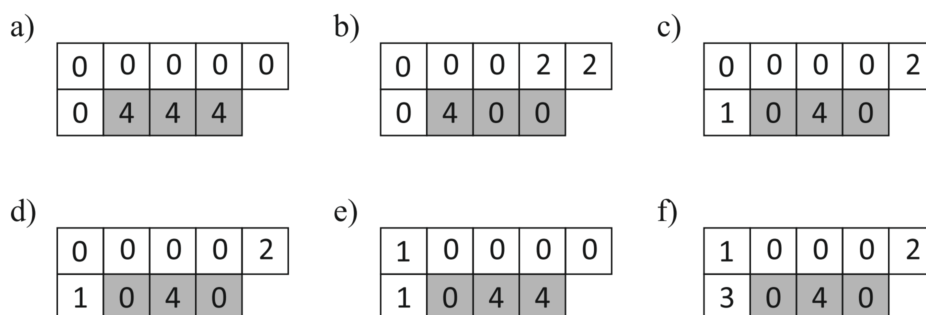


Figure 3.17: New label assignments during the first pass. A new label is assigned if none of the surrounding pixels has a valid label assigned (a), or if none of the surrounding labels touches a foreground pixel.

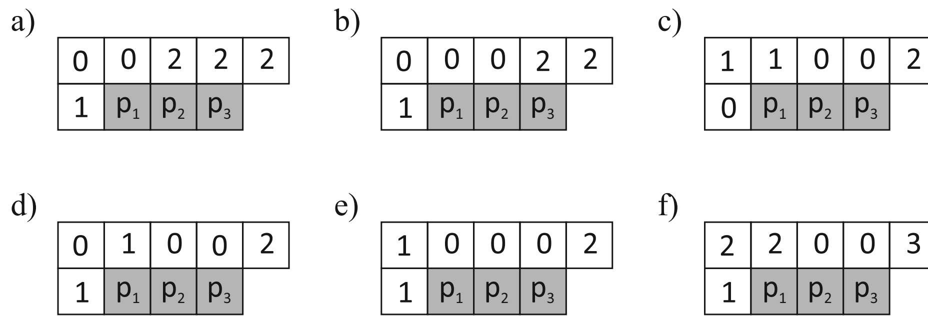


Figure 3.18: This figure shows the different situations where two previously distinct labels need to be joined. in a-e, the labels 1 and 2 must be joined if the connecting pixels are valid: a) p_1 must be valid. b) p_1 and p_2 must be valid. c,d) p_2 and p_3 must be valid. e) p_1, p_2 and p_3 must be valid. f) is a special case. Here, the labels 2 and 3 must be joined if p_2 and p_3 are valid. The labels 1 and 2 have been joined in the previous cycle. In the previous cycle, the current pl was p_3, tl was t_3 and t_1 was tr. a and b are possibilities of the previous cycle.

the image. The assigned region id together with the image data is also forwarded into the calculate features modules.

Feature handling modules

The calculate features modules derive the basic parts of the features (see section 2.3.2) from the image data. Based on the image data flags frame-valid, line-valid and data-valid, the current x and y position of the pixels are calculated. The basic parts for each feature are then calculated in parallel independently for each pixel. In parallel, the changes of the closing operation in the pre-processing module are reverted. Therefore, pixels with no pixel value are not any longer considered foreground pixels, even if they connect two valid foreground pixels. With this behavior, the closing operation has no influence on the calculated features of a region but only on the labeling. In a second step, the independent values of the pixels are combined, based on the valid flags of the pixel: only valid pixels are combined. These steps are pipelined; while the second step is performed for three pixels, the first step is already performed for the next three pixels in the same cycle. The results of this module are the basic features for the three processed pixels. Examples of these features are $\min(x)$, $\max(x)$ or the number of valid pixels. These results are transferred to the join features module, together with the label ID that belongs to these values. Additionally, the current y value is forwarded into the track-regions module, where it is used to keep track of the label.

The basic features of the processed pixels are combined with previously collected data in the join-features module. The join-features module controls one side of a dual-ported RAM that stores the basic feature data for each label. The inputs to this module are the basic features of the pixels and the label ID for these pixels and a flag indicating whether the label is a newly assigned label. If the label is newly assigned, the pixel data is stored into the RAM. Any previous data is overwritten. This is necessary due to label reuse. The RAM may already contain outdated data. If the region is not new, the module reads the previously collected data from the RAM in a first step. In a second step, the values retrieved from the RAM are combined with the data of the current pixels. In a third step, the updated feature data of the label is written back to the RAM. These three steps are pipelined. This is necessary due to the one cycle delay of a RAM read operation. However, this induces a problem if the same label is updated continuously, as the data retrieved from the RAM is not yet correct, due to the two cycle delay. Therefore, the results of the second step are buffered. When the second step is performed, it is checked which label is in the buffer. If the label is equal to the current one, the buffer is used instead of the RAM output.

The above described modules perform simple connected component labeling without addressing the merge of two labels. The merge is detected but not handled in this part of the system. All of the described modules operate in stream mode: They are capable of handling a new set of pixels in each clock cycle and have no need of pausing the process or buffering the image data. The feature data is calculated regardless of the region merge. The idea behind this whole system is that the merging of labels is done in a different part of the system at a time when the labels are not used anymore.

The merge-features module controls the second port of the feature-RAM. Its main purpose is to read the basic features of two distinct labels, combine them and write the result back to one of those labels. It is not allowed to read and write the data of labels that can still grow, because those labels can still be updated by the join-features module. Therefore, only data of discontinued labels may be used by this module. This requirement is not handled by this module, but by the equivalence handling in the linked-labels module. The merge-features module has a simple control interface to the linked-labels module. It features an input line for a label ID and several control flags, namely *storeToA*, *storeToB*, *mergeBuffers* and *sendDataOut*. The module features two buffers called buffer A and buffer B. Its control interface features the flags *storeToA* and *storeToB*. If one of these flags is set to 1, the basic feature values of the assigned label ID is stored in buffer A or buffer B, respectively. The buffers cannot be both assigned in the same clock cycle. The data stored in both buffers will be merged in the next clock cycle. Assigning the *mergeBuffers* flag will store the merged data of the buffers to the feature RAM at the position of the label ID currently assigned to the input line.

With this simple interface, the linked-labels module can control the merging of feature data in a simple manner. If the *sendDataOut* flag is assigned, the feature data of the label ID assigned to the input line will be sent out in the next cycle. Thus, the linked-labels module can instruct this module to output the data of a fully processed region. The data is always assigned to the output port of this module, only the *valid* output flag is controlled by the *sendDataOut* input flag.

The basic-filter module filters the outgoing finished regions by checking basic features of the regions. This is used to remove very small, very large or unwanted regions from the results. In this module, the regions can only be filtered by basic features that are available before the post-processing step. By using this filter, less post processing steps have to be performed and more time is available for classification. The filter forwards the feature data from the merge-features module, but alters the *valid* flag based on the filter result. The basic filter can filter out regions based on their number of pixels and their x and y coordinates. For the number of pixels, an upper bound and a lower bound can be used. For x and y, a region of interest (ROI) can be assigned. Only if the finished component is partly inside this ROI, the filter will output the component.

Equivalence handling modules

The task of the track-regions module is to check whether a label is no longer in use. This is the case if a label has not been assigned in the last line. If it has not been assigned, it cannot appear in the neighborhood for the current line and therefore cannot be assigned anymore. The module uses the small last-line RAM that stores the last y position of each label (see figure 3.14). The size of the RAM depends on the maximum number of labels. If a label ID is forwarded from the first pass, the y value of this label is updated in the last line RAM. The current y value is forwarded from the calculate features module into this module. Additionally to the RAM that maps label IDs to the y position, internally the module uses a small round robin queue that contains all labels that are currently in use. If a newly assigned label is reported from the first pass module, this label is pushed into the queue. Otherwise, the next label is taken from the queue and its last occurrence is checked against the current y position. If the last occurrence was more than one line before the current one, this label is considered finished and pushed into the finished queue. If it is not finished, it is pushed back into the round-robin queue. The results of this module are pushed into the finished queue, from where the information is later used by the linked labels module.

The linked-labels module is the most crucial and complex part of the system. It controls the disjoint-set RAM where the information about equivalences are tracked (see section 3.3). The state-machine deployed in this module is also described

in section 3.3 and is shown in figure 3.8. Here, the signals and connections to the other modules will be described. The module reads the empty flags of the join-queue and the finished-queue. These are evaluated during the incoming data check. If the join-queue is not empty, the pop flag of this queue will be assigned and the checkJoin state will be entered, starting the handling of label joining. If the join-queue is empty but the finished-queue is not, the pop flag of this queue will be assigned and the next state will be the getFinished state, starting the handling of discontinued labels. If both queues are empty, the idle state will be entered.

When labels are removed from a subset, they are pushed into the free-labels queue by this module. This is the case when the *checkMasterFlags* state is entered and both masters are finished (see figure 3.11). The master that is removed from the subset will be pushed into the free-labels queue. This is also the case when a subset consisting of only one region is encountered in the *checkFinished* state and during the *checkMasterFin* state, if the finished element's master is also finished. In both cases, the element that was read from the finished queue will be pushed into the free-labels queue.

The module needs to control the merging of feature data. The state machine is designed in such a way that the data of discontinued labels is always merged to the master of the subset. As it is not allowed to merge data of a label that is still in use, the master of a subset must be finished before feature data of a different label can be merged. Therefore, if the master is not finished the master of the subset is exchanged with the finished label, and the finished label is not yet removed from the subset (see section 3.3 for more details). Therefore, if a merge is executed it is always executed between two finished labels. The merge is conducted by the merge-features module and is controlled by forwarding a label ID, as well as with the flags *storeToA*, *storeToB* and *mergeBuffers*.

Label merging can occur in various states of the state-machine. When the *check-MasterFlags* state is entered and both masters are finished (see figure 3.11), one master will be removed from the subset and its feature data must be merged to the master that is kept. Therefore, the id of the removed label is forwarded to the merge-features module, together with the command to store it in buffer A (*storeToA*). In the next state, the *joinFinMasters* state, the *storeToB* flag is set together with the id of the finished master that is kept. In the following state (*updateMasters*) state, the *mergeBuffers* flag will be assigned, again together with the finished master that is kept (however, this is only done if the previous state was the *joinFinMasters* state). After this state, the feature data of those two labels are merged and stored at the address of the master of the subset. The same sequence of commands to the merge-features module is executed when a discontinued label is removed by the states of the left part of the state-machine shown in figure 3.8. Here, the *storeToA* flag and the ID of the finished label is assigned during the

getFinished state. The id of the master and the *storeToB* flag is assigned during the *checkMasterFin* state. During the *closeGap* state, the master ID and the *mergeBuffers* flag are assigned. After this state, the subset has been shrunk and the feature data of the discontinued label has been merged with the master of the subset.

If the last element of a subset is removed, the *sendDataOut* flag together with the last label of the subset is assigned to the merge-features module. This can occur at two states. First, if the only element of a subset is finished, it will be encountered during the *checkFinished* state. In this case, the subset was never united or shrank. The label ID together with the *sendDataOut* flag is forwarded to the next module. Second, it can occur if the last child of a finished master is removed from it. Then, the *sendDataOut* flag will be assigned at the same time as the *mergeBuffers* flag. This is possible, as the feature RAM is a write first mode RAM. Such a RAM outputs the write data instead of the RAM cell content if data is written. Therefore, the merged content feature data of the last child and the finished master is sent to the basic-filter module.

The output of the hardware systems is a collection of basic features together with a *valid* flag. The basic features are the subparts of features before the basic post-processing step. The working principle of the hardware-system is independent of the used features. New features can be added easily by extending basic data types and the modules calculate-features, join-features and merge-features. Additionally, the size of the feature RAM has to be updated.

3.4.1 Parameters and RAM sizes

The hardware system can be parametrized before it is synthesized by the FPGA-software. The parameters are the maximum image width and height, the pixel depth as well as the number of label IDs. The maximum image width and height are used to calculate the width and depth of various RAMs in the system. Additionally, it is possible to enable or disable the calculation of different features in order to save RAM space. The RAM depth refers to the number of elements that can be stored in a RAM, while the RAM width refers to the size of elements that can be stored. For all RAMs in the system, the depth, the width or both are dependent on the parameters. For example, the line-cache queue needs to hold a label ID for each column of the image. Therefore, its depth must be equal to the maximum image width while its width must be large enough to hold each possible label ID. Table 3.2 gives an overview of all RAM sizes depending on the parameters maximum width, height and number of label IDs. The element size is given in bits that are needed to store the information. For a natural number n, the number of needed bits is the next natural number of $\log_2(n+1)$. This can be written as

$\lceil \log_2(n+1) \rceil$. The +1 is needed, because the number 0 is included in the size. For example, an eight bit register (2^8) can hold 256 different combinations, it can store the numbers 0 to 255. Therefore, to store the number 256, a nine bit register is needed, but $\log_2(256) = 8$.

RAM/queue name	number of elements	element size
line-cache queue	width_{max}	$\lceil \log_2(\text{labels}_{max}) \rceil$
free-labels queue	labels_{max}	$\lceil \log_2(\text{labels}_{max}) \rceil$
join queue	fixed	$2 \cdot \lceil \log_2(\text{labels}_{max}) \rceil$
finished queue	fixed	$\lceil \log_2(\text{labels}_{max}) \rceil$
last-line RAM	labels_{max}	$\lceil \log_2(\text{height}_{max}) \rceil$
disjoint-set RAM	labels_{max}	$3 \cdot \lceil \log_2(\text{labels}_{max}) \rceil$
feature RAM	labels_{max}	dependent of features

Table 3.2: The number of elements and the element sizes for the different RAMs and queues in the system. The sizes depend on the maximum width, height and number of used labels.

The free-labels buffer contains all possible labels if the system is inactive. Therefore, its size must be equal to the maximum possible number of label IDs and its width must be large enough to hold the maximum label ID. In this system, the label IDs are natural numbers starting at 1 and counting up. Therefore, the maximum number of labels is equal to the maximum label ID. The join queue must be able to hold two labels that have to be joined, while the finished queue needs to hold the finished label. Both queues have a fixed size. The size will be discussed in section 4. The last-line RAM contains the last y position for each label. Therefore, its depth has to be the number of labels while its width has to be large enough to hold each possible y value. The disjoint-set RAM contains the linked list of the disjoint-set of currently used labels. The linked list elements are stored at the address of the label ID and contain pointers to other labels, namely the next, the previous and the master element of the subset (see section 3.3 for more details). Therefore, its depth is equal to the number of elements while its width has to be large enough to hold three other label IDs.

The feature RAM contains the basic feature data for each label. Therefore, its depth is equal to the maximum number of labels. The width of the feature RAM is dependent on the activated features as well as on the needed bits for each feature. The needed bits differ for each feature and are discussed below. The overall width of the feature RAM is the sum of all needed bits of all activated features. For

example, the maximum number of pixels in a 512×512 image is 2^{18} (if there is one region that fills the whole image); 18 bits are needed to store the number of pixels. For a 2048×2048 image the maximum number of pixels is 2^{22}. Therefore, 22 bits are needed to store the number of pixels.

For the number of pixels, the following equation (equation 3.11) can be used to determine the required bits (mw = maximum width, mh = maximum height):

$$bits_{numPix} = \lceil \log_2 ((mw \cdot mh) + 1) \rceil \tag{3.11}$$

For the minimum and maximum values of x, the highest possible number is the last pixel of each line pixel. The number of bits needed for the bounding rectangle are shown in equation 3.12 and equation 3.13. The algorithm starts the counting of the x and y position with 0 instead of 1, therefore the maximum width -1 is needed.

$$bits_{minX} = bits_{maxX} = \lceil \log_2(mw) \rceil \tag{3.12}$$

$$bits_{minY} = bits_{maxY} = \lceil \log_2(mh) \rceil \tag{3.13}$$

While the number of bits for the bounding rectangle and the number of pixels are directly derived from the maximum width and height, other features have to be considered more carefully.

The center of gravity is determined by calculating the maximum sum of all x and y values. In the worst case, if the whole image is a single foreground region, the highest possible sum consist of all elements. In each line, all x values from 0 to the maximum width-1 are summed together, this is possible for every line. Therefore, the needed bits for the sum of x and y can be calculated as shown in equation 3.14 and equation 3.15.

$$\text{sumX}_{max} = mh \cdot \sum_{x=0}^{mw-1} (x)$$

$$\text{bits}_{sumX} = \left\lceil \log_2 \left(\left(mh \cdot \sum_{x=0}^{mw-1} (x) \right) + 1 \right) \right\rceil \tag{3.14}$$

$$\text{bits}_{sumY} = \left\lceil \log_2 \left(\left(mw \cdot \sum_{y=0}^{mh-1} (y) \right) + 1 \right) \right\rceil \tag{3.15}$$

For the principal component analysis, the different elements for the calculation of the covariance matrices must be stored. In addition to the sums of all x and y (see equation 3.14 and equation 3.15), the sums of $x \cdot y$, x^2 and y^2 must be stored. As before, the worst case occurs when all pixels in the image belong to one region. For x^2 and y^2, the equations can be created in analogy with equation 3.14 and equation 3.15, respectively (see equation 3.16 and equation 3.17). For the maximum value of $x \cdot y$, for each line the sum off all x values must be multiplied with the current y value. This is shown in equation 3.18.

$$\text{bits}_{sumXSquare} = \left\lceil \log_2 \left(\left(mh \cdot \sum_{x=0}^{mw-1} (x^2) \right) + 1 \right) \right\rceil \tag{3.16}$$

$$\text{bits}_{sumYSquare} = \left\lceil \log_2 \left(\left(mw \cdot \sum_{y=0}^{mh-1} (y^2) \right) + 1 \right) \right\rceil \tag{3.17}$$

$$\text{bits}_{sumXY} = \left\lceil \log_2 \left(\left(\sum_{x=0}^{mw-1} \sum_{y=0}^{mh-1} (x \cdot y) \right) + 1 \right) \right\rceil \tag{3.18}$$

For the weighted center of gravity (see equation 3.1), the pixel weight is multiplied with the current x value or y value, respectively. Additionally, the average weight of all pixels must be stored. The equations for the bits' sizes of the weighted center of gravity are shown in equation 3.19, 3.20 and 3.21, where v_{max} is the maximum pixel value. For a default eight bit gray-scale image, $v_{max} equals 255$.

$$\text{bits}_{sumX} = \left\lceil \log_2 \left(\left(mh \cdot \sum_{x=0}^{mw-1} (x \cdot v_{max}) \right) + 1 \right) \right\rceil \tag{3.19}$$

$$\text{bits}_{sumY} = \left\lceil \log_2 \left(\left(mw \cdot \sum_{y=0}^{mh-1} (y \cdot v_{max}) \right) + 1 \right) \right\rceil \tag{3.20}$$

$$\text{bits}_{sumV} = \lceil \log_2 ((mw \cdot mh \cdot v_{max}) + 1) \rceil \tag{3.21}$$

For the bounding octagon, a rotated rectangle has to be stored additionally to the bounding rectangle. For a rectangle rotated by 45°, the maximum and minimum values of $x+y$ and $x-y$ must be stored. This is the only feature with the possibility of negative values. Equation 3.22 gives the number of bits for $x+y$, equation 3.23 gives the number of bits for $x-y$. Here, an extra bit must be added to store the sign of the number.

$$\text{bits}_{x+y} = \lceil \log_2 (mh + mw - 1) \rceil \tag{3.22}$$
$$\text{bits}_{x-y} = \lceil \log_2 (\max(mh, mw)) + 1 \rceil \tag{3.23}$$

If the features number of pixels, bounding box, center of gravity and PCA are enabled for an image with 1024×1024 pixels, the following bits are needed to store the basic feature data:

$$\begin{aligned}
bits_{numPix} &= \left\lceil \log_2(1024 \cdot 1024 + 1) \right\rceil = 21 \\
bits_{minX} = bits_{maxX} &= \left\lceil \log_2(1024) \right\rceil = 10 \\
bits_{minY} = bits_{maxY} &= \left\lceil \log_2(1024) \right\rceil = 10 \\
\text{bits}_{sumX} &= \left\lceil \log_2 \left(\left(1024 \cdot \sum_{x=0}^{1023} (x) \right) + 1 \right) \right\rceil = 29 \\
\text{bits}_{sumY} &= \left\lceil \log_2 \left(\left(1024 \cdot \sum_{y=0}^{1023} (y) \right) + 1 \right) \right\rceil = 29 \\
\text{bits}_{sumXSquare} &= \left\lceil \log_2 \left(\left(1024 \cdot \sum_{x=0}^{1023} (x^2) \right) + 1 \right) \right\rceil = 39 \\
\text{bits}_{sumYSquare} &= \left\lceil \log_2 \left(\left(1024 \cdot \sum_{y=0}^{1023} (y^2) \right) + 1 \right) \right\rceil = 39 \\
\text{bits}_{sumXY} &= \left\lceil \log_2 \left(\left(\sum_{x=0}^{1023} \sum_{y=0}^{1023} (x \cdot y) \right) + 1 \right) \right\rceil = 38 \\
\text{bits}_{total} &= 21 + 2 \cdot 10 + 2 \cdot 10 + 29 + 29 + 39 + 39 + 38 \\
\text{bits}_{total} &= 235
\end{aligned} \tag{3.24}$$

The calculations show that the needed RAM size highly depends on the used features as well as the needed image size. Section 3.6.3 gives an overview of the utilization for different FPGAs.

3.5 Further throughput improvement

Further speed improvement can be achieved by handling more than three pixels in one clock cycle. The following paragraph explains the steps that are needed to upgrade the system for the handling of nine concurrent pixels.

The above described system is designed to handle three pixels at each clock cycle. One of the constraints for this system is that any three pixels that are adjoined horizontally belong to the same component, even if there is a hole between two pixels. This is ensured by the specialized 2×1 closing operating in the preprocessing module. If a similar constraint is formulated for any three pixels, horizontally, vertically or diagonally, nine pixels that are in a block of 3×3 belong to the same component. This can easily be ensured by changing the closing operation in the preprocessing module to this constraint. By doing so, the system can operate on nine pixels in one clock cycle by changing only two other submodules. The first pass step needs to be updated, as more possible combinations for label assignment can occur. If operating on a 3×3 block, two more options for the last region need to be taken into account (see figure 3.19).

tl	t_1	t_2	t_3	tr
l_1	p_{11}	p_{21}	p_{31}	
l_2	p_{12}	p_{22}	p_{32}	
l_3	p_{13}	p_{23}	p_{33}	

Figure 3.19: Advanced first pass neighborhood for nine pixels. The pixel p_{11} to p_{33} need a label assigned. The three left pixels l_1, l_2, l_3 and the five pixels above p_{11}, p_{21} and p_{31} are checked: The top-left pixel tl, the pixels directly above (t_1, t_2, t_3) and the top-right pixel tr.

The last module that needs to be adapted in the calculateFeatures module. This module summarizes the features of all pixels in the block before the result is forwarded to the feature-RAM. Currently, the system has two pipelined steps. First, the basic features for each of the three pixels are calculated independently. In the second pipeline step, the basic features of all foreground pixels are combined. This module is adjusted to handle nine concurrent pixels. Adjusting the first pipeline step resulted in additional hardware, as now more multiple calculations need to take place. Especially the needed multipliers have increased. Adjusting the second pipeline step results in additional hardware as well as a longer calculation path. It is necessary to summarize the features of nine pixels in two pipeline steps where the first pipeline step summarizes the features of each line while the second pipeline step summarizes the combined features of these lines.

Besides the three mentioned modules, no additional modules need to be changed in order to calculate nine pixels in one clock cycle.

3.6 Implementation and test strategy

The system was implemented using the Toolchain provided by Xilinx[1]. The system was implemented iteratively. Each module was independently developed and simulated using waveform simulations (see section 3.6.1). Each module was implemented as simple as possible first, to get a working prototype. The first prototype could only calculate the number of pixels for each region and could handle one pixel per clock cycle. This prototype was then tested using a hardware in the loop (HIL) system with lower timing performance (see section 3.6.2). With this system, well-defined images could be transferred from a PC to an FPGA for testing. When the first prototype was functional in the HIL system, more functionality was added to the prototype; each iteration included waveform simulation and HIL testing. After the fully developed system was functional in the HIL system, it was transferred to a live system, connected to a camera. After transferring it to the live system, several timing optimizations had to be made, to allow for the necessary throughput. Then, the live system was validated against the same algorithm running on a PC (see section 4).

3.6.1 Waveform simulation

Each module was simulated using the waveform simulator Xilinx ISim[2]. With waveform simulators, all inputs, outputs and internal signals can be simulated and observed. Usually, the test inputs are described as a sequence of input assignments and timing statements. Such inputs are unintuitive and poorly maintainable if a system is tested that is designed for images. Therefore, additional simulation files have been developed to allow for better image simulation.

First, a virtual camera module has been developed. This virtual camera module reads text files and interprets them as image data. In these text files, each character can be a digital number between 0 and 9, 0 indicating background and 1-9 indicating different foreground intensities. The virtual camera translates this into the basic image signals frame valid, line valid, data valid and pixel data. The virtual camera has different configuration optins, that can be altered for each simulation. These options include frame blanking time, line blanking time and several features to produce different variations of basic image signals for the same image. All options for the virtual camera are shown in table 3.3.

A second module was developed that has the region data of the first pass module as its input. This module writes the assigned regions for each pixel into a text file. Thus, the correct behavior of the first pass label assignment can be reviewed.

[1] `http://www.xilinx.com`, Last access: October 15, 2014.
[2] `http://www.xilinx.com/tools/isim.htm`, Last access: October 15, 2014.

option name	data type	default	description
clock period	time	20 ns	clock period for the pixel clock
input file	string	sim_testData.txt	The name of the file containing the image data
hblanking	integer	6	the number of horizontal blanking clock cycles
vblanking	integer	6	the number of vertical blanking clock cycles
dvblanking	integer	1	the number of clock cycles between the rising edge of line valid and the rising edge of data valid
num conc pixel	integer	3	The number of pixels read at the same clock cycle
uses dv skipping	boolean	true	If this option is activated, every second cycle data valid is set to 0 and no pixel data is sent
pixel size	integer	8	The number of bits for each pixel.
repeat	boolean	false	If set to false, the image is sent only once. If set to true, it will be sent again after the vertical blanking time

Table 3.3: The different configuration options of the virtual camera module.

Additionally, the information in this file is useful when simulating other parts of the system. The third extra module is a module that has the basic feature data as its input and writes the data into a text file to provide a more intuitive view for the algorithm's results. Figure 3.20 shows the waveform simulation architecture.

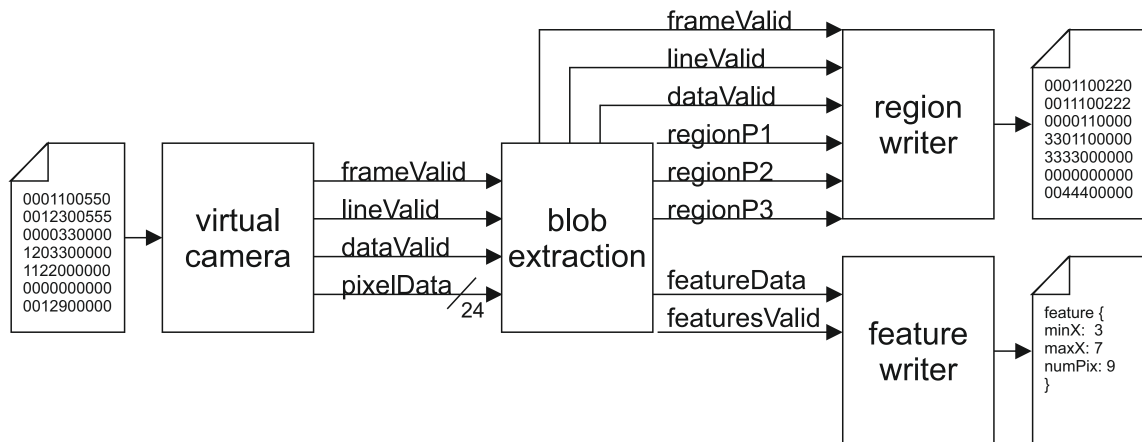


Figure 3.20: Architecture of the waveform simulation. The virtual camera reads the text file and transforms the data to the image stream data signals. The outputs of the BLOB extraction are sent to two different modules. The label information is streamed to the region writer in a similar way as the image stream data. The feature Data is streamed to the feature write. Both modules write the received data into a text file, if a format that is readable by humans.

With this architecture, several different image types and complex union situations could be tested fast and easy. Furthermore, if wrong behavior by the algorithm was detected during the HIL testing, the problem could be isolated and transferred to a simulation text file. Then, the problem would be reproduced in the waveform simulation environment. There it could be identified and tracked more easily, because the waveform simulation allows the observation all internal signals of the system.

3.6.2 Hardware in the Loop

With the hardware in the loop interface, the behavior of the synthesized system can be observed and tested with well-defined data. To be used in a HIL, the BLOB-extraction module was used with an FPGA Development Board. For this,

the Digilent Atlys Board[3] was used. The board features a Xilinx Spartan-6 LX45 FPGA. But most importantly, it contains an EZ-USB FX2LP[4] programmable USB2 Chip. The chip is connected to the FPGA by a parallel interface. This chip is used to transfer image data from the PC to the FPGA. It is also used to transfer image and feature data from the FPGA to the PC host. Figure 3.21 shows the coarse design of the HIL architecture.

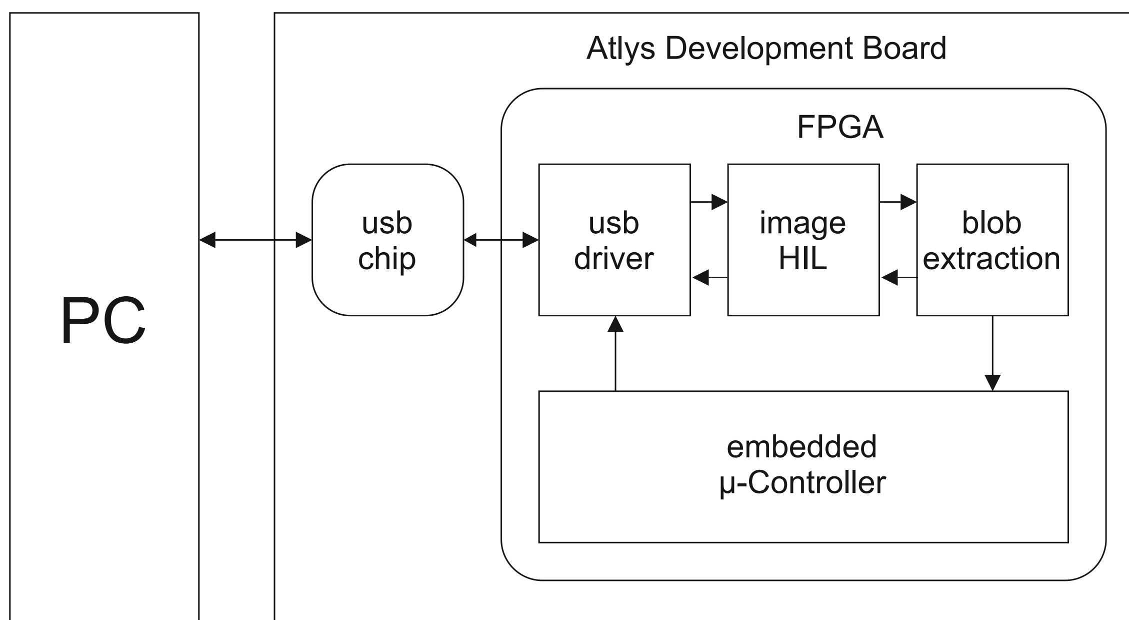


Figure 3.21: Architecture of the hardware in the loop system. The PC sends image data to the USB chip on the Atlys development board. The usb-driver module implemented on the FPGA controls the USB-chip. The received data is transmitted to the image-HIL system. The image-HIL system interprets the data and converts it to the image-stream signals. The image-stream signals received from the BLOB-extraction are converted to data frames that are sent back to the PC. The raw feature data of the BLOB-extraction is sent to the embedded µ-controller that is synthesized on the system. The µ-controller performs the final post-processing steps and sends the feature data to the PC using the usb-driver.

The behavior of the HIL system is as follows: The PC sends image data to the USB chip on the Atlys development board. The usb-driver module implemented on the

[3] `http://www.digilentinc.com/atlys`. Last access: October 15, 2014.

[4] `http://www.cypress.com/?id=193`. Last access: October 15, 2014.

FPGA controls the USB-chip. It stores the data received from the PC to an output FIFO. The image-HIL system reads the first word containing the image size from the FIFO. Then it waits until at least one line is available at the FIFO. The line is read from the FIFO and converted to the image-stream signals. These stream signals are forwarded to the BLOB extraction system. The image-stream signals received from the BLOB-extraction are converted to data frames that are sent to the usb-driver's input FIFO. The raw feature data of the BLOB-extraction is sent to the embedded µ-controller that is synthesized on the system. The µ-controller performs the final post-processing steps and sends the feature data to the PC using the usb-driver's special purpose registers.

The correct computational behavior was validated using the HIL system. The test images were transmitted to the FPGA and the region information was transmitted back. State of the art software-algorithms were also used to calculate the connected components and their features. For this, the Open source Computer Vision (OpenCV[5]) library was used (Bradski, 2000). On the software side, after thresholding the image, the outer contours of objects were calculated (Suzuki and Abe, 1985). Image moments were used to derive the center of gravity and the number of pixels (Chaumette, 2004). The bounding rectangle can be directly calculated from the points in the contour. The main axis is calculated by creating the covariance matrix from all points in the image and calculating the eigenvalues and eigenvectors (see section 3.2). For all of the operations described above, methods from the OpenCV library were use to ensure correct behavior of the reference system.

The results of both implementations were automatically compared and recorded using several especially prepared images. The goal was to ensure that all possible combinations of label joining and shrinking were tested. In addition to the handcrafted images, several randomly generated images were used for testing.

Besides the validation of the feature data, the maximum occupancy of all queues in the system were collected after each image. From this, the maximum capacity for all queues was computed. Especially the number of needed labels is interesting, as a small number significantly reduces the systems RAM usage and the computational complexity of the linked labels module. In addition to the images shown in figure 3.22, 25000 randomly generated images were tested. These images were the size of 1023×1024 pixel. They are one pixel less than the standard size of 1024×1024 because three pixels are transmitted in one clock cycle, therefore the image width must be dividable by 3. In the images, between 500 and 5000 objects were randomly placed. The objects were ellipsoid and had random expansions and rotation. The expansion sizes of the ellipses were between 2 and 30 pixels. The random placement allowed objects to intersect each other. Figure 3.23 shows

[5] http://www.opencv.org. Last access: October 15, 2014.

Figure 3.22: Different test images. The images in the upper line are small test images for special join cases. The lower left image features different shapes. The center image is designed to verify the PCA calculation. The right image is the worst case scenario for the connected component labeling with 3 concurrent pixels.

examples of the randomly generated test images. For all randomly generated images, the maximum number of needed labels was 148. The maximum size of the join queue was 7 and the maximum size of the finished queue was 50 elements. For the random test, the calculated features were not validated. Only the queue and buffer sizes where checked, as well as the number of found objects.

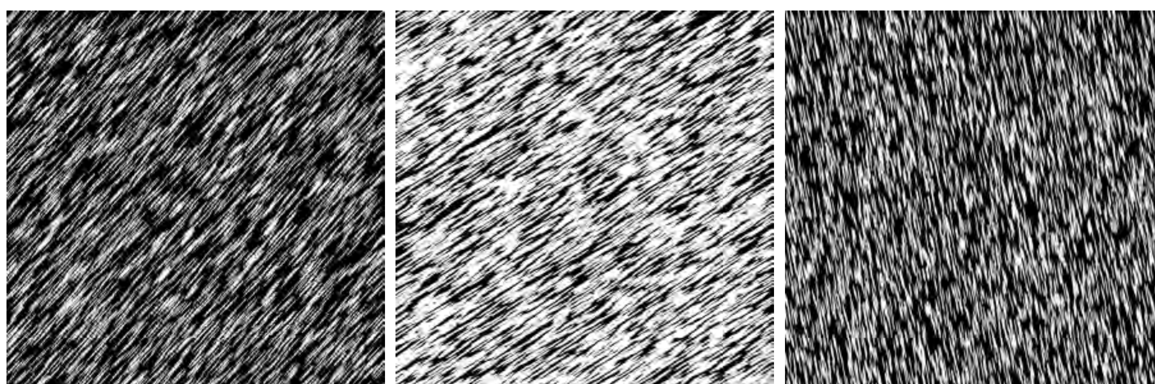


Figure 3.23: Randomly generated test images. The images are 1023×1024 pixels in size. Less than 150 labels are needed for all randomly generated images. The maximum queue sizes for the join queue is 7 elements and 50 elements for the finished queue.

As stated in section 3.1, the algorithm should work on all kinds of images. Therefore, in addition to the test of hand-crafted images and images with random objects,

noisy images were tested. As the algorithm behavior depends on the segmentation based on thresholding, salt and pepper noise was identified to be the noise with the greatest effect on the algorithm. To ensure that the algorithm does not fail on noisy images, random salt and pepper noise was generated and tested with the algorithm. 25000 randomly generated images with different sizes were tested. Images with the sizes of 1023×1024 pixel, 510×512 pixel, 255×256 pixel and 126×128 pixel were tested. Additionally, images with the size of 126×1024 were tested to investigate the impact of the height on the buffer sizes. For each size, 5000 images were tested. To generate the images, uniformly distributed random numbers between 0 and 255 were assigned to every pixel of the test image. Then, the image was thresholded to create a black and white image. The threshold value was a random number between 0 and 255. By using a random threshold, not only the distribution of the noise varied in every image but also the black to white ratio. Table 3.4 shows the resulting needed buffer sizes.

resolution	2046^2	1023^2	510^2	255^2	126^2	126×1024
labels	502	261	139	73	38	41
join queue	18	10	6	4	2	3
finished queue	333	173	89	49	26	27
labels (% of width)	25 %	25 %	27 %	29 %	30 %	33 %
join queue (%)	1 %	1 %	1 %	2 %	2 %	2 %
finished queue (%)	16 %	17 %	17 %	19 %	20 %	21 %

Table 3.4: The queue sizes for the different tested resolutions with salt and pepper noise images. It can be seen from the results, that the queue size depends on the image width. It is also shown that the image-height has no significant impact.

The measurements of the noise and the random objects show that the algorithm's architecture is robust against noise and can handle any kind of image. Furthermore, the sizes of the different queues in the system can be deduced from the maximum desired image size. As in such a system robustness is more important than speed and area optimization, the estimation should be made conservatively. The following equations were used to layout the system used in the applications.

$$\#\text{labels} = \max\left(\frac{mw}{2}, 128\right) \tag{3.25}$$
$$\text{queueSize}_{finished} = \max\left(\frac{mw}{3}, 64\right) \tag{3.26}$$
$$\text{queueSize}_{join} = \max\left(\frac{mw}{8}, 32\right) \tag{3.27}$$

3.6.3 Resource usage and throughput

The throughput was analyzed by using the statistical analysis of the Xilinx Toolchain after the designs has been translated, routed and mapped to a system. The process was performed for different kinds of FPGAs to elaborate the FPGA types and architectures that fulfill the requirements stated in section 3.1.

The following features were utilized for this analysis: number of pixels, bounding box, center of gravity and PCA. Later, the BLOB extraction algorithm must be connected to an embedded processor to carry out the post-processing steps of the features. In this case, it is desirable for the BLOB extraction to be clocked with the same clock as the communication bus. For most systems, this bus runs at a frequency of 100 Mhz. Therefore, this frequency should be achievable by the hardware design.

Four different Xilinx FPGA models and one Xilinx System on Chip (SoC) were analyzed. At the publishing date of this thesis, the FPGA series 7 was the most recent Xilinx FPGA series. The following FPGAs and SoCs were analyzed. An FPGA model for each FPGA series was chosen that is available on a ready to use evaluation platform.

- The Spartan 6 FPGA, which is a well established low-cost FPGA. The XC6SL45 (Speed grade -2) was used, which is integrated in the Digilent Atlys[6] development board.
- The Artix 7 FPGA, which is the successor of the Spartan series. The XC7A100T (speed grade -1) model was used, which is integrated in the Digilent Nexys 4[7] development board.
- The Virtex 7, the high-end FPGA series. The XC7VX485T (Speed grade -2) model was used, which is integrated in the Virtex 7 VC707 Evaluation platform.

[6] `http://www.digilentinc.com/atlys/`. Last access: October 15, 2014.
[7] `http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS4`. Last access: October 15, 2014.

- The Kintex 7, a medium priced FPGA located between the Artix and the Virtex. The XC7K325T (Speed grade -2) was used, which is embedded in the Kintex 7 KC705 Evaluation platform.
- The Zync 7000 SoC system, which features a processing system with an ARM core and programmable logic. The XC7Z020 (Speed grade -1) was used, which is integrated in the ZedBoard[8] development platform.

The design was synthesized, translated and mapped to each of the chosen models. Whether the design fits on the chosen model and how many resources it needed of the critical FPGA key resources look-up tables (LUTs), flip-flops, block RAM and multipliers/DPS cells was analyzed. As the design uses several queues, buffers and RAMs, especially the block ram utilization is of interest. Furthermore, the throughput was analyzed using the Xilinx *Post-Place & Route statistic timing analyzer*, embedded in the ISE design suite. The timing analyzer finds the longest path in the system and analyzes the needed times for the logic and the signal transmission between the logic elements. Based in the minimum achievable time of this path, the maximum clock frequency of the system can be calculated. Table 3.5 shows the needed resources and maximum system clock for each model. For the analysis, the maximum width and height were both set to 2048 (4 MP). The number of labels and the queue sizes were set as hinted at in equations 3.25 to 3.27. Table 3.6 shows the needed resources and maximum system clock for each model for the image size 1024×1024.

Tables 3.6 and 3.5 show that for most FPGA architectures the device utilization does not change significantly. Only the spartan architecture has a significantly higher block RAM usage for larger images. This is due to the fact that the Spartan6 architecture has smaller block RAMs than the series 7 devices. The block RAMs in the 7 series are 36 bit wide while the block RAMs in the Spartan6 series are only 16 bit wide. When changing the maximum image size, the sizes of the different features are exceed 16 bit and need several RAMs.

In table 3.7, the different achievable throughputs and resource utilizations for different maximum image sizes are shown. It can be seen that with increasing image sizes, the number of needed block RAMs increases slightly while the maximum throughput decreases slightly. When using image sizes of up to 4096×4096 (16 MP), the number of block RAMs increases significantly. This is due to the fact that for the image size of 16 MP, several features exceed one of the natural RAM combination sizes (for the 7 series 18, 36, 54 ...) and need the next RAM combination, leading to higher RAM utilization.

Overall, the device utilization and the speed is dependent on the maximum image size, with a small throughput decrease for larger images.

[8] `http://www.zedboard.org`. Last access: October 15, 2014.

model	Spartan 6	Artix 7	Virtex 7	Kintex 7	Zynq
Slices available	6,822	15,850	75,900	50,950	13,300
Slices used	1,825	1,875	1,943	1,782	1,537
Slices used (%)	27 %	12 %	3 %	3 %	12 %
block RAM available	116	135	1,030	455	140
block RAM used	45	25	25	25	25
block RAM used (%)	39 %	19 %	2 %	6 %	18 %
DSP slices available	58	240	2,800	840	220
DSP slices used	7	7	7	7	7
DSP slices used (%)	12 %	3 %	1 %	1 %	3 %
system frequency	108 Mhz	112 Mhz	190 Mhz	190 Mhz	120 Mhz
Throughput	0.96 GB/s	1.00 GB/s	1.70 GB/s	1.70 GB/s	1.07 GB/s

Table 3.5: The needed resources and the maximum achievable timing for different models of different families of Xilinx for images with the size of 2048×2048.

model	Spartan 6	Artix 7	Virtex 7	Kintex 7	Zynq
Slices available	6,822	15,850	75,900	50,950	13,300
Slices used	1,850	1,698	1,769	1,721	1,832
Slices used (%)	27 %	11 %	2 %	3 %	14 %
block RAM available	116	135	1,030	455	140
block RAM used	25	24	24	24	24
block RAM used (%)	22 %	18 %	2 %	5 %	17 %
DSP slices available	58	240	2,800	840	220
DSP slices used	7	7	7	7	7
DSP slices used (%)	12 %	3 %	1 %	1 %	3 %
system frequency	118 Mhz	125 Mhz	178 Mhz	204 Mhz	120 Mhz
Throughput	1.05 GB/s	1.11 GB/s	1.59 GB/s	1.82 GB/s	1.07 GB/s

Table 3.6: The needed resources and the maximum achievable timing for different models of different families of Xilinx for images with the size of 1024×1024.

model	4096×4096	2048×2048	1024×1024	512×512
Slices used	1,918	1,537	1,832	1,618
Slices used (%)	14 %	12 %	14 %	12 %
block RAM used	48	25	24	21
block RAM used (%)	34 %	18 %	17 %	15 %
system frequency	113 Mhz	120 Mhz	122 Mhz	128 Mhz
Throughput	1.01 GB/s	1.07 GB/s	1.09 GB/s	1.14 GB/s

Table 3.7: Resource usage for different maximum image sizes for the Zynq architecture

The decision which FPGA/SoC system to use depends on several additional factors. The targeted throughput needs to be taken into consideration, as well as other hardware components that will be utilized on the system in addition to the BLOB detection system. For this thesis, the Zynq System-on-Chip[9] (SoC) architecture was chosen. This architecture contains an ARM dual-core Cortex-A9 processor as well as programmable logic that can be directly coupled with the processor. Furthermore, the chip features many communication peripherals. The system is ideal, because it features programmable logic powerful enough to implement the BLOB extraction with the desired throughput and additional features and a processing system that is more powerful than the Soft-core processors used on pure FPGA systems. The processing system is well-suited to perform the post-processing steps of the binary large object extraction and the classification of objects.

3.7 Conclusions

In this chapter, the original idea and the implementation details of a novel approach for hardware-based BLOB detection were given. The chapter describes novel ideas and implementations that differ from the state of the art.

Three novel features were introduced that allow for better classification and more precise positioning.

Furthermore, a novel hardware approach of the BLOB extraction algorithm was described. The algorithm is extremely robust against different image contents and allows for a very high throughput. The throughput goal was exceeded by every possible FPGA configuration. For high-end FPGAs, the achievable throughput is 2.5 times higher than the goal. With the maximum throughput of approx. 1.8 GB/s, it is possible to track an object with an update-rate of 1.8 KHz if an image size of 1 MP is used. when operating inside a ROI (e.g. 200×200 pixel), speeds of up to 450 KHz become possible.

In the next chapter, the validation of latency and jitter will be discussed.

[9] `http://www.xilinx.com/products/silicon-devices/soc/`. Last access: October 15, 2014.

4 Validation

While section 3.6 analyzed the robustness and the correct computational behavior, this chapter validates the timing requirements for update-rate, latency and jitter. Three different systems were validated using the same measurement setup. Besides the FPGA-based system, the state-of-the-art approach using PC-based image-processing was measured. The last measured system was a PC running a real-time kernel. A real-time kernel can execute image-processing without interruptions through other processes on the system and therefore should optimize the timing behavior of PC-based image-processing in terms of jitter.

4.1 Test setup

The timing behavior was tested using the setup shown in figure 4.2. It features a high-speed CameraLink camera monitoring a set of eight LEDs. The camera used is the model EoSens-3CL from Mikrotron[1]. The camera is configured to record a video of 1000×1000 pixels. The shutter-time, the gain and the white balance are configured to show a black image with the enabled LEDs visible as bright regions, as shown in figure 4.1

The camera is connected via a CameraLink base interface to the FPGA programmed with the BLOB extraction algorithm. The BLOB extraction FPGA sends the number of detected regions after each image to a second control-FPGA via an UART[2] bus. This control-FPGA is configured to control the LEDs and measure the timing between frames. It sends the measured timing to a control PC over another UART connection. The timer equipped in the measuring-FPGA has a resolution of 10 ns.

The UART buses in this setup were chosen because this way the FPGA containing the BLOB-detector can be easily exchanged with a PC, as shown in figure 4.3. This allows for direct timing comparison between software-based image-processing and FPGA-based image-processing. For comparison, a PC with a CameraLink framegrabber was used. The operation system was Ubuntu Linux, equipped with a

[1] `http://www.mikrotron.de/high-speed-camera-solutions/machine-vision-kameras/cameralinkr.html`. Last access: October 15, 2014.

[2] universal asynchronous receiver/transmitter

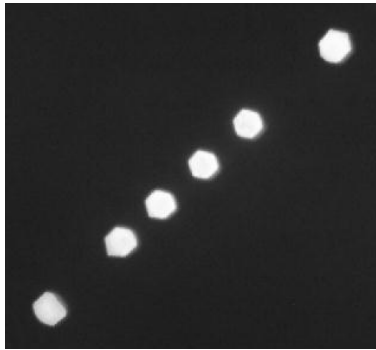


Figure 4.1: Image observed by the test setup. The image currently shows six active LEDs, the corner LEDs and the four center LEDs.

real-time enhanced kernel. The software BLOB algorithm was measured with and without enabled real-time features of the kernel. The used PC system is a Core i5 with 3800 Mhz. The attached CameraLink framegrabber is a Matrox Solios eCL[3] PCIe framegrabber.

Several different measurements were carried out validating the update-rate jitter, the latency and the latency jitter. The influence of the workload on the PC was analyzed as well as the influence of the image content (i.e. number of objects). In the next section, the FPGA prototype that was used to test the FPGA implementation is described. Afterwards, the different measurements are presented and analyzed.

4.2 Prototype

The live system facilitates the hardware configuration shown in figure 4.4. The Xilinx Zynq development board Zedboard was used for the reason that it features the Zynq System-on-Chip[4] (SoC) architecture. This architecture contains an ARM dual-core Cortex-A9 processor as well as programmable logic that can be directly coupled with the processor. Furthermore, the chip features many communication peripherals. The system is ideal because it features programmable logic powerful enough to implement the BLOB extraction with the desired throughput as well as a processing system that is more powerful than the soft-core processors used on pure FPGA systems. The processing system is well-suited to perform the post-processing steps of the binary large object extraction and the classification of objects. The chosen development board contains a Zynq Z-7020 SoC device. It was extended with a self-developed add-on board. The add-on board extends

[3] `http://www.matrox.com/imaging/de/products/frame_grabbers/solios/solios_ecl_xcl_b/`. Last access: October 15, 2014.

[4] `http://www.xilinx.com/products/silicon-devices/soc/`. Last access: October 15, 2014.

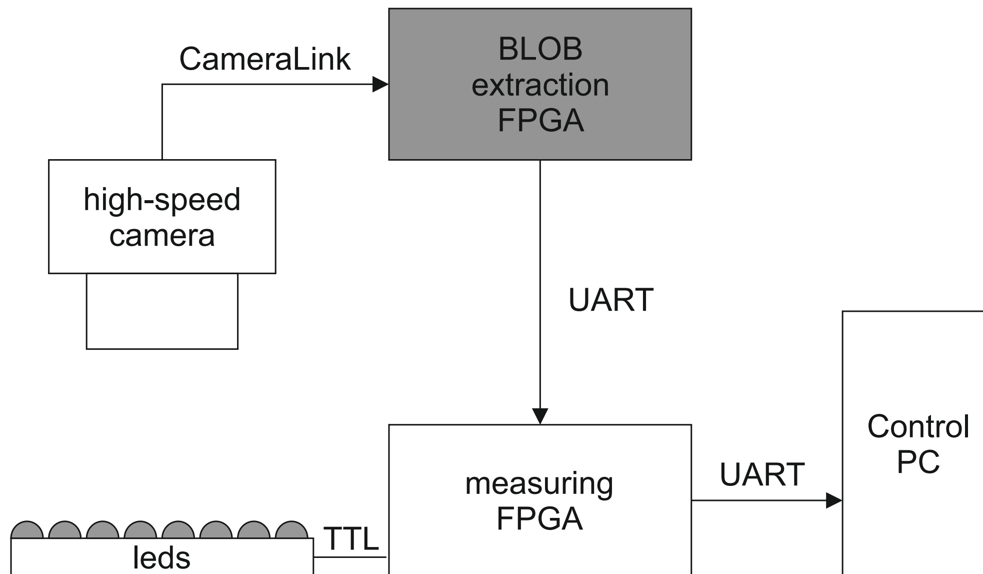


Figure 4.2: The setup for timing validation. An LED array consisting of eight LEDs in controlled by an FPGA. The high-speed camera is configured to show bright regions in a black image for each powered LED. The result of the BLOB extraction is transferred to the measuring FPGA. This FPGA contains a high-resolution timer that can perform different measurements. The results of the measurements are transferred to a control PC.

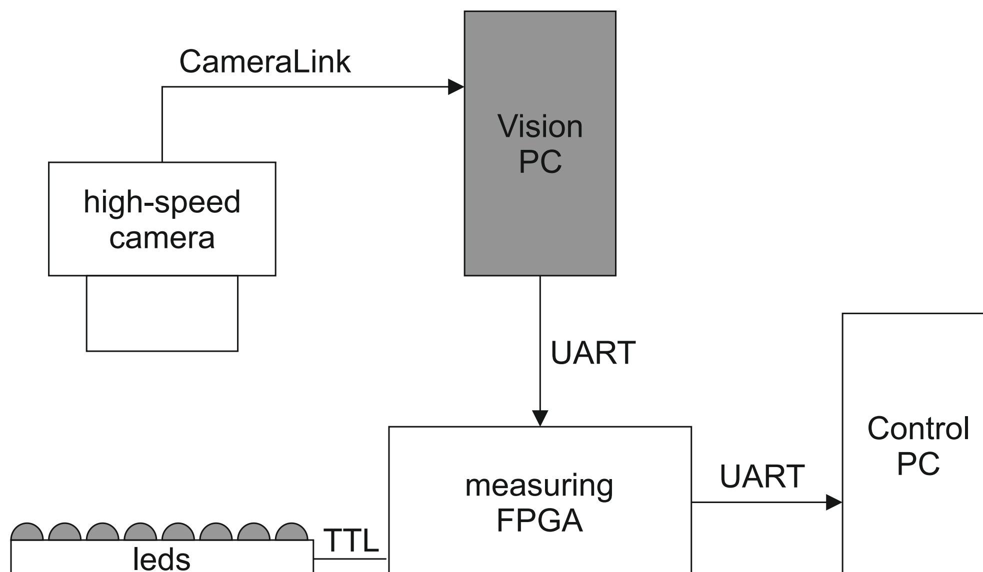


Figure 4.3: The setup is similar to the setup shown in figure 4.2. Only the BLOB extraction FPGA was replaced with a image-processing PC.

the Zedboard with a CameraLink interface. The board contains an MDR-26 connector to which CameraLink cameras can be connected. It contains the channel link receiver DS90CR288A[5] from Texas Instruments. The channel link receiver transforms the LVDS data of the CameraLink signal to a parallel signal. The parallel signal and the recovered clock are transferred to the FPGA. The channel link receiver outputs 24 bits of data. Therefore, only three concurrent pixels can be received with this board. The receiver can operate with pixel clocks of up to 85 Mhz, which is the fastest specified speed in the CameraLink protocol. Furthermore, the board features simple LVDS converters for the CameraLink control signals and the serial interface (The DS90LV047A[6] for the CameraLink control and the DS90LV019[7] for the serial connection, both from Texas Instruments).

The communication of the found objects can be done through different channels. The development system features many different communication peripherals such as Gigabit Ethernet or serial interface (UART over USB). Furthermore, the processing system features more communication protocols that can be used if a corresponding extension board is connected to the board's pin connectors. Two different communication protocols were used. The serial interface was used for testing. The serial interface was chosen because it is also available at an off-the-shelf PC. This way, the PC and the hardware system can be evaluated against each other using the same interfaces. The second communication interface used is a Controller Area Network (CAN) controller. The CAN interface has the benefit of being able to have multiple participants. This is necessary in one of the targeted applications (section 5.1).

The BLOB extraction described in chapter 3 was integrated into the system. It uses the three pixel approach instead of the nine pixel approach, to ensure better comparability with PC-based image processing that can only handle CameraLink base setups (i.e. three pixels with a pixel clock of 85 MHz). To be able to process data from the ChannelLink chip, the data must first be converted. Also, it is desirable for the programmable-hardware modules to be driven by a clock that is generated by the system rather than by a clock that is driven by an attached extension hardware. Therefore, the data arriving from the channel link chip are pushed into a dual-ported FIFO with asynchronous read and write ports. Data is only pushed into the FIFO if valid data arrives (data valid is high) or if one of the other control signals, frame valid or line valid, change their state. The data is pulled from the FIFO using the system's main clock. The system clock has a frequency of 100 Mhz. Thus, the system clock is always faster than the camera clock, as the maximum frequency of CameraLink is 85 Mhz. The FIFO is only needed to transfer the image data from one clock domain to the other and the size

[5]http://www.ti.com/product/ds90cr288a. Last access: October 15, 2014.
[6]http://www.ti.com/product/ds90lv047a. Last access: October 15, 2014.
[7]http://www.ti.com/product/ds90lv019. Last access: October 15, 2014.

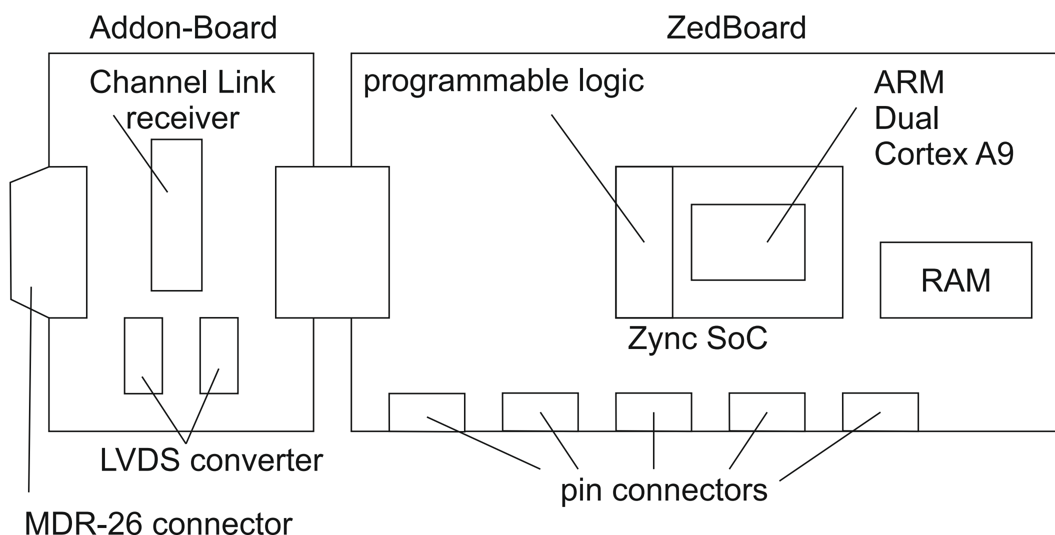


Figure 4.4: Overview of the hardware platform used. A CameraLink camera can be connected to the MDR-26 connector of the add-on board. The channel link receiver transforms the CameraLink LVDS data to parallel data. The parallel data is streamed into the programmable logic part of the SoC. The camera can be controlled by using the serial signal of the CameraLink cable or by using the CameraLink control lines.

can therefore be set as small as possible, which is 16 elements. The read side of the FIFO is faster than the write side. Therefore, the FIFO will be empty regularly during the image transmission. If the FIFO is empty, the current frame valid and line valid signals are kept and data valid signal is set to low, indicating a short pause in the transmission.

If a color-camera is attached to the system, the image transferred is Bayer encoded (Bayer, 1976). To generate a gray image, the incoming data must first be converted to RGB color data and than converted from RGB to grayscale data. When converting from RGB to gray, the different color channels are treated differently (Poynton, 2012):

$$E'_Y = 0.299E'_R + 0.587E'_G + 0.114E'_B \tag{4.1}$$

For a hardware system, this can be approximated. The goal is to have a formula that has multiplications with integer numbers and a division by 2^n, as such a division can be realized with a shift operation. The equation 4.2 is the transformed equation 4.1 that meets these criteria.

$$E'_Y = \frac{77E'_R + 150E'_G + 29E'_B}{256} \tag{4.2}$$

The BLOB extraction is clocked with the 100 Mhz system clock. As this clock is slightly faster than the camera clock, at least 15 % of the cycles will be cycles with no new data. due to this condition, the BLOB extraction has more time during the parallel equivalence handling thus relaxing the constraints of the system.

The BLOB extraction module is integrated with an AXI bus controller. The bus controller is responsible for communicating with other components in the system and interpreting the bus signals. A small FIFO buffering feature data of finished regions is implemented in the wrapper. The BLOB extraction pushes finished data into this FIFO. If the FIFO contains at least one element, an interrupt is triggered to the ARM processor. A second interrupt is triggered when the BLOB extraction system signals the end of an image frame. Additionally, the AXI bus wrapper defines different registers that are readable and writable from the ARM processor using the AXI bus. The writable registers include registers for the threshold, the basic filters and control registers. The control registers are used to pop a value from the FIFO and to reset the interrupts. The readable registers consist of several registers to read the basic feature data. If a value is popped from the FIFO, the results are stored in these registers. Additionally, a status register can be read. The status register contains flags for the FIFO state and the finished state of a frame as well as error flags if any of the system queues overflowed.

The system architecture of the SoC is shown in figure 4.5

Several different measurements were carried out validating the update-rate jitter, the latency and the latency jitter. The influence of the workload on the PC was analyzed as well as the influence of the image content (i.e. number of objects). In the next section, the different measurements are presented and analyzed.

4.3 Update-rate jitter

To measure the update-rate jitter, the time between UART messages arriving from the image-processing system (PC or BLOB-extraction FPGA) was measured. The measurement was carried out for each of the three systems. For each system, the jitter was measured for different image contents. Furthermore, for the PC based image processing, an artificial workload was added to the PC to analyze the impact of other processes onto image-processing. Two different measurements were carried out: for one object and for eight objects. Both were measured with and without additional workload for PC-based image processing.

The update-rate was measured continuously for 10.000 consecutive updates. The measured value is the time that has elapsed since the last update.

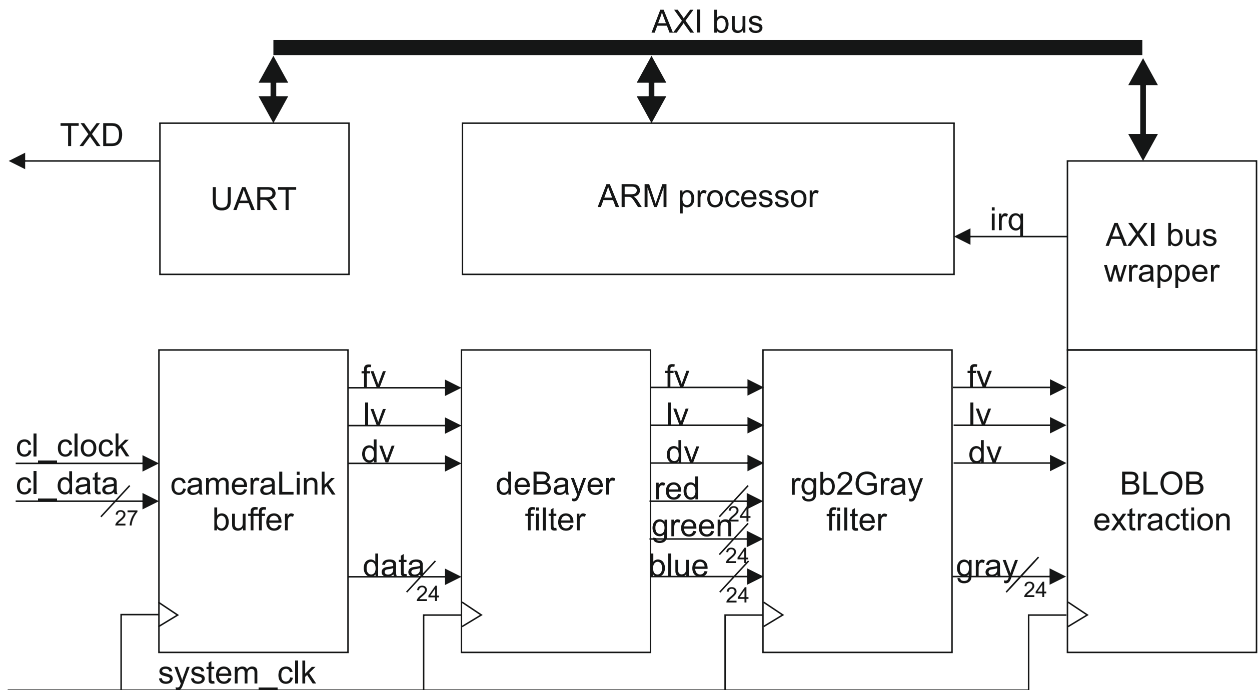


Figure 4.5: Overview of the FPGA design used. The image data is received by the CameraLink buffer. It extracts the correct data and pushes the data into a dual ported FIFO. The data is pulled from the FIFO with the system clock. The system clock has a frequency of 100 Mhz. All following subsystems work with the system clock. After the image data is transformed to color data, it is transformed to gray-scale data before it is forwarded into the BLOB extraction module. The AXI bus wrapper of the BLOB module signals the ARM processor if a region was found or the image is finished.

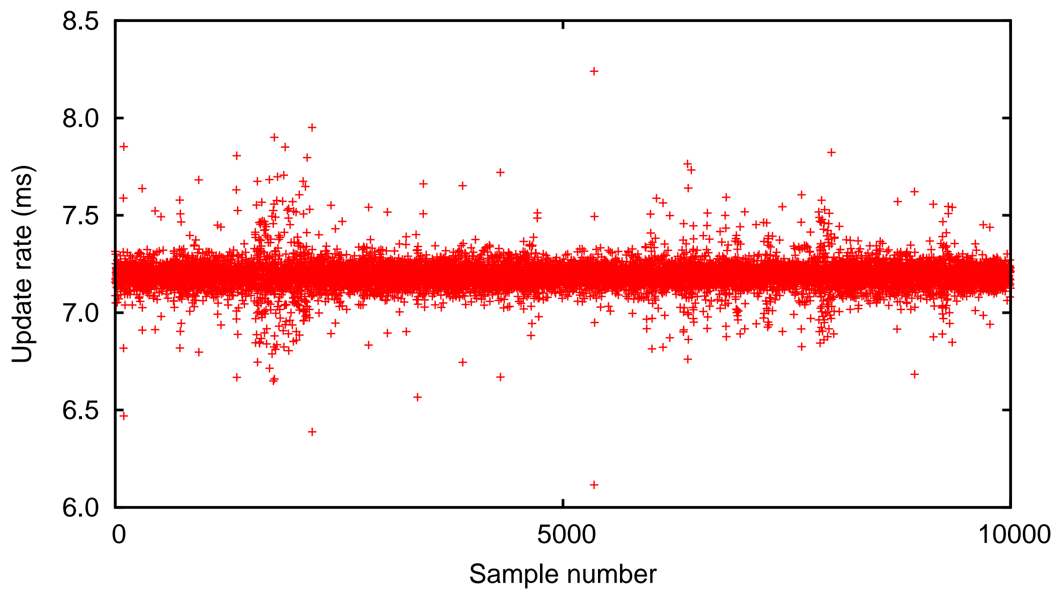


Figure 4.6: Update distribution of one object with low processor load. The elapsed time between two UART updates was measured continuously for 10000 consecutive updates. The measured value is the time that has elapsed the last update. The update-rate is between 6.12 ms and 8.24 ms with a mean value of 7.19 ms and a standard deviation of 0.08 ms.

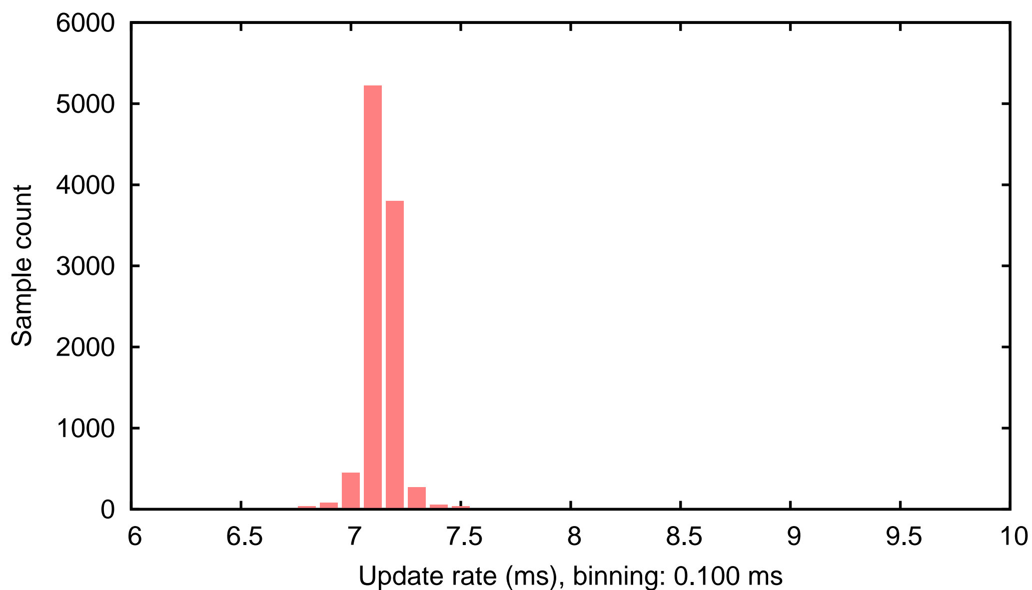


Figure 4.7: Update histogram of one object with low processor load. The update-rate is between 6.12 ms and 8.24 ms with a mean value of 7.19 ms and a standard deviation of 0.08 ms.

Figure 4.6 shows the results of the update-rate test for PC-based image-processing for one LED. The update-rate is between 6.12 ms and 8.24 ms, with most of the update distances clustering around 7.19 ms with standard a deviation of 0.08 ms. The distribution of the update-rate is shown in figure 4.7. The range of the distribution was chosen between 6 ms and 10 ms to have a better comparability between different measurements. The update-rate distribution for eight objects only differed marginally and is not displayed here (see Table 4.1 for a comparison of all measurements). Therefore, the number of objects have no impact on the update-rate distribution when the system does not have additional processor load. Observing figure 4.6, it can be seen that there is a correlation between long update times and short update times. Such a behavior is natural because the camera is sending update-rates with a constant frequency and the update-rate is measured as time that has elapsed the last update. Therefore, when one update arrives later and the next one correctly, the measured time will be shorter.

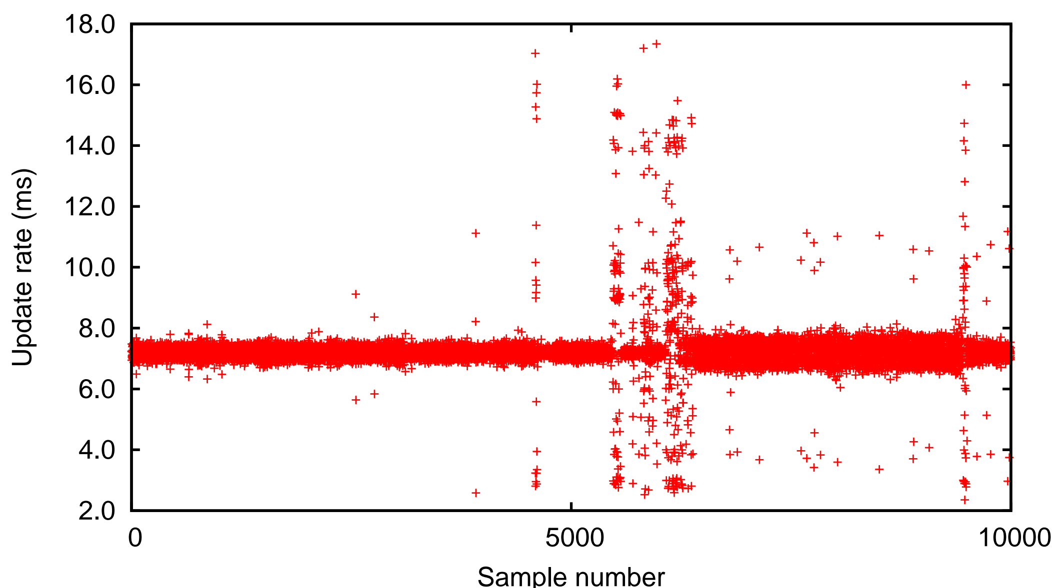


Figure 4.8: Update distribution of one object with the full processor load. The update-rate is between 2.35 ms and 17.35 ms with a mean value of 7.23 ms and a standard deviation of 0.84 ms.

The update-rate distribution changes significantly when an additional load is applied to the PC. To apply an additional load to the PC, the Linux system tool *stress*[8] was used.

[8] `http://people.seas.harvard.edu/~apw/stress/`. Last access: October 15, 2014.

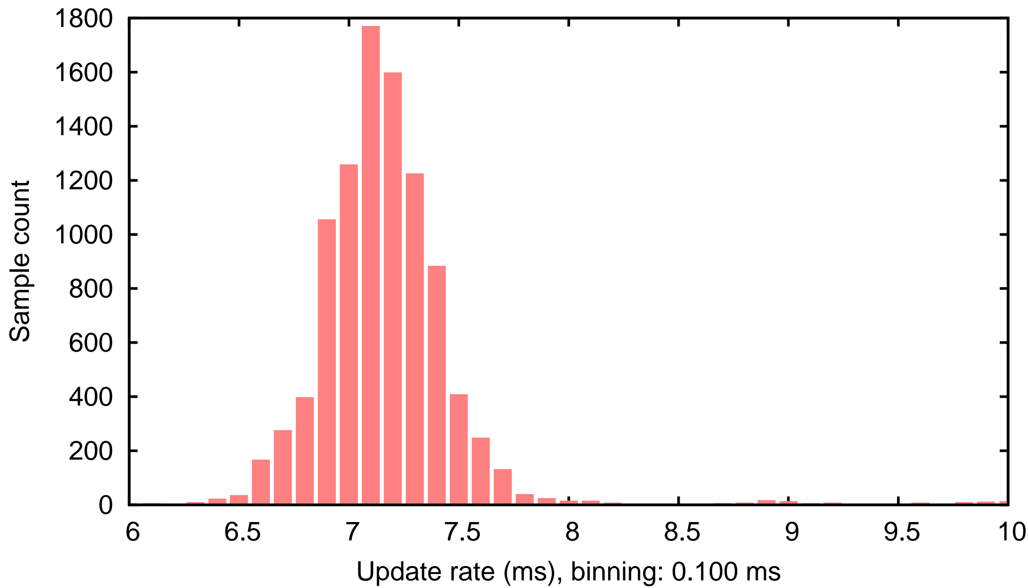


Figure 4.9: Update histogram of one object with full processor load. The update-rate is between 2.35 ms and 17.35 ms with a mean value of 7.23 ms and a standard deviation of 0.84 ms.

figure 4.8 shows the update-rate distribution of one tracked object with a full processor load. The update times range from 2.35 ms to 17.35 ms, have a mean value of 7.23 ms and a standard deviation of 0.84 ms. It can be seen in the image and the histogram image shown in figure 4.9, that still most of the updates cluster around the mean value but with a much larger distribution even if the large outliers are ignored. Furthermore, it can be seen from figure 4.8 that the large outliers are not uniformly distributed over the full measured range. Instead, bursts of periods with large outliers can be observed. One explanation for those bursts is that the process calculating the motion tracking is suppressed by other processes running on the system for several ms. The supressed process then needs several updates to recover.

The same behavior can be observed when tracking eight LEDs, but with even larger outliers than with one LED. The distribution is shown in figure 4.10. When tracking multiple objects, the tracking process has a slightly longer runtime. Therefore, it is interrupted more often by the scheduler to give CPU-time to other processes. This results in larger outliers. It can also be observed that most outliers lie in the same range as the outliers that appear when tracking one object (below 18 ms). There are less than ten outliers that have a longer update time, which is less than 0.1%. However, it can also be observed that the distribution for eight objects is larger than for one object. The histogram is shown in figure 4.11. The standard deviation for the update-rate for eight objects is 1.32 ms: quite a bit larger than

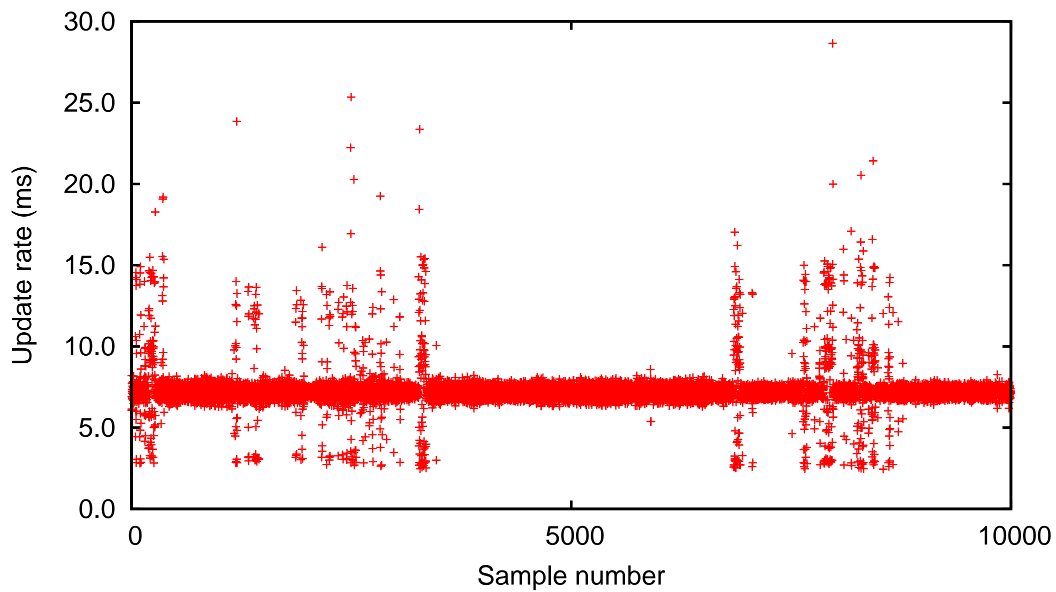


Figure 4.10: Update distribution of eight objects with a full processor load. The update-rate is between 2.45 ms and 28.65 ms with a mean value of 7.29 ms and a standard deviation of 1.32 ms.

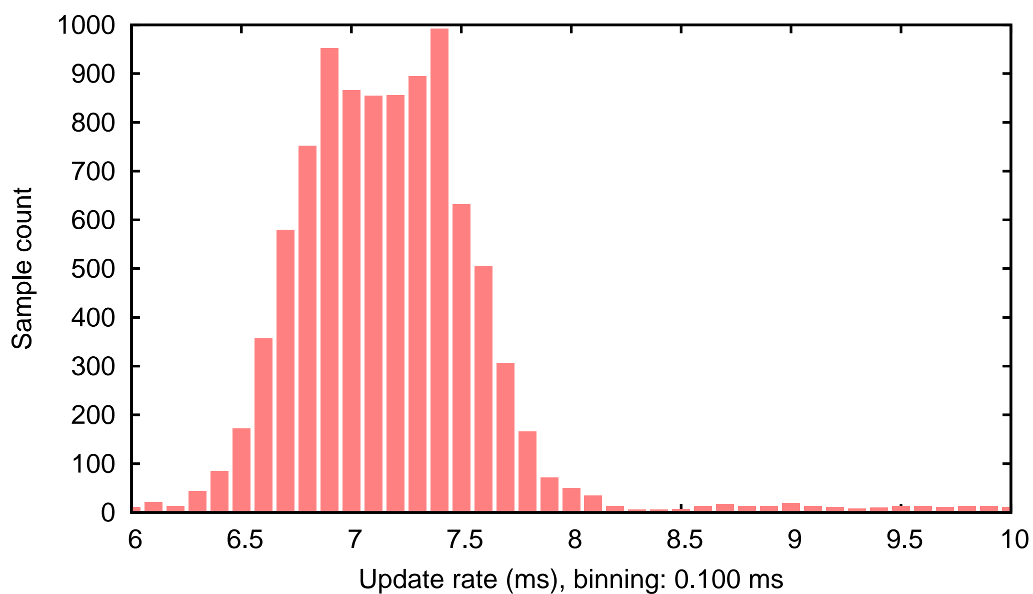


Figure 4.11: Update histogram of eight objects with a full processor load. The update-rate is between 2.45 ms and 28.65 ms with a mean value of 7.29 ms and a standard deviation of 1.32 ms.

for one object. It can be concluded that the update-rate deviation with the full processor load is larger with more image content. For the above tests, noise-free, well defined images were used. For noisy images or images with more or larger objects, the update-rate jitter will get worse. For eight objects, the update-rate is between 2.45 ms and 28.65 ms with a mean value of 7.29 ms.

The mean value of both tests with a full processor load is higher than the camera update-rate of 7.19 ms. For one object, the mean value is 7.23 ms and for eight objects 7.29 ms. The mean value would be expected to be the same as the camera update-rate, which is the case for all other measurements.

The mean value is computed from the update time between one update arriving from the tracking-PC and the next. However, the tracking PC does not have a queue for incoming images. Instead, it uses the last arrived image that was not yet processed. Therefore, updates can be lost by PC-based tracking. These lost updates are not detected by the measuring FPGA. Therefore, the higher mean values are caused by lost updates.

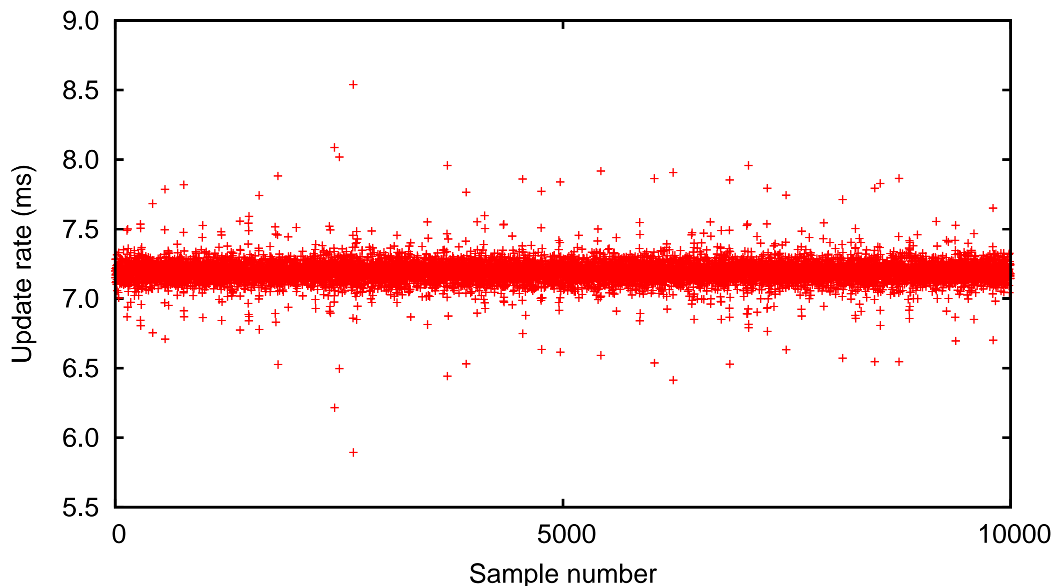


Figure 4.12: Update distribution of one object with a real-time extension. The update-rate is between 5.94 ms and 8.63 ms with a mean value of 7.19 ms and a standard deviation of 0.08 ms.

With a real-time kernel, the distribution of the update-rate can be improved. First the update-rate was measured without an artificial CPU load. Figure 4.12 shows the update-rate distribution for eight tracked objects using a real-time extension

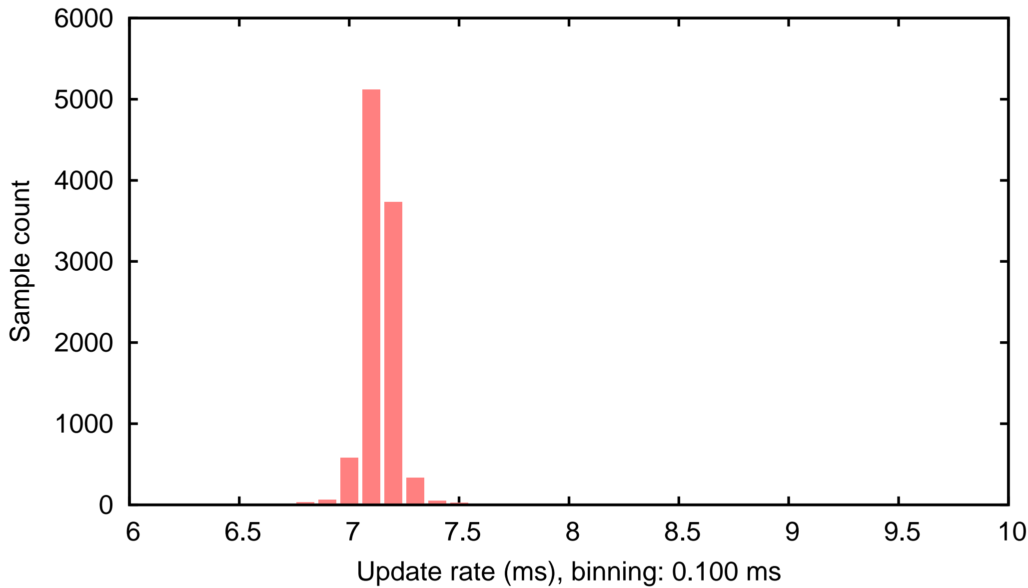


Figure 4.13: Update histogram of one object with a real-time extension. The update-rate is between 5.94 ms and 8.63 ms with a mean value of 7.19 ms and a standard deviation of 0.08 ms.

and figure 4.12 shows the corresponding histogram. The update times range from 5.94 ms to 8.63 ms with a mean-value of 7.19 ms and a standard deviation of 0.08 ms. The measurements for one object are in the same range. It can be seen from the values and the figures that, with no artificial load, the real-time extension has no benefit over the PC-based image processing without a real-time extension.

With full processor load, the real-time extension significantly improves the update-rate deviation of PC-based image processing. The update-rate is between 6.03 ms and 8.26 ms with a mean value of 7.19 ms and a standard deviation of 0.17 ms. The update-rate distribution for eight objects is shown in figure 4.14. Figure 4.15 shows the corresponding histogram. The results for one object are comparable. First of all, it can be observed than no updates are lost when using the real-time extension. Furthermore, the overall range of the update-rate did not degrade compared to the measurements with a low processor load. However, the overall distribution in the range is wider.

With the real-time extension, image-processing is executed with the highest priority and cannot be interrupted by other processes. Therefore, once image-processing has started, it will run until the result is transferred to the FPGA. Therefore, no updates are lost and no outliers outside the regular distribution are detected.

Still, the overall distribution is wider than without a load; the standard deviation increases from 0.08 ms with a low processor load to 0.17 ms with a high processor

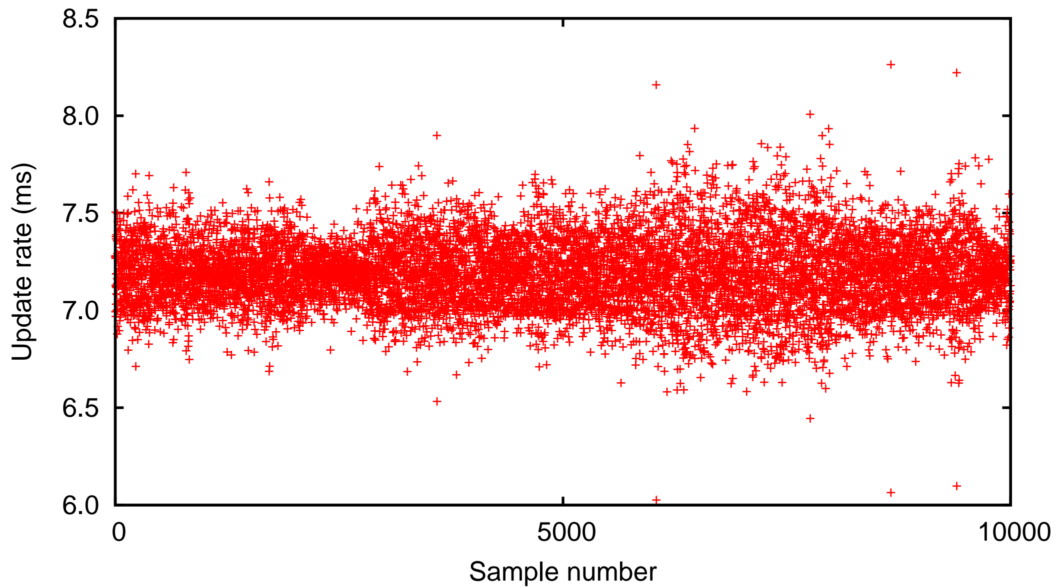


Figure 4.14: Update distribution of eight objects with a real-time extension and a full processor load. The update-rate is between 6.03 ms and 8.26 ms with a mean value of 7.19 ms and a standard deviation of 0.17 ms.

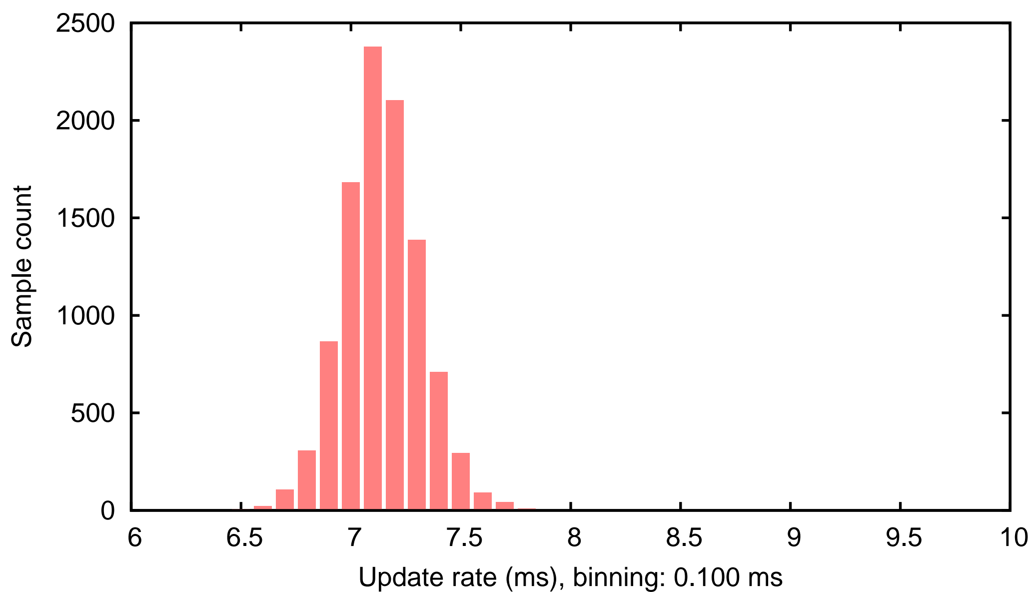


Figure 4.15: Update histogram of eight objects with a real-time extension and a full processor load. The update-rate is between 6.03 ms and 8.26 ms with a mean value of 7.19 ms and a standard deviation of 0.17 ms.

load. Therefore, image-processing is either delayed or interrupted in its execution. Image-processing cannot be interrupted by other processes, because it has the highest real-time priority. Therefore, it can only be interrupted by the kernel itself. One explanation of the wider distribution is that, with a high load, the kernel needs to reschedule all processes more often and therefore needs to interrupt image-processing dependent on the current schedule. Another explanation is that the scheduler runs with fixed intervals for context changes. If this is the case, image-processing is not started directly once a new image is available, but has to wait until the next scheduling point if all cores are assigned running processes. With low processor load, image-processing can directly be started on an idle core without having to wait for a context switch.

In any case, the overall update-rate is improved by the real time extension. There are no more lost updates and the bursts that can be observed without real-time extensions are eliminated as well. Furthermore, the minimum and maximum update times only deviate from the mean by approx. 1.2 ms (17 %).

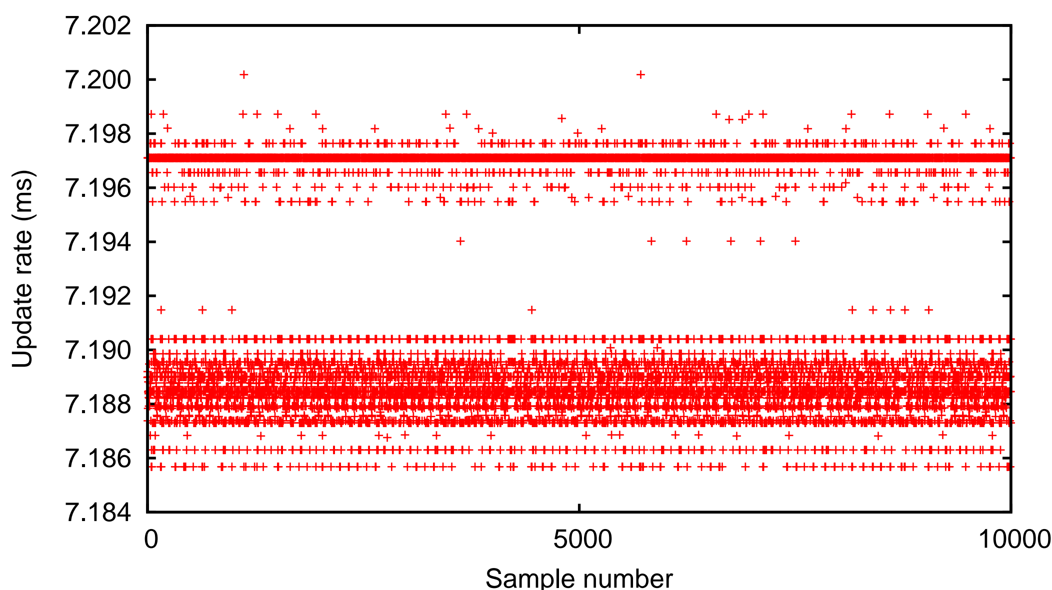


Figure 4.16: Update distribution of eight objects tracked with FPGA. The update-rate is between 7.19 ms and 7.20 ms with a mean value of 7.19 ms and a standard deviation below 0.01 ms (4.0 µs).

Using FPGA-based object tracking, the update-rate deviation is virtually jitter-free. The update-rate ranges from 7.19 ms to 7.2 ms. The standard deviation is 4 µs. The update-rate deviation for eight objects is shown in figure 4.16, the corresponding

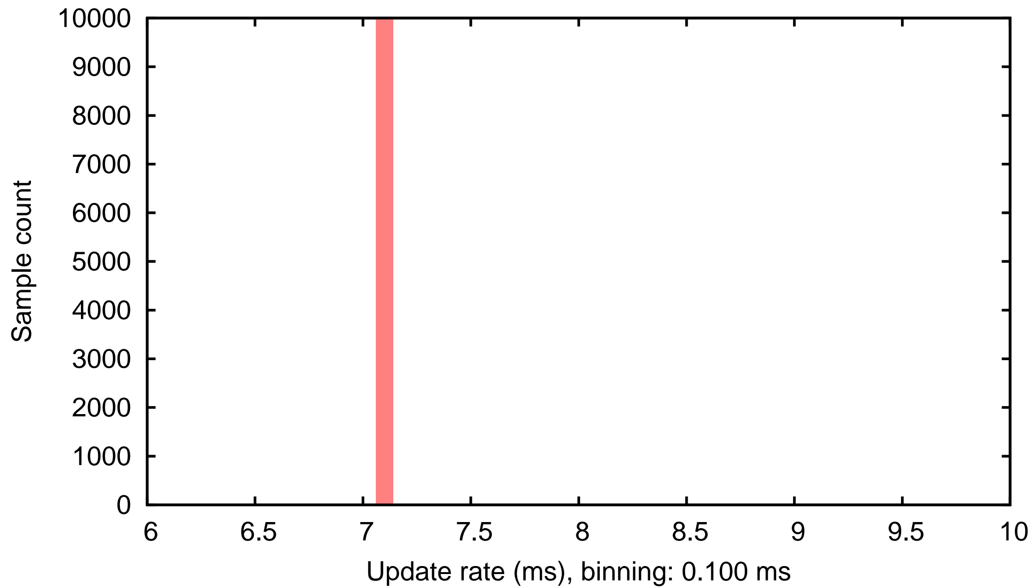


Figure 4.17: Update histogram of eight objects tracked with FPGA. The update-rate is between 7.19 ms and 7.20 ms with a mean value of 7.19 ms and a standard deviation below 0.01 ms (4.0 µs)

histogram is shown in figure 4.17. The deviation for one object is not different from the behavior with eight objects.

In the deviation shown in figure 4.16, it can be observed that the updates form lines with periodic distances in two clusters. These are likely to be caused by the UART transmission. The two clusters are approx. 9 µs apart. This distance is the transfer time of one bit for a UART speed of 155.200 Baud. The used UART sender is clocked with the UART frequency and can therefore only start a transmission every 8.7 µs. The lines that are formed around the clusters are likely to be caused by the UART receiver. A detailed analysis of the behavior is given in section 4.4, when the latency behavior is analyzed.

Table 4.1 shows the characteristics of the different measurements.

First conclusions can be drawn from the update-rate measurements. Firstly, when there is no additional load on a processor, neither the number of objects nor the use of a real-time extension have an impact on the update-rate behavior. Secondly, an additional processor load leads to the loss of updates and a large deviation, and the image content has an impact on the deviation. Thirdly, a real-time operation system can be used to tackle the loss of updates and large outliers, but still has a larger distribution than without the additional load. Lastly, FPGA-based image processing is one magnitude better than PC-based image processing without an artificial load concerning update-rate deviation. It is two magnitudes better than

Method*	#objects	min (ms)	max (ms)	std dev (µs)	updates lost
PC	1	6.12	8.24	77	no
PC	8	6.03	8.44	88	no
PC-load	1	2.35	17.34	840	yes
PC-load	8	2.45	28.65	1,321	yes
PC-RT	1	5.94	8.62	78	no
PC-RT	8	5.89	8.53	83	no
PC-RT-load	1	5.91	8.45	501	no
PC-RT-load	8	6.03	8.26	178	no
FPGA	1	7.186	7.199	4	no
FPGA	8	7.186	7.200	4	no

Table 4.1: Overview of update-rate deviations.
* RT = real-time extension. load = full processor load

image processing on PCs with an additional load, regardless of the usage of a real-time extension. Furthermore, the jitter of FPGA-based image processing is mainly caused by the UART transmission. While the FPGA optimizes the jitter of the update-rate, it has additional benefits when looking at the latency.

4.4 Latency and latency jitter

The latency is the time from the occurrence of an event until the point in time when the event has been processed by the sensor and is transmitted to the control system.

To measure the latency, the measuring system first disables all LEDs. When the attached image-processing sends a notification about 0 found regions, one or more LEDs are powered and the timer is started. The LED power-on is synchronized with opening of the camera shutter. The timer is stopped when the correct number of objects is transmitted by the image-processing system. Then the LEDs are disabled again, and the process is repeated. The time between the opening of the shutter and the end of the image transfer is approx. 10.14 ms. Like for the update-rate test, two different measurements were carried out, both with and without an artificial workload on the PC. Every measuring technique was used for images with one object and for images with eight objects.

figure 4.18 shows the results of the latency test for PC-based image-processing for one LED. The latency is between 13.35 ms and 14 ms, with most of the updates clustering around 13.54 ms with a standard deviation of 0.07 ms. The distribution of

the latency is shown in figure 4.19. The range of the distribution was chosen between 13 ms and 18 ms to ensure a better compatibility between different measurements. figure 4.20 shows the latency for eight enabled LEDs; figure 4.21 shows the distribution. The latency is between 13.52 ms and 14.2 ms with most of the updates clustering around 13.67 ms. The standard deviation is 0.06 ms.

When comparing the results of the test with one LED and eight LEDs it can be concluded that the number of LEDs has no significant impact on the distribution if the number is not altered. Furthermore, the conclusion that the number of objects in the image has an impact on the latency can be drawn. The mean latency increases by 0.22 ms. For both tests, the images were noiseless and with clearly defined objects and the object sizes were comparatively small. Therefore, the impact of the image contents on the latency should not be neglected.

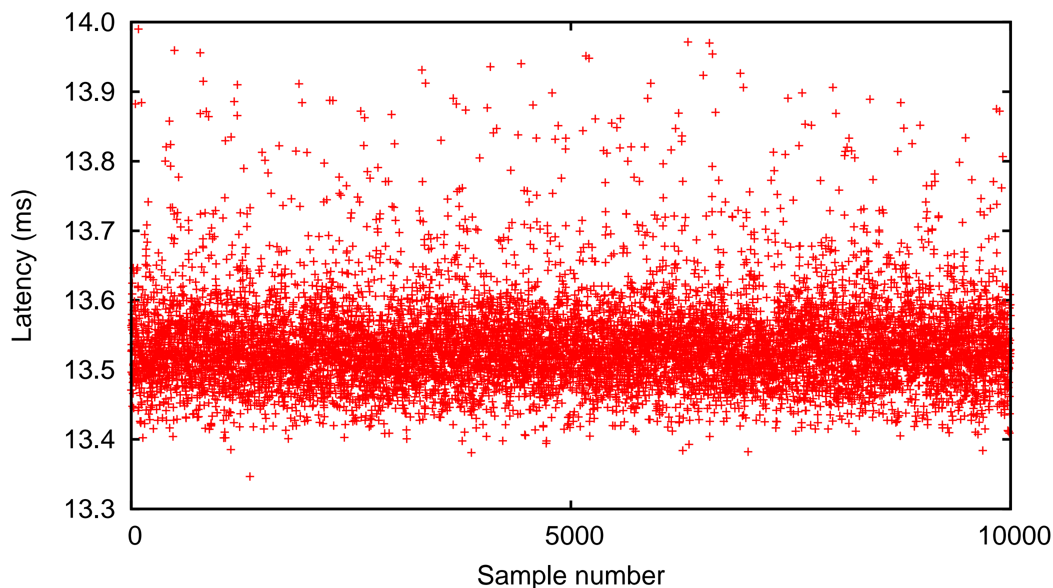


Figure 4.18: Latency distribution of one object. The latency is between 13.35 ms and 14 ms with a mean value of 13.54 ms and a standard deviation of 0.07 ms.

The distribution changes when an additional load is applied to the PC. As in previous tests, the load was applied using the command line tool stress. The latency for one object tracked with a full processor load is shown in figure 4.22. The latency ranges from 13.55 ms to 29.41 ms, has a mean value of 14.34 ms and a standard deviation of 0.9 ms. It can be seen in the image and the histogram image shown in figure 4.23, that still most of the updates cluster around the mean value but

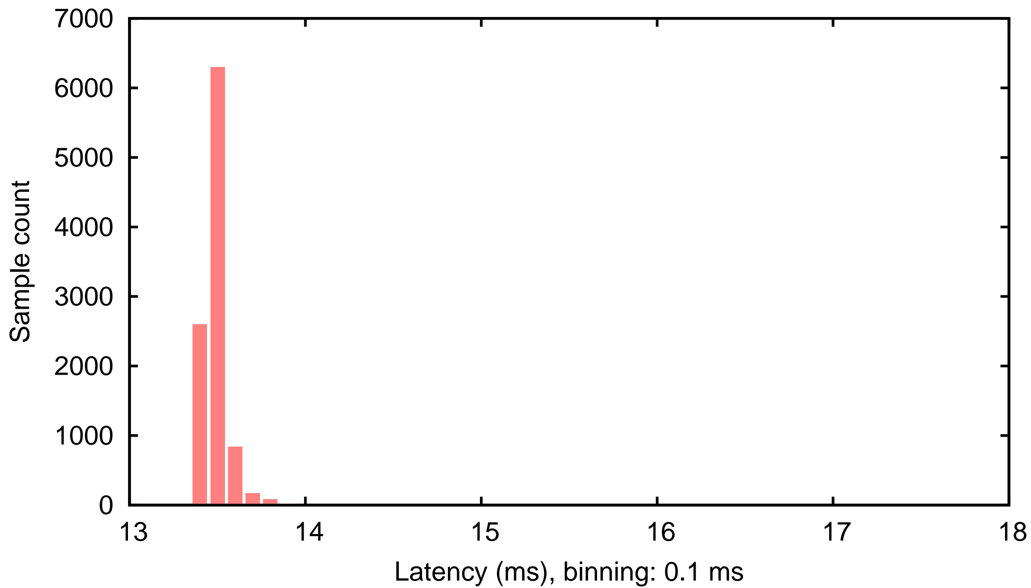


Figure 4.19: Latency histogram for one object. The latency is between 13.35 ms and 14 ms with a mean value of 13.54 ms and a standard deviation of 0.07 ms. The binning is 0.1 ms

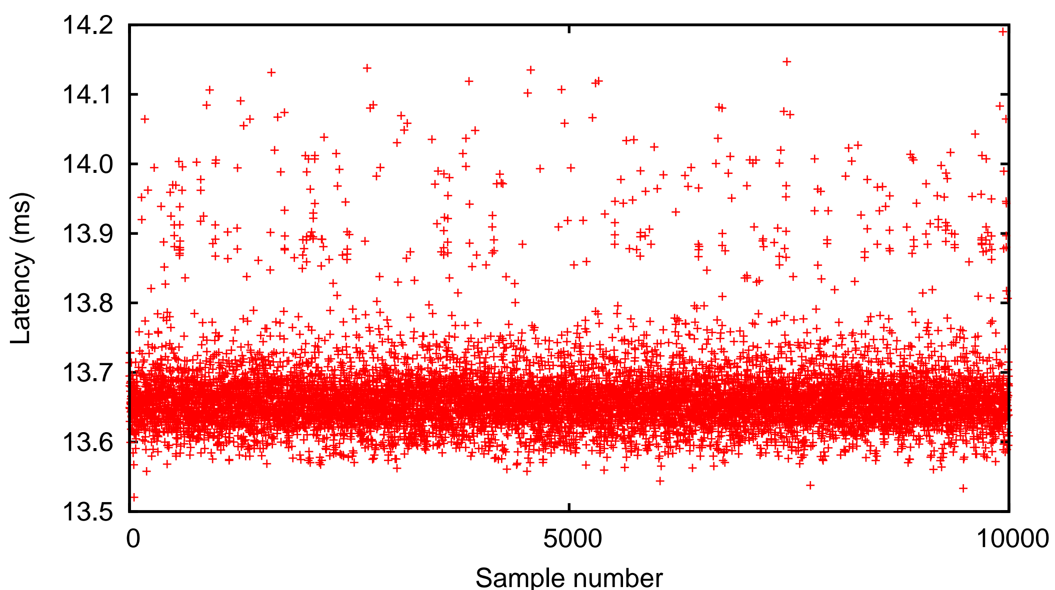


Figure 4.20: Latency distribution of eight object. The latency is between 13.52 ms and 14.2 ms with a mean value of 13.67 ms and a standard deviation of 0.06 ms.

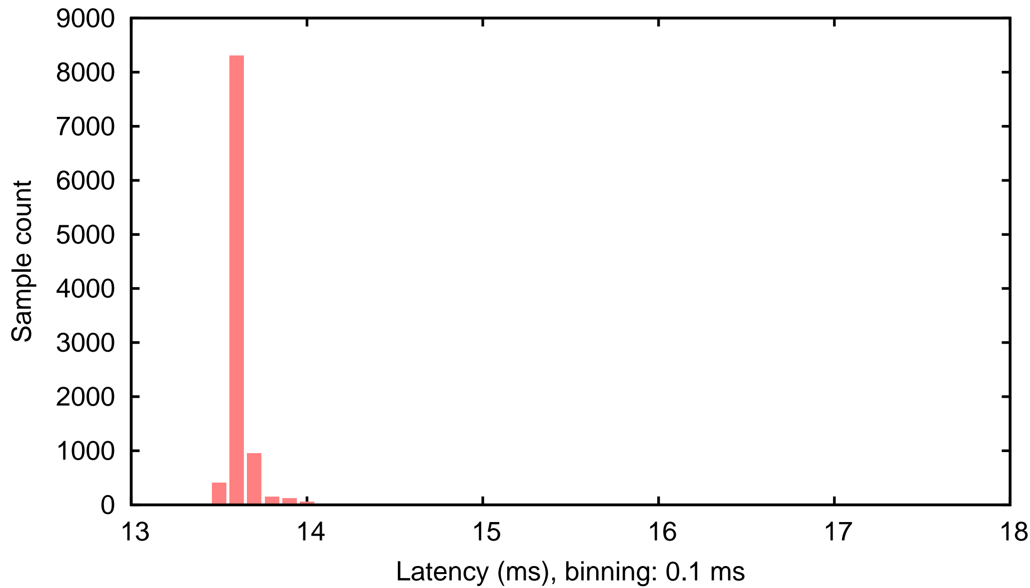


Figure 4.21: Latency histogram for eight objects. The latency is between 13.52 ms and 14.2 ms with a mean value of 13.67 ms and a standard deviation of 0.06 ms. The binning is 0.1 ms

with a much larger distribution even if the large outliers are ignored. Additionally, latencies up to 29 ms exist, which is more than four times the update-rate of the camera. This implies that camera updates are lost and not processed. The overall result of the tracking of eight regions with an additional load is comparable to the tracking of one region, but with a larger distribution and a larger standard deviation. The latency ranges from 13.63 ms to 37.99 ms. The mean value is 14.58 ms and the standard deviation is 1.3 ms. The distribution can be seen in figure 4.24 and the histogram in figure 4.25.

With a real-time extension, the distribution with the high CPU-load can be improved. The results of real-time image processing without an artificial load are comparable to the results without real-time extensions. For one object, the distribution ranges from 13.35 ms to 13.96 ms with a mean value of 13.52 ms and a standard deviation of 0.05 ms. For eight objects, the distribution ranged from 13.57 ms to 14.07 ms with a mean value of 13.72 ms and a standard deviation of 0.05 ms. With real-time image processing, the standard deviation is slightly smaller than without. The figures 4.26, 4.27, 4.28 and 4.29 show the deviations and the histograms of real-time image processing with one and eight regions, respectively.

If the processors of the system are under full load, the real-time extension improves the deviation significantly. The distribution and the deviation are larger than without load on the CPU, but without the large outliers that occur in image processing without a real-time extension. The distribution and the histogram can

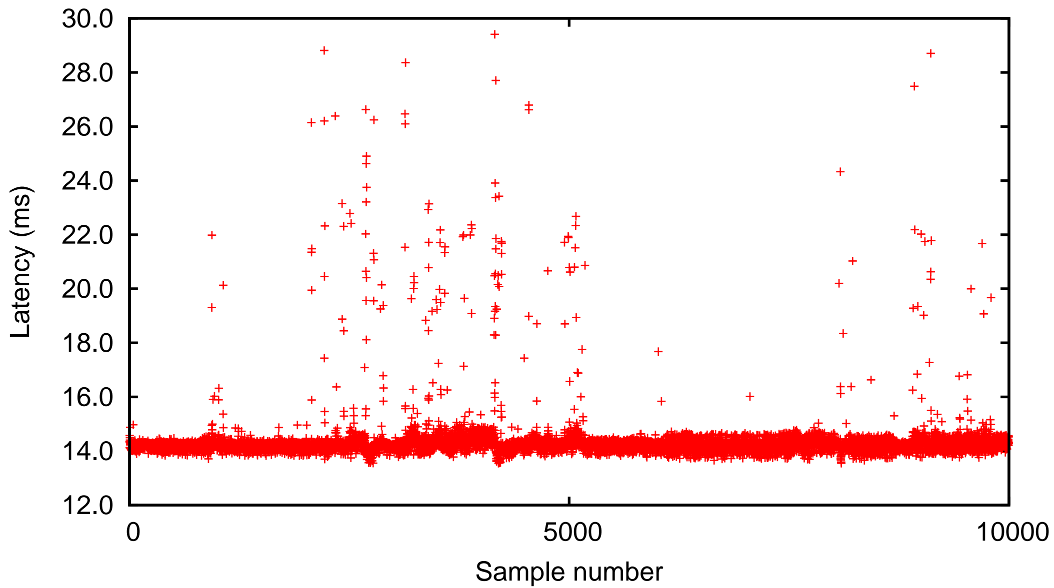


Figure 4.22: Latency distribution of one object with a full processor load. The latency is between 13.55 ms and 29.41 ms with a mean value of 14.34 ms and a standard deviation of 0.9 ms.

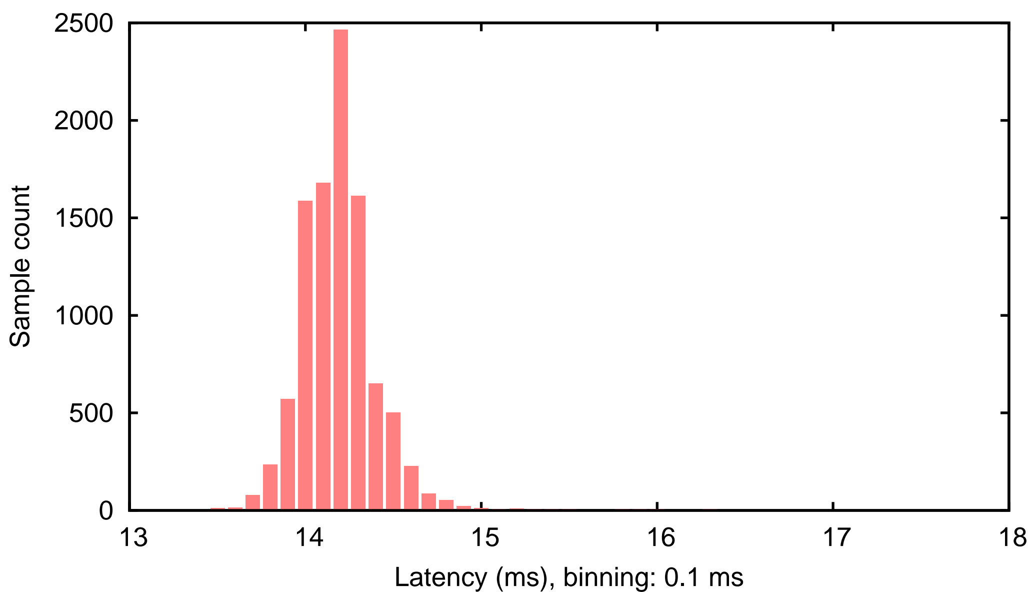


Figure 4.23: Latency histogram for one object with a full processor load. The latency is between 13.55 ms and 29.41 ms with a mean value of 14.34 ms and a standard deviation of 0.9 ms. The binning is 0.1 ms

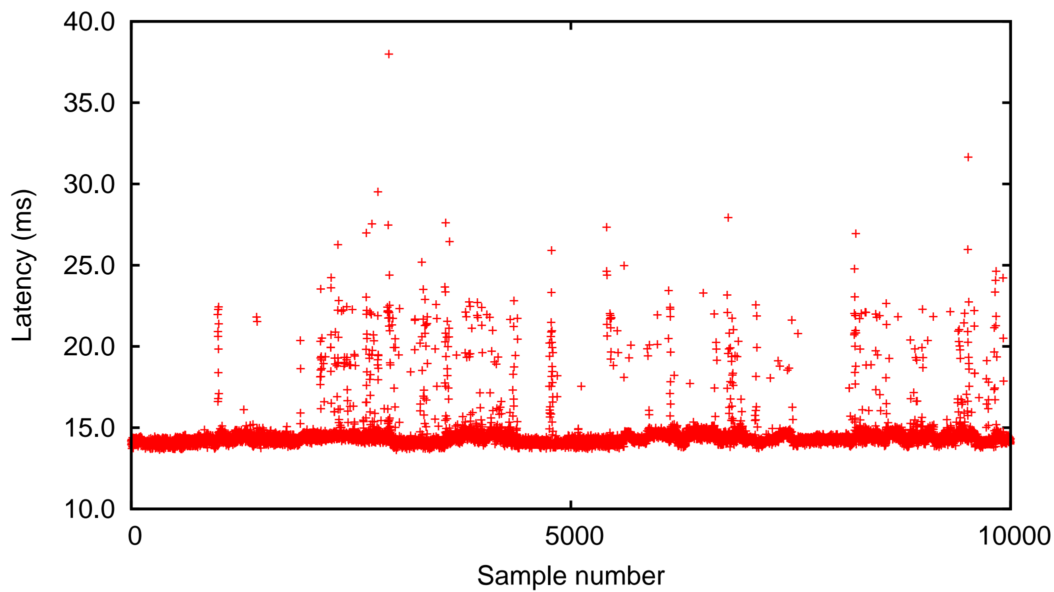


Figure 4.24: Latency distribution of eight objects with a full processor load. The latency is between 13.63 ms and 37.99 ms with a mean value of 14.58 ms and a standard deviation of 1.3 ms.

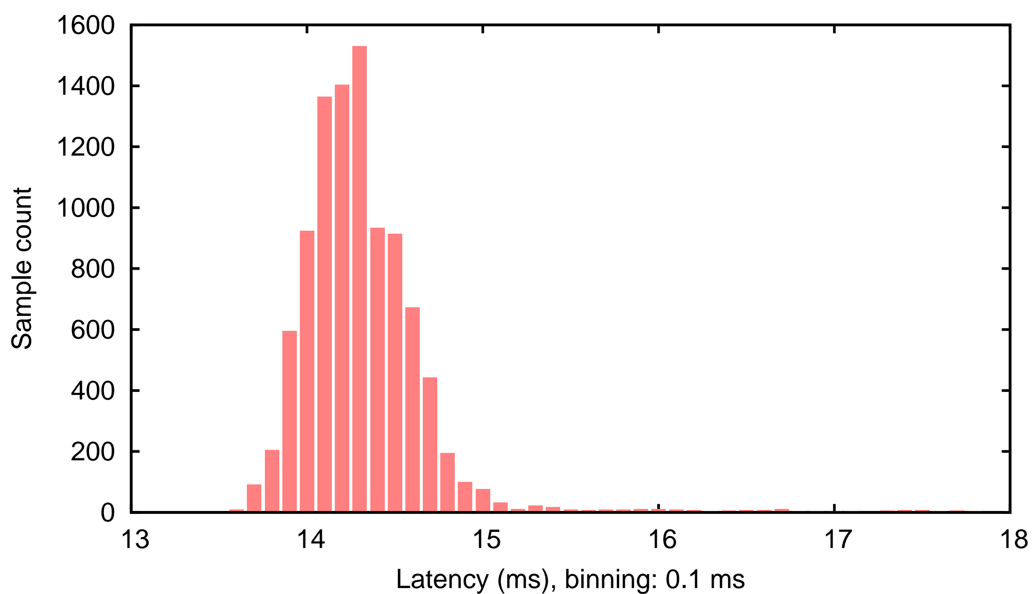


Figure 4.25: Latency histogram for eight objects with a full processor load. The latency is between 13.63 ms and 37.99 ms with a mean value of 14.58 ms and a standard deviation of 1.3 ms. The binning is 0.1 ms

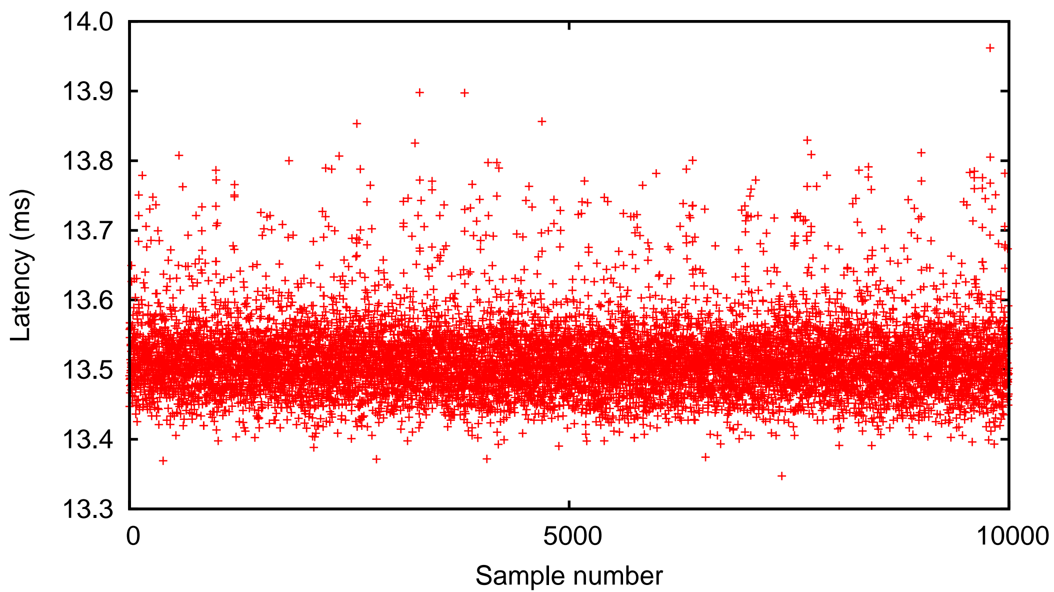


Figure 4.26: Latency distribution of one object with a real-time extension. The latency is between 13.35 ms and 13.96 ms with a mean value of 13.52 ms and a standard deviation of 0.05 ms.

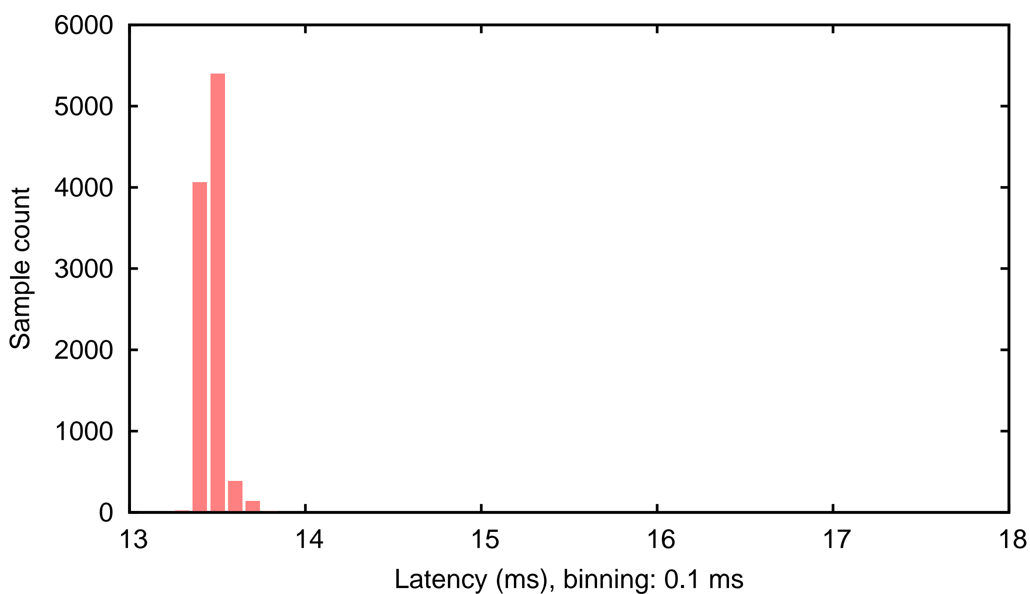


Figure 4.27: Latency histogram for one object with a real-time extension. The latency is between 13.35 ms and 13.96 ms with a mean value of 13.52 ms and a standard deviation of 0.05 ms. The binning is 0.1 ms

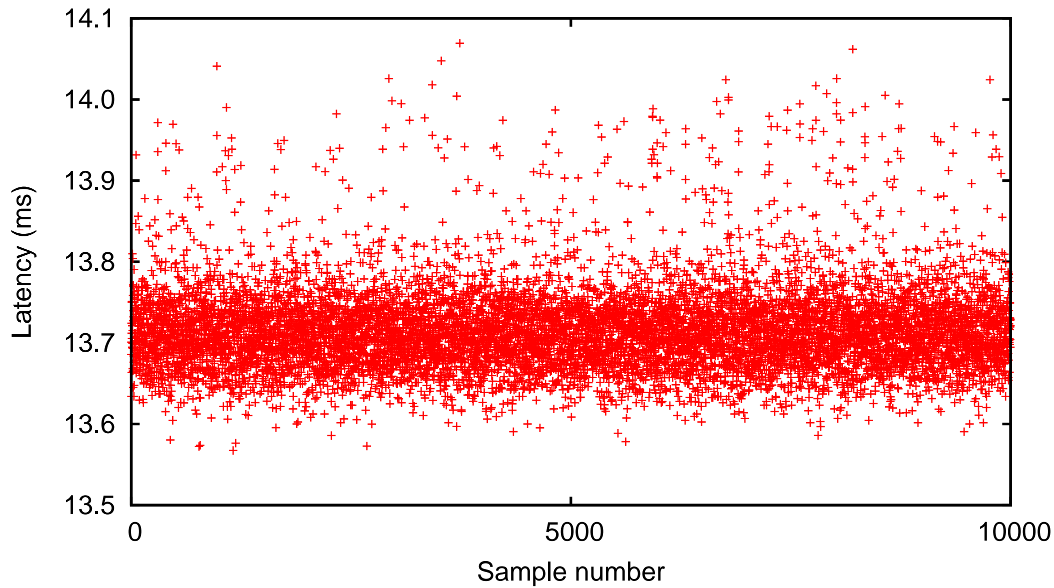


Figure 4.28: Latency distribution of eight object with a real-time extension. The latency is between 13.57 ms and 14.07 ms with a mean value of 13.72 ms and a standard deviation of 0.05 ms.

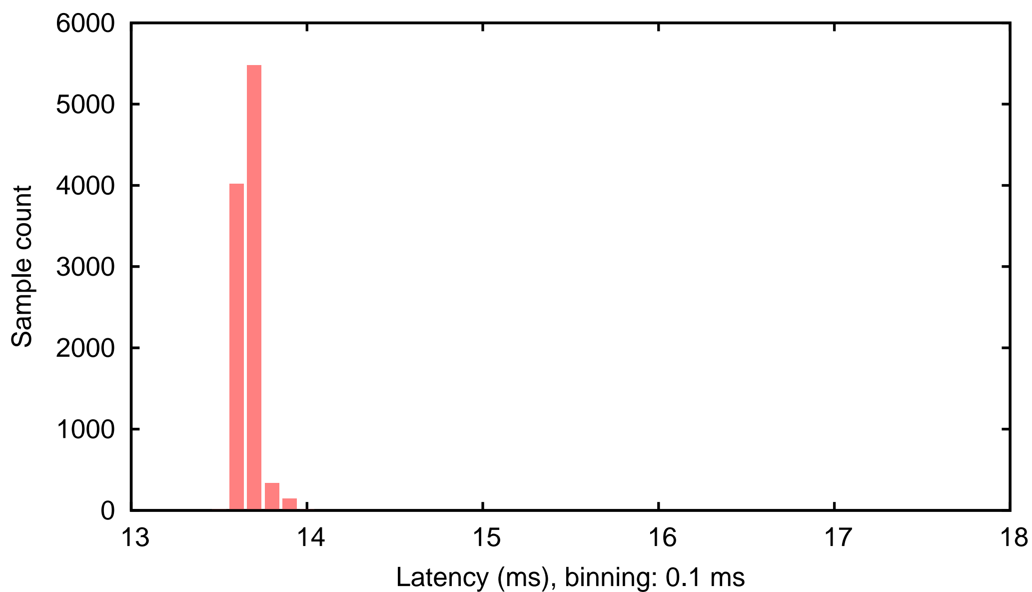


Figure 4.29: Latency histogram for eight objects with a real-time extension. The latency is between 13.57 ms and 14.07 ms with a mean value of 13.72 ms and a standard deviation of 0.05 ms. The binning is 0.1 ms

be seen in figure 4.30 and 4.31, respectively. For one object, the latency ranges from 13.41 ms to 15.11 ms with a mean value of 14 ms and a standard deviation of 0.2 ms. For eight objects, the latency ranges from 13.62 ms to 15.38 ms with a mean value of 14.28 ms and a standard deviation of 0.2 ms. The distribution and the histogram can be seen in figure 4.32 and 4.33, respectively.

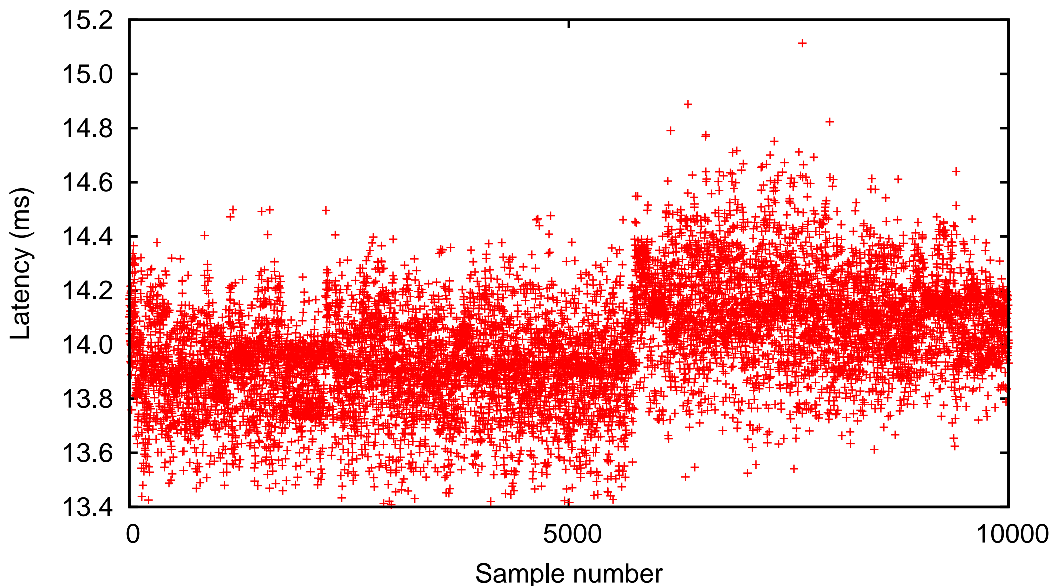


Figure 4.30: Latency distribution of one object with a real-time extension and a full processor load. The latency is between 13.41 ms and 15.11 ms with a mean value of 14 ms and a standard deviation of 0.2 ms.

FPGA-based object detection shows an outstanding behavior compared to PC-based image-processing. First of all, the latency is approx. 3 ms lower than with PC-based image-processing. The mean value of FPGA-based image-processing is 10.15 ms. This underlines the assumption that FPGA-based image-processing can operate faster than PC-based image-processing. This is due to the fact that FPGA-based image-processing is operating on the image stream instead of the full image. PC-based image-processing has to wait until the whole image is stored in the memory. Furthermore, each image-processing step is processing the whole image and writing it back to the memory before the next image-processing step can start. It can be deduced from the measurements that PC-based image-processing needs at least 3.2 ms for processing.

Furthermore, the distribution is in the range of 17 µs compared to 500 µs to 800 µs for PC-based image-processing under optimal circumstances. The better behavior

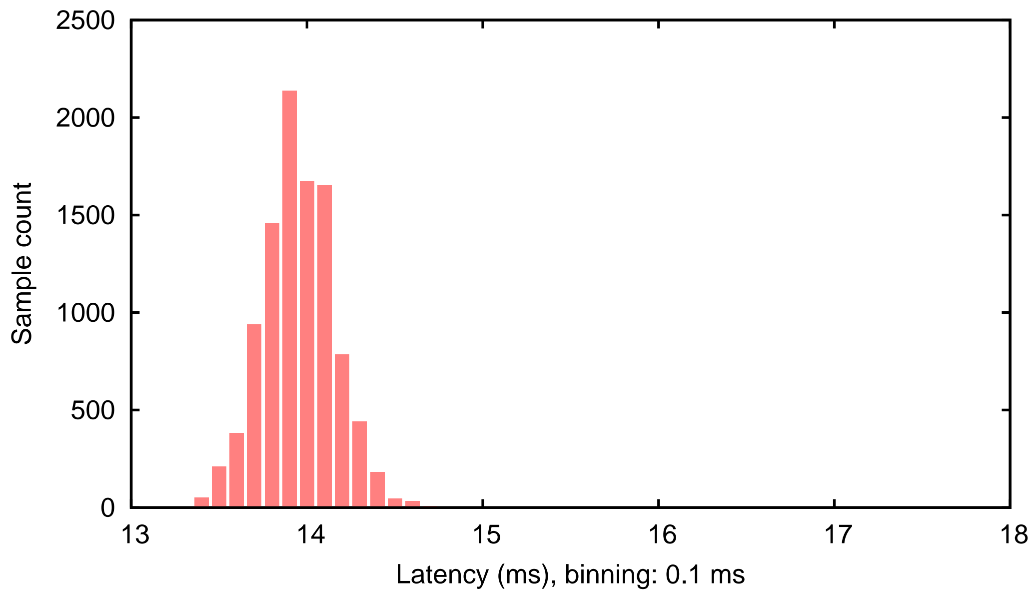


Figure 4.31: Latency histogram for one object with a real-time extension and a full processor load. The latency is between 13.41 ms and 15.11 ms with a mean value of 14 ms and a standard deviation of 0.2 ms. The binning is 0.1 ms

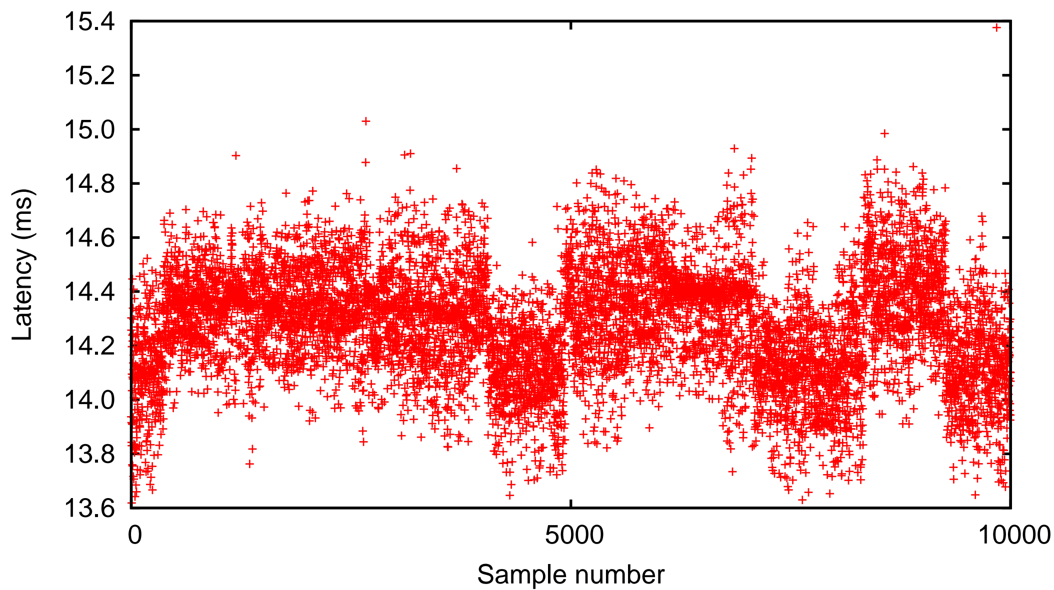


Figure 4.32: Latency distribution of eight objects with a real-time extension and a full processor load. The latency is between 13.62 ms and 15.38 ms with a mean value of 14.28 ms and a standard deviation of 0.2 ms.

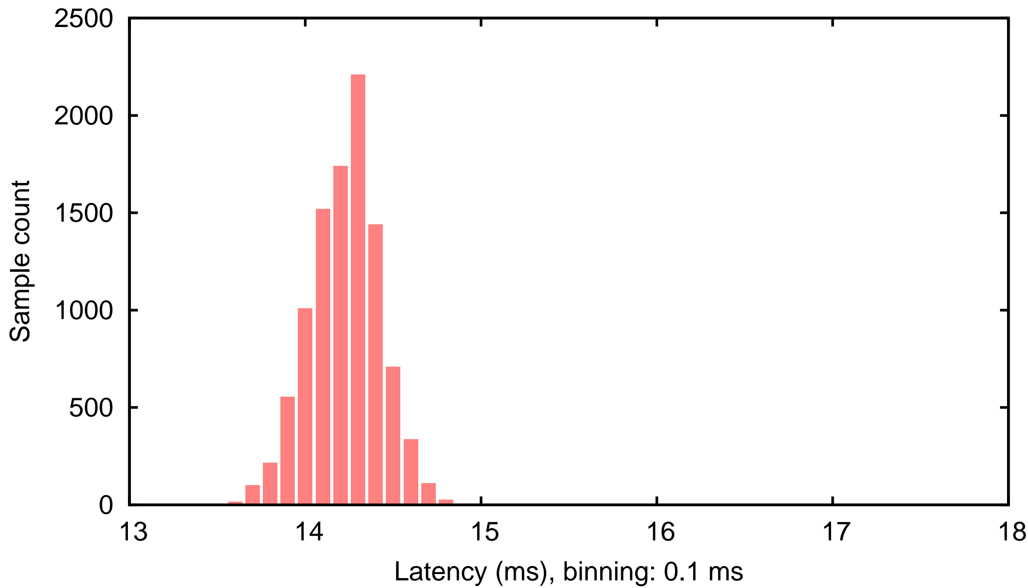


Figure 4.33: Latency histogram for eight objects with a real-time extension and a full processor load. The latency is between 13.62 ms and 15.38 ms with a mean value of 14.28 ms and a standard deviation of 0.2 ms. The binning is 0.1 ms

in distribution is underlined by the standard deviation of 7.7 µs compared to 50 µs for PC-based image processing with a real-time extension and 60 to 70 µs without.

Additionally the image content has no impact on the latency. The measured latencies differ only marginally in mean value and distribution. These differences are statistical noise and are not based on the image content.

Both latencies' measurements range from 10.1432 ms to 10.1459 ms. The latency for one object has a mean value of 10.1452 ms and standard deviation of 0.8 µs. The latency for eight objects has a mean value of 10.1453 ms and standard deviation of 0.8 µs.

The deviation and the histogram for one object are shown in figure 4.34 and figure 4.35, respectively. The deviation and the histogram for eight objects are shown in figure 4.36 and figure 4.37, respectively.

The deviation of both measurements show that the updates are forming lines with periodic distances. The distances are approx. 1.6 µs. This distance represents the sampling rate of the UART receiver of the measurement FPGA. The UART transfer rate was set to 115.200 Baud. At this frequency, the cycle time for one bit is approx. 8.7 µs. Most UART receivers have an oversampling of 8 or 16 times the baud rate. The sampling frequency for an UART with 16-bit oversampling is 0.54 µs. Three oversampling steps represent the distances between the update-rate

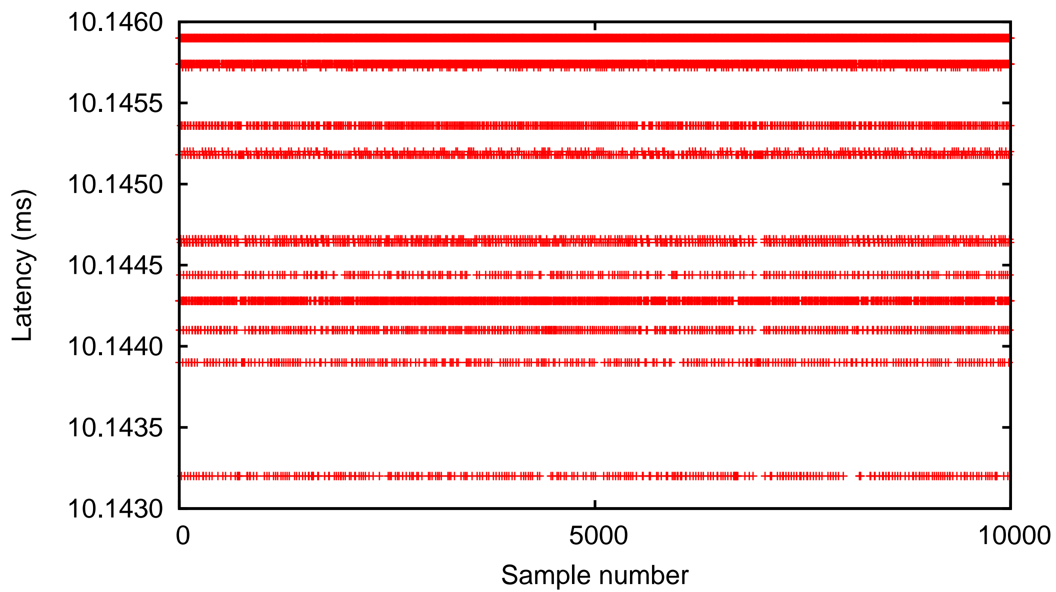


Figure 4.34: Latency distribution of one object tracked with FPGA.

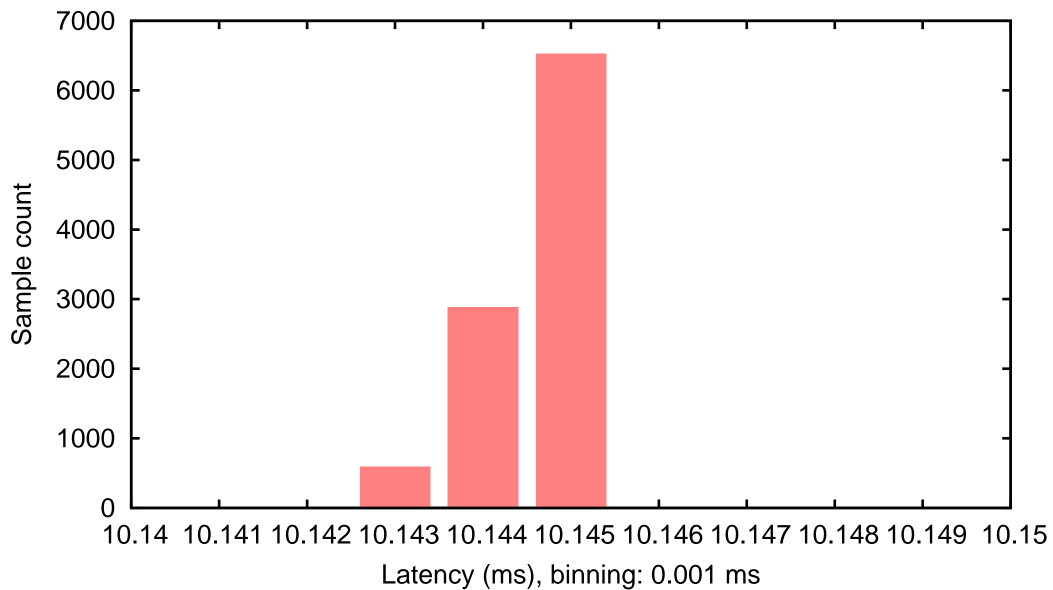


Figure 4.35: Latency histogram for one object tracked with FPGA.

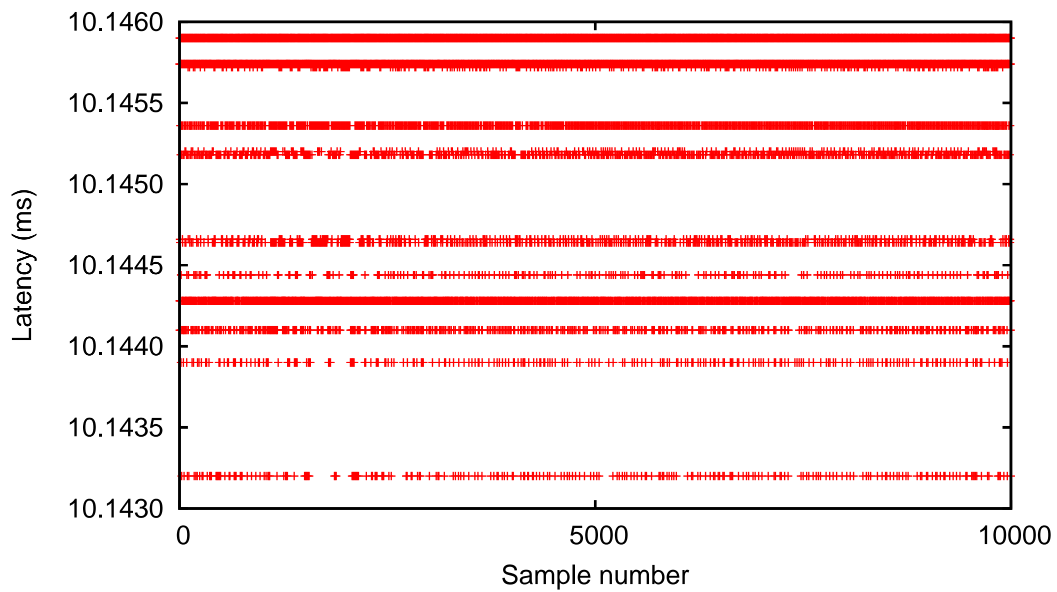


Figure 4.36: Latency distribution of eight objects tracked with FPGA.

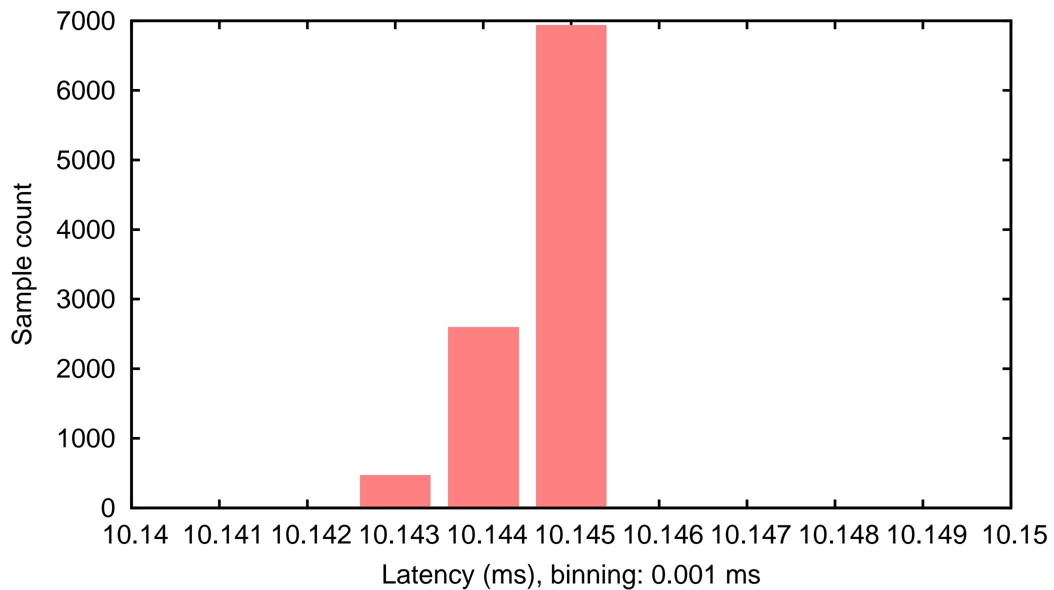


Figure 4.37: Latency histogram for eight objects tracked with FPGA.

clusters. The used UART receiver is a proprietary FPGA-IP-core provided by Xilinx. Therefore, no further analysis regarding the distances could be performed.

Method*	#objects	mean (ms)	min (ms)	max (ms)	std dev (µs)
PC	1	13.54	13.35	14.00	65
PC	8	13.67	13.52	14.19	62
PC-load	1	14.34	13.55	29.41	899
PC-load	8	14.58	13.63	37.99	1,298
PC-RT	1	13.52	13.35	13.96	53
PC-RT	8	13.72	13.57	14.07	53
PC-RT-load	1	14.00	13.41	15.11	202
PC-RT-load	8	14.28	13.62	15.38	200
FPGA	1	10.145	10.143	10.146	0.8
FPGA	8	10.145	10.143	10.146	0.8

Table 4.2: Overview of latency deviations.
* RT = real-time extension. load = full processor load

Table 4.2 shows the overview of all latency measurements.

In PC-based image-processing, the image content (e.g. noise or the number of objects) has an impact on the latency. Furthermore, as expected, the processor load has a significant impact on the latency; up to the point where several updates are lost and the latency has a length of multiple update-rates. A real-time extension can be used to eliminate the loss of updates and keep the latency in a usable window.

An FPGA implementation has a latency distribution that is two magnitudes better than any PC-based approach and three to four magnitudes better than PC-based image-processing with a high processor load. Furthermore, the latency of FPGA-based image-processing is 3.2 ms lower than the best latency achievable by a PC-based implementation. The latency of the FPGA-tracking is only a few µs slower than the acquisition and transfer time of the camera and the latency is independent of the image-content.

4.5 Conclusions

In this chapter, object-detection using the BLOB-extraction approach has been validated using software-based image processing and FPGA-based image processing. Software-based image processing has been tested with and without a real-time extension. The update-rate and the jitter has been recorded and analyzed for all systems.

From the update-rate measurements, several conclusions can be drawn.

Update rate

The workload of the PC has a significant impact on the update-rate deviation. With a high processor load, the update-rate becomes unstable up to the point where updates are lost because the processing takes too long. The image content (e.g. noise, number of objects) has no impact on the update-rate deviation if the CPU has a low workload. However, it has an impact if the processor has a high load. When tracking multiple objects, the tracking process has a slightly longer runtime. Therefore, it is interrupted more often by the scheduler to give CPU-time to other processes. This leads to larger outliers and a wider distribution. For the tests, noise-free, well-defined images were used. For noisy images, or images with more or larger objects, it is likely that the update-rate jitter will get worse.

The loss of updates and the wide update-rate distribution can be tackled by using a real-time operating system. However, the real-time extension offers no improvement to the update-rate for a low processor load. For a high processor load the real-time extension improves the update-rate significantly. Firstly, no updates are lost. Secondly, the deviation is within the same borders as with a low processor load. However, the distribution is wider; the standard deviation degrades from approx. 80 µs to approx. 200 µs. Reasons for the larger deviation with a high load can be interrupts by the system scheduler or fixed rescheduling points.

Lastly, FPGA-based image processing is one magnitude better than PC-based image processing without artificial load concerning update-rate deviation. It is two magnitudes better than image processing on PCs with an additional load, regardless of the usage of a real-time extension. Furthermore, the jitter of FPGA-based image processing is mainly caused by the UART transmission.

The overview of all update-rate measurements can be found in Table 4.1.

Latency

Regarding the deviation, similar conclusions can be drawn. The workload of the PC influences the deviation significantly up to a point where updates are lost and the latency degrades to several update cycles. With a high load, the image content impacts the deviation, for the same reasons as when it impacts the update-rate. Similar to the update-rate, a real-time extension can be used to improve the deviation for a high processor load. Similar to the update-rate, the real-time extension has no impact for a low processor load and a wider distribution for high loads. The standard deviation degrades from approx. 50 µs to approx. 200 µs.

The image content has an impact on the latency. For this measurement, the mean latency of all tests increased by between 150 µs and 280 µs when tracking eight objects instead of one. The objects were small and well-defined and the images noiseless. The latency will most likely increase for more complex images with more and larger objects and for noisy images.

An FPGA implementation has a latency distribution that is two magnitudes better than any PC-based approach and three magnitudes better than PC-based image-processing with a high processor load. Furthermore, the latency of FPGA-based image-processing is 3.2 ms lower than the best latency achievable by a PC-based implementation. The latency of FPGA tracking is only a few µs slower than the acquisition and transfer time of the camera and the latency is independent of the image-content.

The overview of all update-rate measurements can be found in Table 4.2.

A real-time extension eliminates the worst outliers of the distribution and keeps the update-rate and the latency within a usable window. However, the measurements above are only valid if the object detection process has the highest real-time priority and no other processes have the same priority. As image-processing PCs are often used to perform multiple image-processing tasks, this precondition may not be feasible.

The validation has shown that the FPGA-implementation is superior to PC-based approaches for every timing related aspect.

The latency and update-rate of FPGA-based image processing is independent from the image content. It can detect objects during the image acquisition, resulting in shorter latencies. The measured jitters are in the range of µs, and mainly caused by the chosen transportation channel.

The measurements in this chapter were performed with a real-time camera running in the Camera-Link base mode. In this mode, the camera sends images of 1000×1000 pixels with an update-rate of 139 Hz. Using the Camera-Link full protocol, the update-rate would be 371 Hz, resulting in an update-rate of 2.7 ms instead of 7.2 ms.

It can be deduced from the measurements that PC-based image-processing needs at least 3.2 ms for processing. Therefore, for higher update-rates, PC-based image processing is not feasible anymore.

5 Applications

The FPGA-based BLOB detection system can be used for a variety of different applications. The development of FPGA-based object detection was started to optimize the closed-loop performance of mobile micro-robots. However, due to its versatility it is not limited to motion tracking, but can also be used for classification.

Two applications are presented where the developed algorithm was deployed. The first application is the motion tracking for micro-robot closed-loop coarse positioning. For this application, the robustness and the predictability of the image-processing latency was the key to enhancing the closed-loop positioning time significantly.

The second application was a case study performed by a student group regarding traffic sign recognition. In this application, the latency of the overall process was of secondary importance. However, the goal of the developed system was that it should be cheap, small and have very low power requirements. for this reason, a traffic sign recognition system based on a low-cost FPGA and a low-cost camera chip was used. The fast and robust hardware implementation of the BLOB algorithm allowed real-time detection without using the embedded processor.

In the following sections, the two applications will be described in detail. The focus is on the FPGA-BLOB detection algorithm and its use for the targeted application.

5.1 Automated handling of microspheres

An early implementation of the BLOB detection algorithm was used for the coarse positioning of a mobile micro-robot. In the specific domain of micro- and nanohandling, the term coarse positioning refers to a positioning accuracy in the µm range.

The work on the closed-loop positioning of the mobile micro-robot was carried out by the author together with Dr. Daniel Jasper. While the author was responsible for the motion tracking system, Dr. Daniel Jasper developed the control system for the mobile microrobot. Work on this specific topic was published in a range of journals and conferences earlier (Diederichs, 2010; Jasper et al., 2010; Edeler et al., 2010; Diederichs, 2011; Fatikow et al., 2011; Jasper et al., 2011a,b).

5.1.1 Comparison of software-based and FPGA-based tracking

The closed-loop control was developed for a mobile nanohandling robot. The nanohandling robots perform a step-wise motion based on the stick-slip actuation principle. Figure 5.1 shows a bottom up view of the actuation unit based on laser-structured piezoceramic actuators (Edeler et al., 2008; Jasper and Edeler, 2008). Three active piezo segments can rotate a ruby hemisphere around its center point (figure 5.1 left). A single actuation unit (figure 5.1 middle) is comprised of three such ruby hemispheres. These hemispheres in turn can rotate a steel sphere using a friction contact. Three such steel sphere actuators are embedded into a printed circuit board (PCB) and carry the mobile robot (figure 5.1 right).

The mobile robots have two infrared LEDs at the bottom. The robots move on a glass plate. A camera with an infrared filter is mounted underneath the glass plate. The camera parameters are modified to produce black pictures with bright regions at the LED positions (see figure 5.2). The old tracking system uses a Videology USB-camera with VGA resolution (640×480 pixel) and an update-rate of 25 Hz. The image is transferred to a computer vision library running on a standard PC. The found position is transferred to the mobile robot via USB (see figure 5.3).

Before the FPGA-based system was deployed, the robots were tracked by a software-based approach (Dahmen et al., 2008). The approach calculates the weighted center of gravity using a region growing approach of the binary-large object detection algorithm.

For the timing evaluation, the updates were recorded at the closed-loop controller. Figure 5.4 shows the update-rate deviation. The average update-rate was 40 ms, as expected. The update-rate jitter has a standard deviation of 1.74 ms (4.4 % of the average update-rate) and a min-to-max jitter of 17.2 ms.

The deviation is larger than the deviations recorded in the measurements described in chapter 4. The reason for this is the complicated software architecture that was used for object tracking. The software architecture was based on the middleware

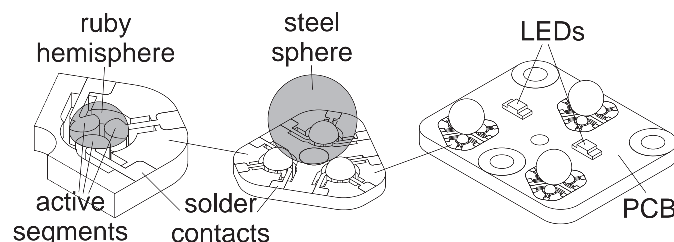


Figure 5.1: Mobile robot actuation unit (Jasper et al., 2010).

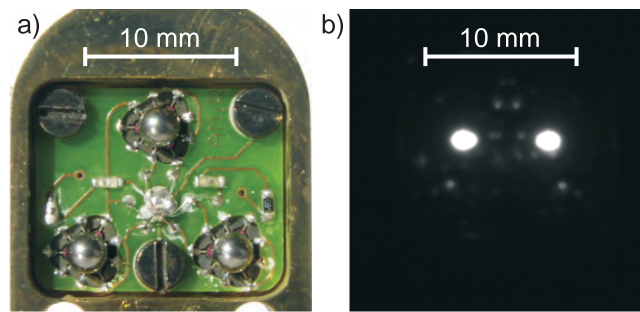


Figure 5.2: a) Bottom view of the mobile robot. b) Image taken by the tracking camera with an infrared filter.(Diederichs, 2010)

CORBA[1], resulting in several threads, processes and communications during the tracking process. Furthermore, the system was a general purpose image-processing system, performing several image processing tasks at once. Therefore, the conditions for object detection where not optimal.

To record the sensor's latency, the robot performed linear movements with constant speed and duration. The sensor position updates were recorded by the robot controller. Each movement started with a sensor update. After each movement the robot paused for ten sensor updates. The robot moved forward 100 times forward and then backwards 100 times. Figure 5.5 shows a section of the recorded movement. The average latency is 47.5 ms, composed of a full sensor update and additional computation and transfer time. The latency has a standard deviation of 2.3 ms and a min-to-max jitter of 14.7 ms.

To optimize the jitter, an FPGA-based system was used. The system featured a BLOB-detection algorithm that computed the weighted center of gravity.

The new tracking system aimed at minimizing the sensor performance issues described above. Firstly, a different camera with higher a update-rate (58 Hz)

[1] Common Object Request Broker Architecture, `http://www.corba.org/`. Last access: October 15, 2014.

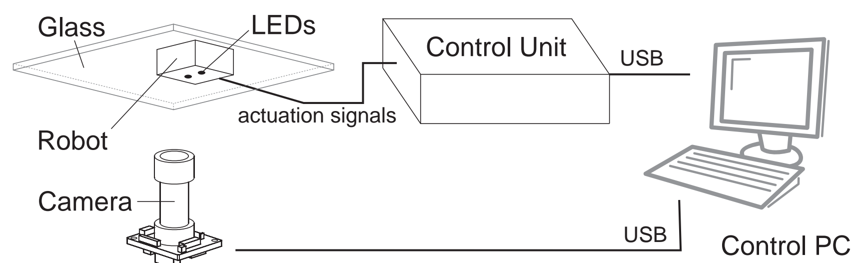


Figure 5.3: Architecture of the tracking system before the FPGA deployment. (Diederichs, 2010)

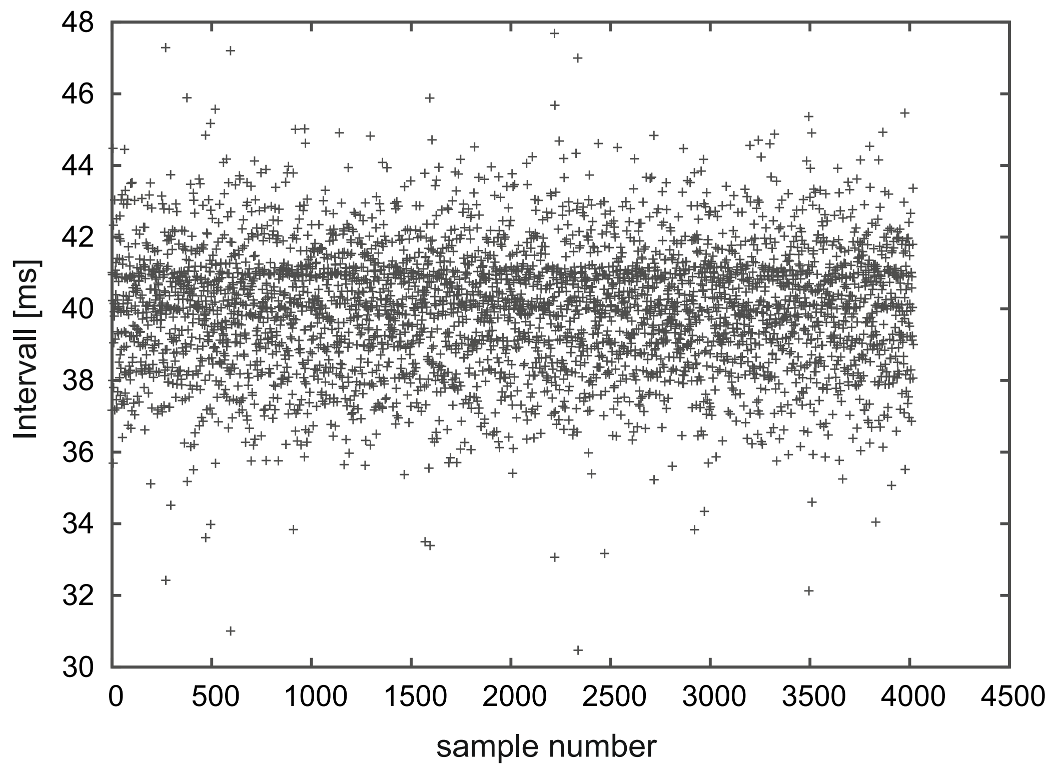


Figure 5.4: Update rate deviation of software-based robot tracking. (Diederichs, 2010)

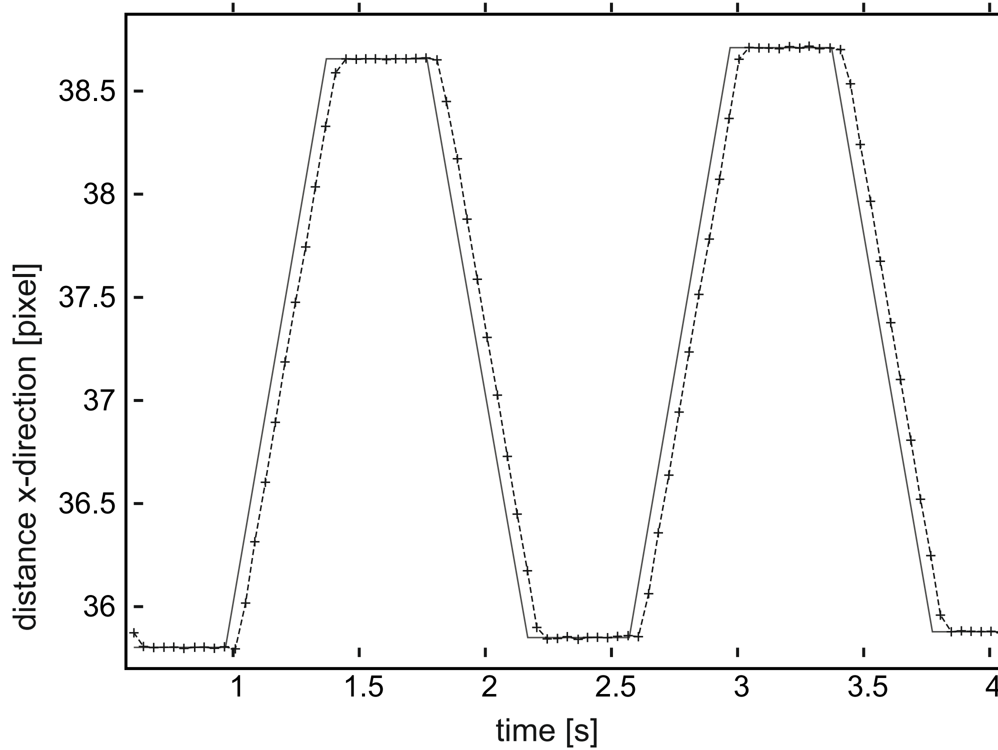


Figure 5.5: Latency of software-based robot tracking. (Diederichs, 2010)

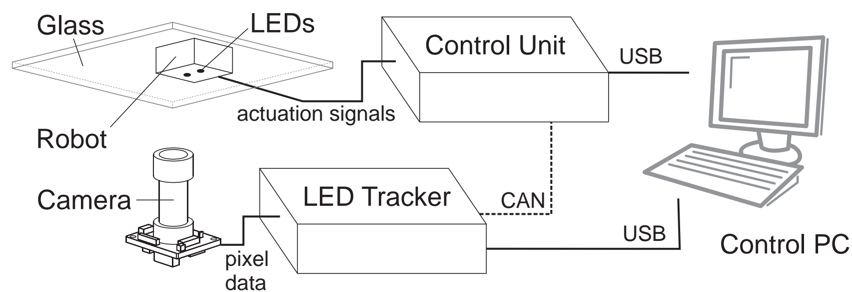


Figure 5.6: Architecture of the tracking system using an FPGA (Diederichs, 2010)

was used. This camera can be controlled via the IIC bus to only scan a region of interest (ROI) which leads to a higher update-rate (up to 200 Hz). Secondly, the position calculation was performed on a dedicated FPGA. While most of the region detection was performed in hardware, the final post processing steps were executed on an embedded processor. The embedded processor used an interrupt-driven architecture without an operating system and is therefore predictable in runtime and jitter.

The found robot positions are transferred to the mobile robot controller using the controller area network (CAN) bus (ISO/IEC, 2003). The CAN bus is a real-time capable bus for multiple controllers (Tindell et al., 1995).

With this architecture, the PC with its unpredictable behavior is omitted from the control loop. However, all signals, as well as the acquired image, are additionally transferred to the PC via USB. The extended architecture is shown in figure 5.6.

The tracking system was implemented on a XILINX Spartan3E FPGA (XC3S1200E). A softcore embedded processor (Microblaze) with a clock frequency of 50 MHz was used.

To evaluate the performance of the new sensor system, the same measurements as for the software system were performed. Initially, the calculated positions were transferred via USB to a PC and then transferred to the mobile robot controller. The average update-rate and latency improved because of the new camera, but in both cases the jitter did not. The update-rate has a standard deviation of 0.84 ms (4.5% of the average) and a min-to-max jitter of 20.25 ms.

The results of the CAN transfer show very good results regarding latency and jitter. Using the CAN bus, the update-rate is at a constant value of 18.81 ms with a negligible standard deviation of 1.4 μs (less the 0.01% of the average) and a min-to-max jitter of 9.4 μs. The superior performance regarding update-rate deviation is shown in figure 5.7.

Concerning latency, the CAN communication is at average 1.2 ms faster than the USB transfer and has a negligible jitter. The jitter has a standard deviation of 1.4 μs (less the 0.01% of the average) and a min-to-max jitter of 9.4 μs. The latency is constant with the value of one update-rate.

The system speed was increased further by selecting a region of interest with the size of 752x150 px, which increased the update-rate to 145 Hz. At this speed, the system was performing with the same stable jitter and latency as for full resolution.

The system was further extended to track multiple robots at the same time (Diederichs, 2011).

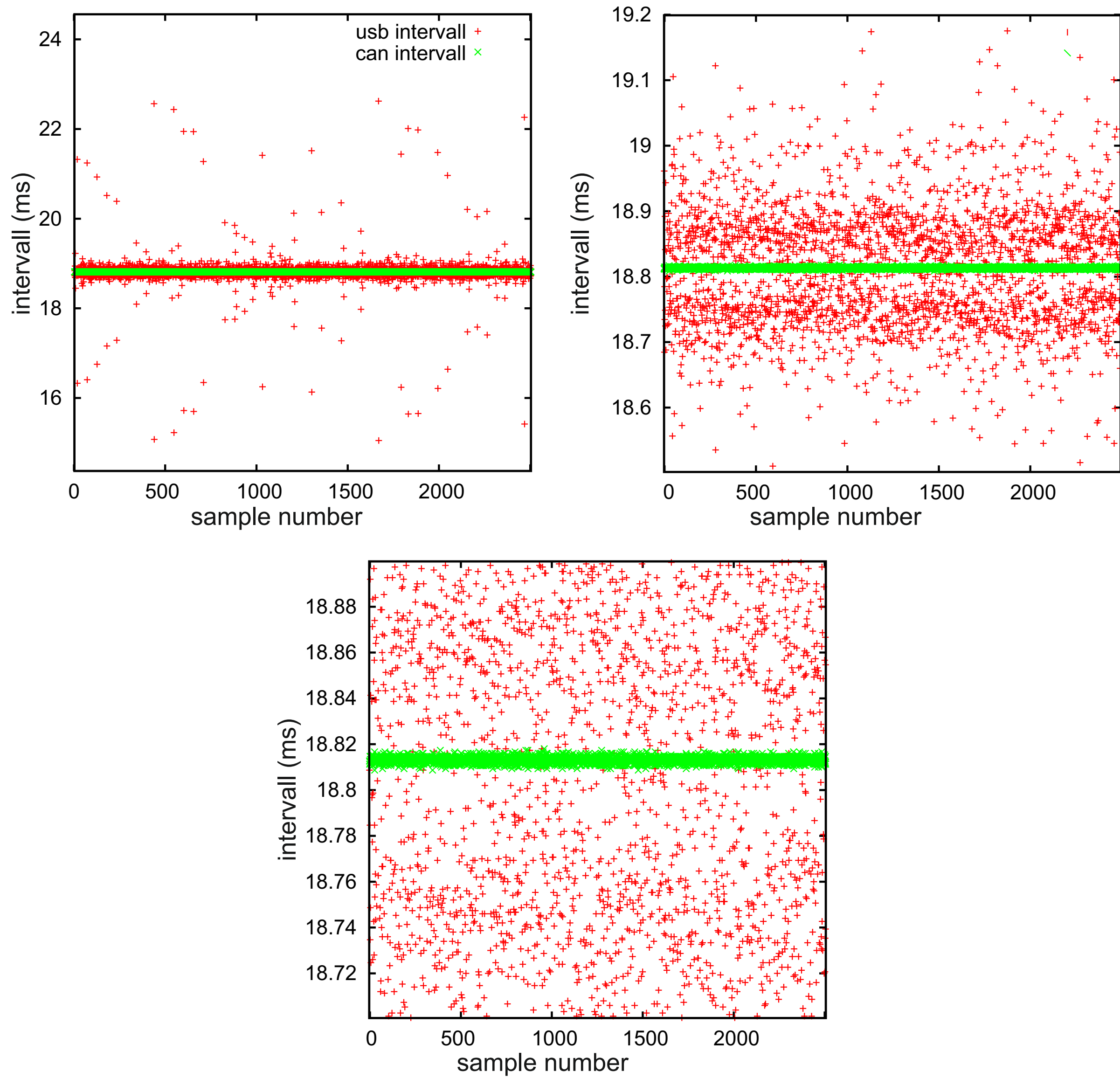


Figure 5.7: Update intervals of hardware-based led tracking with CAN and PC-routed USB updates (Diederichs, 2010).

5.1.2 Tracking of multiple robots

Tracking of multiple robots in a single picture can be done easily by grouping two regions r_1 and r_2 with the minimal distance with the following equations:

$$x(\text{robot}) = \frac{(x(r_1) + x(r_2))}{2} \quad (5.1)$$

$$y(\text{robot}) = \frac{(y(r_1) + y(r_2))}{2} \quad (5.2)$$

$$phi(\text{robot}) = \text{atan2}(y(r_2) - y(r_1), x(r_2) - x(r_1)); \quad (5.3)$$

figure 5.8 shows a visual representation of the position calculation.

However, capturing a full image results in low update-rates and long latencies. Additionally, using the simple approach described above, it is not guaranteed which robot is found first , which introduces additional jitter to the latency.

Robot tracking is executed on the embedded processor inside an interrupt service routine triggered by the region finder hardware component. It can operate in two different states. In the initialization state, the full image is captured with a low update-rate and all regions are mapped to robot positions. The robot center positions are stored in a list and the program enters the fast tracking state.

Operating in the fast tracking state, the program uses a single ROI that covers the regions of one robot (see figure 5.9). The position of the ROI is changed at each interrupt to the last known position of the next robot in the list.

For each robot, a different CAN ID is used for the position update. The robots are able to find the correct CAN ID themselves. The robots can also determine the update-rate as well as the latency of the sensor update. The update-rate of the ROI f_{ROI} depends on the employed camera system as well as the needed ROI size. As a ROI of the same size is used for all robots in the system, the image update-rate f_{ROI} is constant. The position update-rate f_{robot} for each robot is dependent on the number of robots c_r recognized by the tracking system.

$$f_{robots} = \frac{f_{ROI}}{c_r} \quad (5.4)$$

Thus, the update-rate depends on the number of robots, whereas the sensor signal latency remains constant:

$$latency_{robots} = f_{ROI} \quad (5.5)$$

The size of the ROI has to reflect the robot movement speed as well the update-rate, to make sure that the robot's LEDs are inside the ROI at the next update. If the

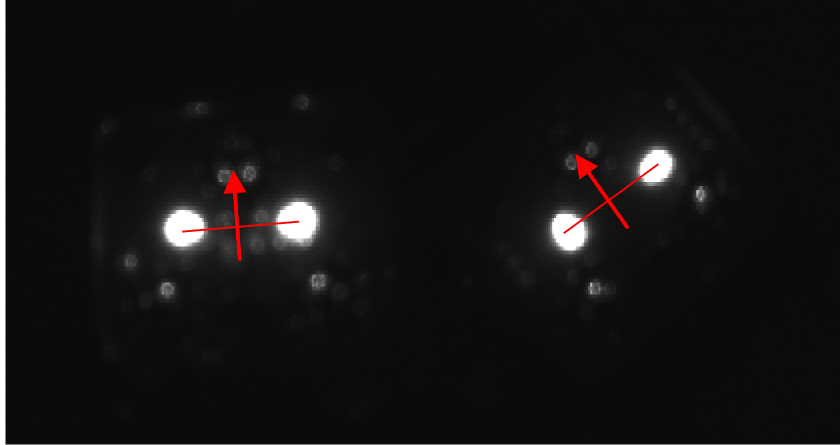


Figure 5.8: Mapping of two region centers to robot position and angle (Diederichs, 2011).

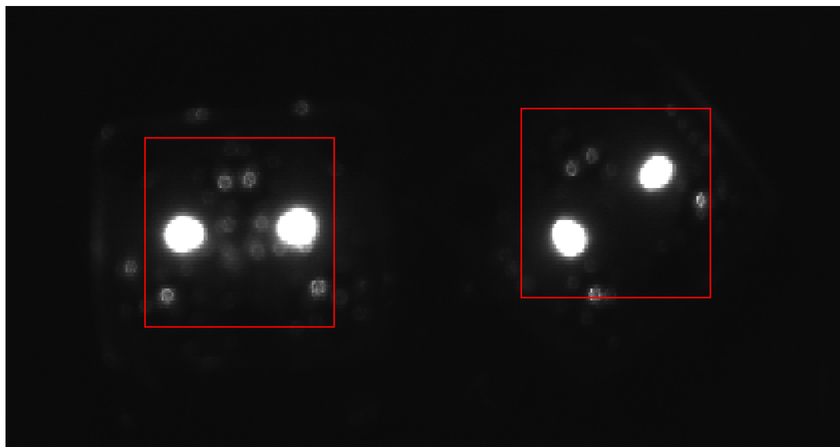


Figure 5.9: Robot tracking performed in alternating ROIs (Diederichs, 2011).

ROI is too small, a region could have moved out, so that the robot position is not trackable anymore or the results are incorrect.

If a robot is not found anymore, the initialization state is entered again. This can also be triggered by a special CAN message, e.g. if a robot is powered on and the tracking system is already inside fast tracking mode.

Aiming at high precision, a camera with high resolution and low physical pixel size was deployed. The camera used with the previous system had a resolution of 752×480 pixels and a physical pixel size of $6 \times 6\,\mu\text{m}$.

The new camera has a resolution of 2592×1944 pixels and a physical pixel size of $2.2 \times 2.2\,\mu\text{m}^2$. Because of the high resolution, the new camera has a low update-rate of about 10 Hz at full resolution. To obtain fast update-rates, tracking has to be performed inside a ROI.

The camera was mounted 80 mm beneath the glass surface (see figure 5.10) and uses a wide angle lens (focal length: 4 mm). With this configuration, the field of view has a physical size of $120 \times 80\,\text{mm}^2$.

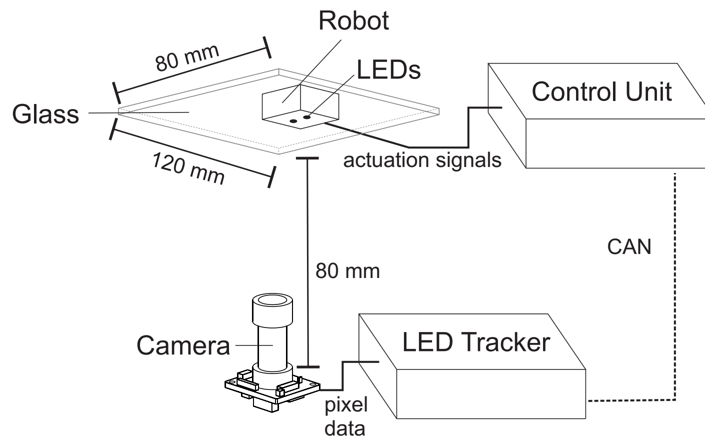


Figure 5.10: Architecture and dimensions of the multiple robot tracking system (Diederichs, 2011).

Using this camera, a ROI for fast tracking has the dimensions of 360×360 pixels, resulting an a ROI update-rate f_{ROI} of 205 Hz.

5.1.3 Distortion correction

As shown in figure 5.11a, the image is distorted by the lens. This leads to nonlinearity of the robot position, especially at the border regions of the image.

To reduce the nonlinearity, a barrel distortion is assumed and corrected. A full image distortion correction is not feasible for the speed the system is designed to perform. However, correcting the distortion of the region positions is a satisfactory approximation to reduce nonlinearity.

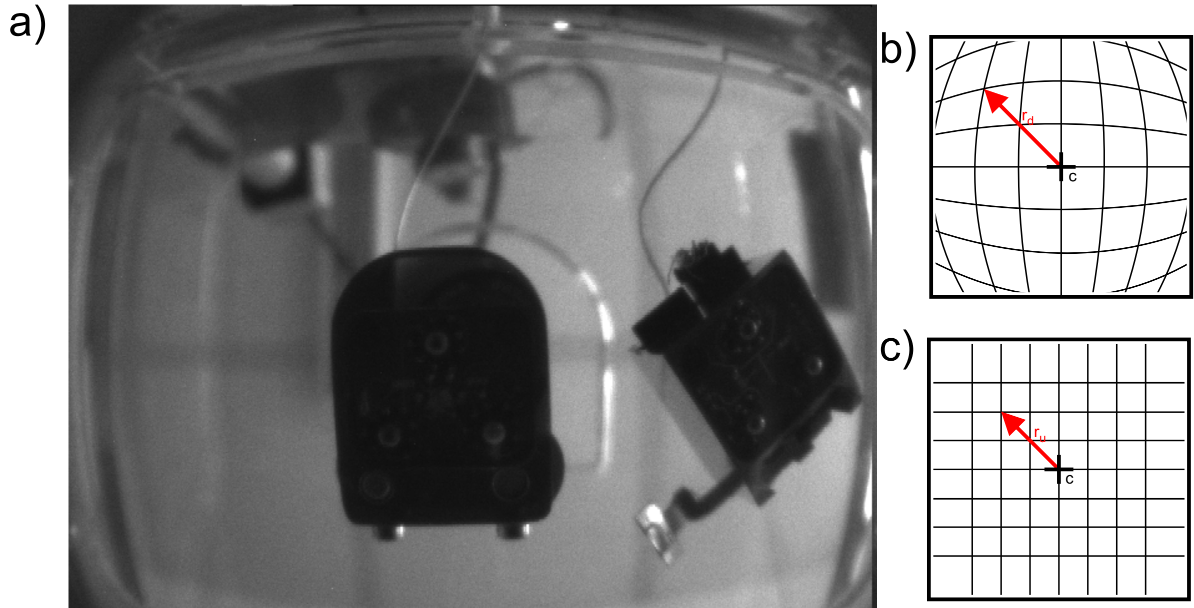


Figure 5.11: Distorted image taken by the camera beneath the glass surface (a) and example of barrel shape distorted (b) and undistorted (c) image. (Diederichs, 2011).

A simple barrel distortion model is (Bailey, 2002):

$$r_u = r_d(1 + r_d^2 k); \tag{5.6}$$

where r_u and r_d are the distances from the center of distortion ($\vec{c}$) in the undistorted and distorted images, respectively, as shown in figure 5.11b and c. k is the lens specific distortion parameter. Each pixel's coordinates $\vec{p_u}$ of the undistorted image can be given as a function to the coordinates of a pixel in the distorted image $\vec{p_d}$:

$$\vec{p_u} = (\vec{p_d} - \vec{c})\frac{r_u}{r_d} + \vec{c} \tag{5.7}$$

Using 5.6 with 5.7:

$$\vec{p_u} = (\vec{p_d} - \vec{c})(1 + kr_d^2) + \vec{c} \tag{5.8}$$

The lens specific distortion parameter k as well as the center of the lens c were computed offline. As the robot position does not need to have its center at the image's top-left pixel, adding $\vec{c}$ in equation 5.8 can be omitted.

$$\vec{p_u} = (\vec{p_d} - \vec{c})(1 + kr_d^2) \tag{5.9}$$

With this equation, the distortion correction for each region can be performed using six multiplications, two subtractions and three additions. These operations need approx. 12.5 µs on the embedded processor, which is negligible for a small number of regions.

5.1.4 Sensor accuracy evaluation

The weighted center of gravity result is highly dependent on the brightness of the image as well as the threshold used for the calculation. The linearity of this approach has not yet been analyzed. First, the impact of the brightness and the threshold on the signal noise is evaluated. For this experiment, a single robot was left stationary while the camera shutter time and the threshold were altered. For each combination 500 signal samples were recorded. The experiment was repeated at two different positions: at the lens center and at the border of the field of view. For each position the experiment was performed with two different angles. Figure 5.12 shows the visual representation of the signal noise standard deviation in the x-direction dependent on the shutter time and threshold for the center position.

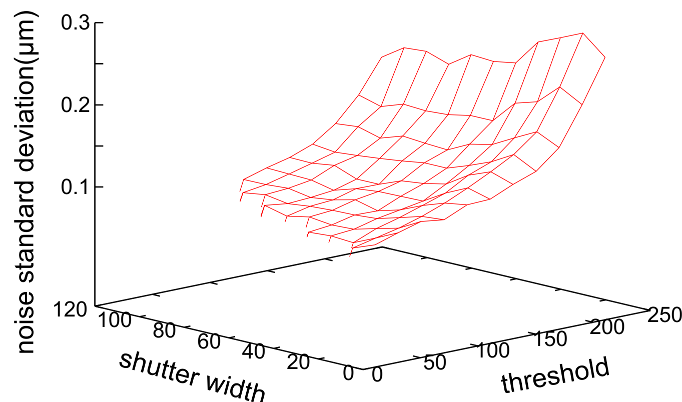


Figure 5.12: Noise standard deviation based on shutter time and threshold at the lens center. The shutter width is the number of lines used for the exposure time (Diederichs, 2011).

The experiment shows that the signal noise is lower for long shutter times and low thresholds. Both values must be carefully chosen to avoid that two regions are recognized as a single region. For the used camera, the shutter time affects the update-rate. As a high update-rate is desired, the shutter time should be chosen as low as possible to reach the desired accuracy.

The achievable noise standard deviation is below 100 nm. The min-to-max jitter (distance between the minimal position value and the maximum position value) is below 600 nm for the best configurations. The min-to-max jitter of different configurations is shown in figure 5.13.

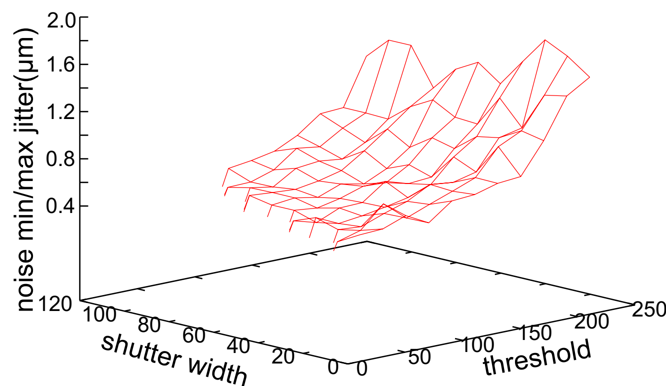


Figure 5.13: Noise minimum to maximum jitter based on shutter time and threshold at the lens center. The shutter width is the number of lines used for the exposure time (Diederichs, 2011).

The robot's angle has no impact on the results of the experiment. The results of the border position show a higher noise level, but the overall shapes are similar to figure 5.12 and figure 5.13. The best achievable standard deviation and min-to-max jitter at the border are 180 nm and 1.6 µm, respectively.

To examine the sensor's linearity, a measurement was conducted against a reference sensor. For this purpose, an optical microscope was mounted above the robot and a 50 µm glass sphere was tracked in the microscope's image. The maximum noise of the microscope-based tracking system is approx. 250 nm, which is sufficient as a reference to camera-based LED tracking. The mobile robot was then closed-loop positioned to a linear grid spanning 1 px of the LED tracking (see figure 5.14). Each position was approached and measured with the reference sensor. There is virtually no non-linearity and the deviations between the two sensors are caused by the noise.

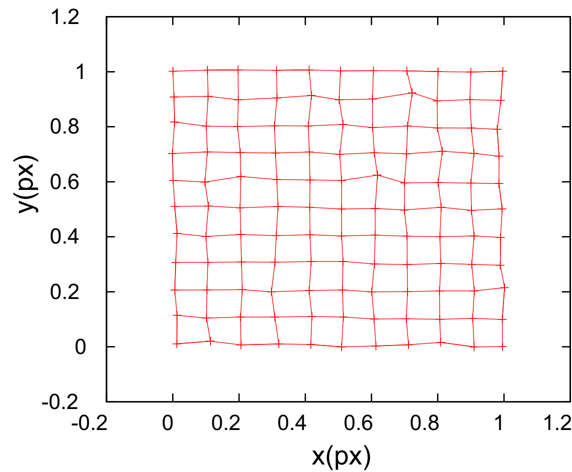


Figure 5.14: Sensor linearity of one square pixel. The mobile robot was then closed-loop positioned to a linear grid spanning 1 px of the LED tracking. Each position was approached and measured with the reference sensor. There is virtually no non-linearity and the deviations between the two sensors are caused by the noise (Diederichs, 2011).

5.1.5 Closed loop performance

Based on the excellent deterministic timing behavior of FPGA-based tracking, a special closed-loop controller was developed by Dr. Daniel Jasper (Jasper et al., 2010; Jasper, 2011).

The hardware-based tracker delivers the pose information with negligible jitter and latency. However, the camera itself delays the sensor update by one frame, because the image is captured (pixels exposed) only for a few µs before the transfer of the pixel data takes most of the time of the update interval. Thus, a sensor update actually contains the robot's pose information at the time of the previous update. Due to its low mass and comparatively low maximum speed, the robot can reach all velocities in less than 1 ms. Thus, its response time is negligible.

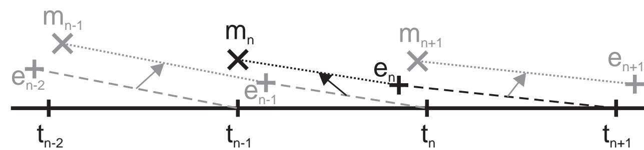


Figure 5.15: Measured poses m and estimated poses e while moving along a trajectory t (Jasper et al., 2010).

The control algorithm compensates for this delay as shown in figure 5.15. At each iteration, the measured pose $\mathbf{m}_n$ is received, which is outdated. The algorithm then calculates the robot's estimated pose $\mathbf{e}_n$ by adding the movement that was calculated during the previous iteration. With $\mathbf{e}_n$, two control deviations can be calculated. First, the distance between $\mathbf{e}_n$ and the following trajectory position $\mathbf{t}_{n+1}$ is the movement that the robot should execute in order to remain on the trajectory. Directly using the open-loop control, this represents a proportional controller part. Additionally, the distance between $\mathbf{e}_n$ and $\mathbf{t}_n$ can be used as it represents movement errors of the open-loop control. The integral of this distance is added to the control values forming an integral controller part. This leads to the PI-controller with the control parameters K_P and K_I:

$$\mathbf{s} = K_P\left(\mathbf{t}_{n+1} - \mathbf{e}_n\right) + K_I \sum_{i=0}^{n} \left(\mathbf{t}_n - \mathbf{e}_n\right) \tag{5.10}$$

$\mathbf{s}$ is then used as the motion (Δx, Δy, $\Delta\varphi$) that has to be completed within the time Δt until the next sensor update. As the open-loop control is accurate, $K_P = 1$ already leads to good movement behavior. The accuracy is slightly increased by the integral part with $K_I = 0.05$ (Jasper et al., 2010).

An important key property of closed-loop control is the repeat accuracy, also called repositioning accuracy. To measure the repeat accuracy, an optical microscope was used as reference similar to the linearity measurements in the previous section. In 1000 repetitions, the robot was moved randomly and brought back to its original position using closed-loop positioning. The final position after each repositioning was recorded with high precision using the optical microscope. The standard deviation within the measurements was 0.0025 pixels (250 nm) and the deviation between the minimum and maximum result, i.e. the maximum error, was 0.018 pixels. The test was repeated at different positions with similar results. As the results are virtually identical to the sensor noise, the noise can be assumed to be the limiting factor on the repeat accuracy.

To verify the mobile robot's capability to stay on a predefined trajectory, several benchmark trajectories have been tested. Figure 5.16 shows the deviation from the trajectory when driving along the edges of a 10 px, i.e. approx. 1 mm, square without rotating. For fast movement, each edge was set to require 350 ms with 50 ms stops at each corner. Thus, the robot needs to move with a maximum velocity of 3 mm/s and the entire square requires 1.6 s. The slow movement is four times slower. The movement errors on the fast trajectory are up to 0.05 px, because with the still limited (145 Hz) update-rate, the controller cannot react fast enough to compensate for motion deviations. For the slower trajectory, the error remains at the noise level.

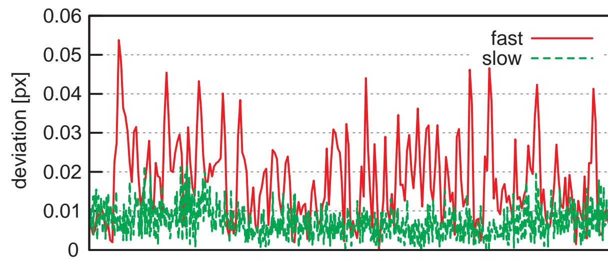


Figure 5.16: Error over time while driving along a linear trajectory at two different speeds (1 px ≈ 100 μm) (Jasper et al., 2010).

Figure 5.17 shows a single edge movement. For the first two sensor updates on the trajectory, an offset between the trajectory and the measurement value is visible. This is caused by the robot not exactly corresponding with the open-loop control model. From the third update on, this deviation is compensated for by the integral part of the controller and the measurements appear identical to the calculated trajectory (Jasper et al., 2010).

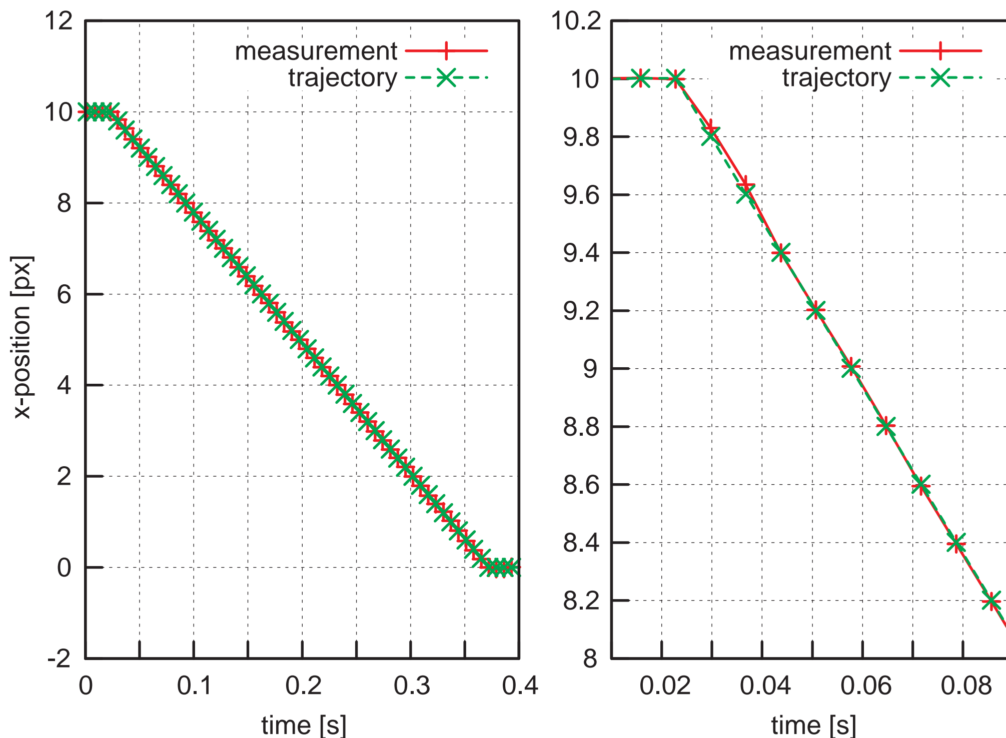


Figure 5.17: Measurements of a linear trajectory and magnification on the first part (1 px ≈ 100 μm) (Jasper et al., 2010).

5.1.6 Conclusions

Based on the robust FPGA-tracking system, a fully integrated handling cell with high accuracy and fast positioning could be developed. The control loop is executed on real-time capable microcontrollers and dedicated embedded logic without the assistance of a computer. A computer can be used to send high-level control commands to the robot controller requiring virtually no computation time. Thus, the control performance and accuracy is independent of the computer's performance and load.

The control algorithm compensating for the delay only works correctly if the latency and update-rate are predictable and have a very small jitter. With the new controller and FPGA-based tracking, the closed-loop positioning speed could be improved by one magnitude compared to software-based tracking. A single pick and place operation of a 50 µm spherical object was achieved in less then two seconds, compared to more than 15 seconds with the previous, software-based system.

In this application, tracking speed, low latency and small jitter were the requirements that made an FPGA implementation necessary. However, an FPGA implementation is also beneficial when looking at small, cost-efficient low power applications.

5.2 Traffic sign recognition

The application presented in this section was developed by a student group at the University of Oldenburg during the project yaDAS[2]. The student group was supervised by Dr. Daniel Jasper and the author. The project aimed to develop a real-time camera-based system for driver assistance.

The main goal was to develop a small, power- and cost-efficient camera-based system for driver assistance that can be retrofitted to any car. The system was created aiming at several sub-goals: the detection and recognition of German traffic signs, the detection of road markings and the observation of the driver for signs of tiredness.

The group used the BLOB-detection algorithm developed by the author for a preprocessing step of the traffic sign recognition system.

The traffic sign recognition is based on color detection. Several German traffic signs are enclosed by a red border. Examples of such traffic signs can be seen in figure 5.18

[2] yet another Driving Assistance System. Project report (german) at `http://www.boles.de/teaching/pg_fb10/endberichte/2011/yaDAS.pdf`. Last access: October 15, 2014.

Figure 5.18: German traffic signs. From left to right: stop, give way, no overtaking, speed regulation, weight regulation, general warning.

The first step was to detect a traffic sign in the camera stream. If a traffic sign of significant size is detected, its general shape is classified based on the binary large object features. Then, if necessary, its content is matched against a set of existing templates to identify the correct sign content.

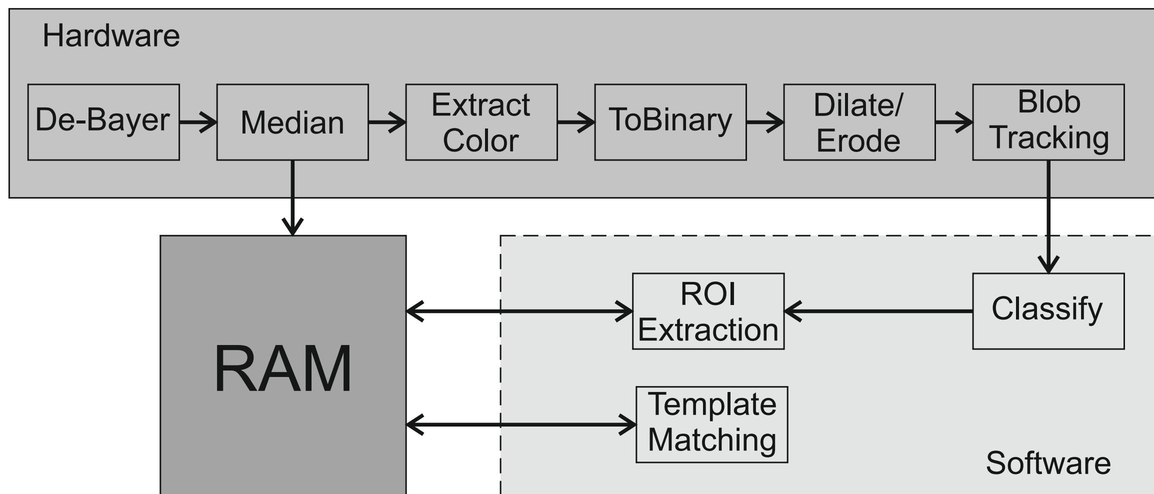


Figure 5.19: The architecture of the traffic sign recognition system. The stream-based hardware implementation performs several pre-processing steps before the BLOB extraction is performed. If a valid sign is detected, the corresponding region is extracted from the memory holding the full image and matched against predefined templates.

Figure 5.19 shows the architecture of the traffic sign recognition system. The used camera is a color camera with an LVDS[3] interface that resembles the camera stream protocol described in section 2.2. The system first uses a de-Bayer filter to create an RGB image from the Bayer encoded image. See section 4.2 for more information on the Bayer pattern. In the next step, the image is smoothened

[3]Low Voltage Differential Interface

using a weighted median filter. Then, the red channel is extracted from the image. Instead of directly using the red channel of the RGB image, the following formula is used:

$$R' = \max\left(0, R - \left(\frac{B}{2} + \frac{G}{2} + |B - G|\right)\right) \tag{5.11}$$

Using this formula, only values with high red values and low blue and green values are above zero and considered foreground for the following filters. The next step in the hardware chain is a binarization filter. This filter reduces the eight-bit image to a one-bit image by applying a threshold. Pixels above the threshold are considered foreground and will have a one assigned in the resulting pixel stream; all other pixels will be assinged a zero in the pixel stream.

A closing morphological operation is performed as the next step. A closing consist of two steps, a dilate step and an erode step (Serra, 1982). See section 2.2 for more information.

After the morphological operation, the BLOB-detection is applied. The BLOB-implementation used is tailored to binary (one bit) images. Filters for the minimum number of pixels and for the minimum bounding box in hardware implementation ensured that only signs with significant size were considered.

An example of the hardware chain is shown in figure 5.20

The RGB images were additionally written to the memory. A double buffer strategy was implemented, in which concurrent images are written into two alternating buffers. The embedded processor can lock one of the buffers if it needs to operate on the image in the memory.

If a traffic sign with sufficient dimensions is detected by the hardware chain, it raises an interrupt to the embedded processor. The software program firstly identifies the general sign shape using the features provided by the BLOB-detection. The goal is to check whether the found object is really a sign and to provide a first classification.

First, the dimensions of the bounding box are checked. The ratio of height to width has to be within specific bounds to be considered a valid traffic sign. In general, the width will not be larger than the height for any traffic sign. Furthermore, if the width is less than $\frac{2}{3}$ of the height, the object either is not a traffic sign or the sign is revolved too far and cannot be analyzed.

The sign can then be classified as one of the following four:

- Round sign with red border
- Triangular sign with red border

- Triangular sign with red border (180° rotated)
- Sign filled with red

A found object can be classified as one of the four categories by checking the number of pixels, the area inside the bounding rectangle, the center of the bounding rectangle and the center of gravity.

For a filled sign (e.g. stop sign) the ratio between the number of pixels and the area inside the bounding rectangle is significantly different from a sign with a red border. Comparing the center of the bounding rectangle and the center of gravity, the particular red-bordered sign can be perceived. For a round sign, the center of gravity and the center of the bounding box will be very close together. For a triangular sign, the center of gravity will be shifted upwards or downwards related to the center of the bounding box. If the center of gravity is geometrically lower then the center of the bounding box, the sign has an edge at the bottom side (e.g. warning sign). If it is geometrically higher, the triangle has an edge at the top side (e.g. give way).

If the region found by the BLOB-detection does not fit into any of the categories, it is not classified further. Some of the categories need to be classified further. A lot of significant signs are round signs with a red border. If such a sign is

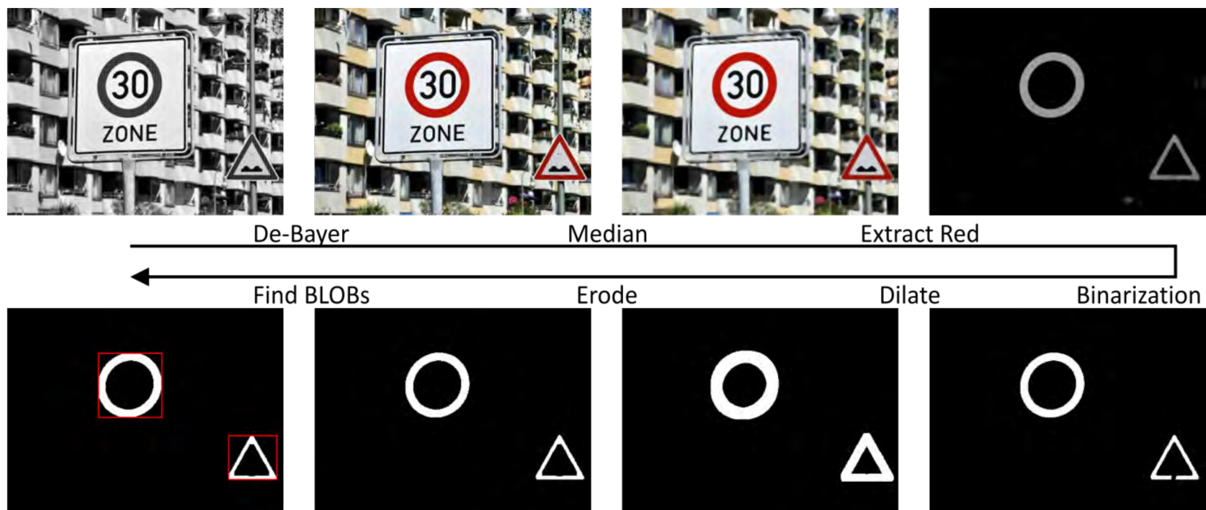


Figure 5.20: The different states of the image in the hardware chain. First, the image is de-Bayered. Then, it is filtered and the red channel is extracted. Afterwards, it is converted to a binary image and a morphological closing operation is performed. Last, the BLOB-extraction is performed. In the hardware chain, these images do not exist in a memory. Every pixel is streamed through the hardware without being stored in intermediate images.

encountered, the RGB image in the memory is locked and the region inside the border is extracted and compared to predefined templates using template matching. If one of the templates matching quality exceeds a boundary, the content of the sign has been successfully identified.

Traffic sign recognition can be implemented in a low cost FPGA. The system was successfully deployed and tested using the Spartan3-1200 development board Nexys2[4]. The development board, together with the camera and an LCD display, needs less than 2.5 W of power. The prototype can find and classify speed restrictions, stop signs, give way signs, warning signs as well as signs prohibiting overtaking. Stop signs and give-way signs are signaled with a blinking LED, the last seen speed restrictions and overtaking restrictions are visualized in the LCD display.

Using an FPGA system, it is possible to implement a traffic sign recognition device that is small, portable and power saving. By using a hardware-based BLOB-detection it was possible to find possible traffic signs in real-time in the camera stream. Only the detected signs were transferred to the embedded processor. Without the hardware image-processing, a more powerful processor is needed to analyze the image.

5.3 Conclusions

Using the FPGA-based object detection system, it was possible to speed up a microhandling cell by one magnitude. A single pick and place operation of a 50 µm spherical object was achieved in less then two seconds, compared with more than 15 seconds of the previous, software-based system. The speedup was possible for two reasons: first, the FPGA-based system was tailored to the application and could control the camera directly and deterministically. This way, it was possible to achieve higher update-rates by using a moving region of interest directly in the camera. Secondly, the deterministic and virtually jitter-free behavior of FPGA-based motion tracking made it possible to develop a controller able to compensate for the delay.

The second application shows the possibilities of FPGA-based object detection. The object detection was used to identify traffic signs in a camera stream. Based on a set of features, it was possible to classify the road signs into different categories. Using an FPGA system, it is possible to implement a traffic sign recognition device that is small, portable and power saving.

The FPGA-based object detection system can be used in many more applications, especially high-speed motion tracking or object detection applications. The robust-

[4] `www.digilentinc.com/nexys2/`. Last access: October 15, 2014.

ness of the algorithm and the high number of features allows for a wide range of scenarios in which it can be deployed.

6 Conclusions and outlook

In this thesis, the use of FPGAs for motion tracking is proposed. Especially in microrobotics, closed-loop positioning is a major step that consumes a great amount of time in automated processes. In many cases closed-loop positioning is based on cameras and image-processing. Image processing is used to detect an object in the camera image. Software based image processing in PCs is prone to nondeterministic timing behavior in both update-rate and processing time. This has been demonstrated in chapter 4.

To allow for deterministic image processing, so called stream processing can be used in FPGAs. Using such a technique, processing times become predictable. The FPGA architecture allows for high parallelization and therefore high throughput. In chapter 3 a novel implementation of the binary large object (BLOB) algorithm based on stream-processing was introduced. Using this algorithm, it is possible to perform object detection on 1 MP images with an update-rate above 1 Khz. The algorithm is robust and has a deterministic runtime behavior that is independent of the image content (e.g. noise, number of objects).

The algorithm was deployed in a motion tracking system and validated. The validation covered the timing-robustness against noise and different image contents. Furthermore, the timing behavior was validated and compared to PC-based image-processing in chapter 4. The validation tests additionally included tests with a real time operating system.

Two applications are presented where the developed algorithm was deployed. The first application is motion tracking for micro robot closed-loop positioning. For this application, the robustness and the predictability of the image-processing latency was the key to enhancing the closed-loop positioning time significantly. The second application was a lightweight low-power traffic sign recognition system. In this application, the latency of the overall process was of secondary importance. However, the goal of the developed system was that it should be cheap, small and power-efficient. For this reason a traffic sign recognition system based on a low-cost FPGA and a low-cost camera chip was used. The fast and robust hardware implementation of the BLOB algorithm allowed the real-time detection of traffic signs without using the embedded processor.

The FPGA-based object detection system can be used in many more applications, especially high-speed motion tracking or object detection applications. The robustness of the algorithm and the high number of detectable features allow for a wide range of scenarios in which it can be deployed.

6.1 Conclusions

The results of this thesis show that hardware-based object detection using FPGAs is superior to software-based image processing concerning robustness in timing behavior. Furthermore, due to the high possible parallelization it can handle more data and can therefore reach higher update-rates. While earlier FPGA implementations of the BLOB extraction algorithm rely on long line and frame blanking times or require severe preconditions to the image content; the novel FPGA implementation proposed in this thesis is robust against any kind of image. This is underlined by the fact that several thousand randomly generated images were tested. The tests included images with small- to medium-sized objects with different orientations and shapes. Furthermore, images consisting of pure noise were tested. The algorithm performed well for all tested images. Furthermore, the hardware utilization of the algorithm is moderate, allowing it to be synthesized even on one of the smallest low-cost FPGAs currently available. Opposite to almost all previous implementations, the implementation in this thesis can handle several pixels in one cycle, allowing for throughputs of up to 1,84 GB/s.

The robustness of the algorithm was tested and compared to software-based image processing, including a real-time operating system. The deviation of the update-rate as well as the achievable latency and the latency jitter was measured and analyzed. The software implementation has timing issues, especially if the CPU has additional tasks and therefore a high load. This results in a long and unpredictable latency and high jitter in the update-rate. With a real-time extension, large outliers and missed updates could be eliminated. However, the deviation in update-rate and latency was still in the range of milliseconds.

FPGA implementation shows superior performance in every timing characteristic. Firstly, the jitter in the update-rate and the latency is in the range of single microseconds. This is two magnitudes better than all software-based approaches. Furthermore, the latency of FPGA-based image-processing is 3.2 ms lower than the best latency achievable by a PC-based implementation. The latency of FPGA tracking is only a few µs slower than the acquisition and transfer time of the camera and the latency is independent of the image-content.

Using the FPGA-based object detection system in a microhandling cell, it was possible to speed up the pick and place operations by one magnitude. A single

pick and place operation of a 50 µm spherical object was achieved in less then two seconds, compared to more than 15 seconds with the previous, software-based system. The speedup was possible for two reasons: Firstly, the FPGA-based system was tailored to the application and could control the camera directly and deterministically. This is how it was possible to achieve higher update-rates by using a moving region of interest directly in the camera. Secondly, the deterministic and virtually jitter-free behavior of FPGA-based motion tracking made it possible to develop a controller able to compensate for the delay.

In a second application, the possibilities of FPGA-based object detection were shown. The object detection was used to identify traffic signs in a camera stream. Based on a set of features, it was possible to classify the road signs into different categories. Using an FPGA system, it is possible to implement a traffic sign recognition device that is small, portable less power consuming.

6.2 Outlook

While the FPGA implementations outperform software implementations in terms of timing performance and throughput, the deployment of such systems is complicated and time-consuming. Furthermore, the development and adaption of the FPGA algorithm needs developers with special skills. A solution could be the upcoming systems that can generate hardware designs from software descriptions. One example of such a system is the CoSynth Synthesizer[1], based on SystemC. With such a system, the development of hardware-based algorithms could be much faster. Unfortunately, the area consumption and the throughput of such generated algorithm are inferior to hand-crafted implementations (Tiemerding et al., 2013). However, the development times using such a system are shorter.

If even higher throughput is needed, a combination of the proposed algorithm with the approach of dividing the image into multiple vertical slices proposed by Klaiber et al. (2013) could be interesting. Both algorithms achieve very high throughputs by using different approaches: The algorithm presented in this thesis calculates multiple adjacent pixels in one clock cycle; the algorithm by Klaiber et al. (2013) calculates one pixel per image slice, but with multiple slices in each clock cycle. Combining both could result in throughputs of up to 10 GB/s.

Further work could go into classification. The presented applications use static bounds to classify the different found regions. Another solution is to use a self-training system. A neural network is well-suited to be used as a hardware-based solution. A small feed-forward neural network can easily be implemented for FPGA-use using a parallel pipelined approach (Salapura et al., 1994). Inputs to

[1] `https://www.cosynth.com/`. Last access: October 15, 2014.

the neural network are the features of the connected component labeling. Since some features are not single values that can be used as input to the neural network, a deduction of the actual features can be used. For a feature, several deductions can be possible. For the bounding rectangle, the enclosed area as well as the factor height/width are relevant sub-features for the classification. The outputs of the neural network are the different possible object types. As customary for neural networks used for classification, only one of the output neurons produces an output. The training of the neural network is performed offline. The FPGA-based network uses the computed weights for the connections and the activation functions for the neurons (Diederichs and Fatikow, 2013).

While the presented BLOB extraction algorithm performs well for setups where foreground and background can be distinguished, it is certainly not suited for every motion tracking application in microrobotics. This work can be a first step towards a toolbox of useful algorithms with different advantages. One of the most popular motion tracking algorithm is the template matching algorithm (Sievers and Fatikow, 2006). However, the number of computations that are necessary for template matching obstruct the development of fast FPGA implementations. While several approaches for FPGA-based template matching exist (Hezel et al., 2002; Gause et al., 2002; Liang and Jean, 2003; Hashimoto et al., 2013; Elfert et al., 2014), most of them need high-end FPGAs. Also, the achievable throughput is not in within the range of high-speed motion tracking. A different algorithm that could be well-suited for high speed motion tracking is the active contours algorithm (Blake and Isard, 1998; Dejnožková and Dokládal, 2005).

Bibliography

A. AbuBaker, R. Qahwaji, S. Ipson, and M. Saleh. One scan connected component labeling technique. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 1283 –1286, nov. 2007. doi: 10.1109/ICSPC.2007.4728561.

BP Amavasai, F. Caparrelli, A Selvan, M Boissenin, JR Travis, and S Meikle. Machine vision methods for autonomous micro-robotic systems. *Kybernetes*, 34 (9/10):1421–1439, 2005.

D. G. Bailey. Raster based region growing. In *Proceedings of the 6th New Zealand Image Processing Workshop*, pages 21–26, 1991.

D. G. Bailey. A New Approach to Lens Distortion Correction. In *Proceedings Image and Vision Computing New Zealand*, pages 59–64, 2002.

D. G. Bailey. *Design for Embedded Image Processing on FPGAs*. John Wiley & Sons (Asia) Pte Ltd, 2011. ISBN 9780470828519. doi: 10.1002/9780470828519.

D. G. Bailey and C. T. Johnston. Single Pass Connected Components Analysis. In *Image and Vision Computing New Zealand (IVCNZ)*, 2007.

B. E. Bayer. Color imaging array, July 20 1976. US Patent 3,971,065.

K. Benkrid, S. Sukhsawas, D. Crookes, and A. Benkrid. An FPGA-Based Image Connected Component Labeller. In Peter Cheung and GeorgeA. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 1012–1015. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40822-2. doi: 10.1007/978-3-540-45234-8_108. URL `http://dx.doi.org/10.1007/978-3-540-45234-8_108`.

A. Blake and M. Isard. *Active Contours*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 25(11): 120, 122–125, November 2000. ISSN 1044-789X. URL `http://www.ddj.com/ftp/2000/2000_11/opencv.txt`.

F. Chang, C.J. Chen, and C.J. Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, 93:206–220, 2004.

F. Chaumette. Image moments: a general and useful set of features for visual servoing. *Robotics, IEEE Transactions on*, 20(4):713–723, 2004.

D. Crookes and K. Benkrid. FPGA implementation of image component labeling. In J. Schewel, P. M. Athanas, S. A. Guccione, S. Ludwig, and J. T. McHenry, editors, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 3844 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 17–23, August 1999.

C. Dahmen, T. Wortman, and S. Fatikow. OLVIS: A modular image processing software architecture and applications for micro- namd nanohandling. In *Proc. of IASTED International Conference on Visualization, Imaging and Image Processing (VIIP)*, 2008.

N. Dechev, W. L. Cleghorn, and J. K. Mills. Microassembly of 3-D microstructures using a compliant, passive microgripper. *Microelectromechanical Systems, Journal of*, 13(2):176–189, 2004.

E. Dejnožková and P. Dokládal. Embedded real-time architecture for level-set-based active contours. *EURASIP J. Appl. Signal Process.*, 2005:2788–2803, 2005. ISSN 1110-8657. doi: http://dx.doi.org/10.1155/ASP.2005.2788.

C. Diederichs. Hardware-Software Co-Design Tracking System for Predictable High-Speed Mobile Microrobot Position Control. In *Proc. of IFAC Symposium on Mechatronic Systems*, 2010.

C. Diederichs. Fast Visual Servoing of Multiple Microrobots using an FPGA-Based Smart Camera System. In *Proc. of the 18th IFAC World Congress*, 2011.

C. Diederichs and S. Fatikow. FPGA-based high-speed object detection and classification inside optical microscopes. In *Proc. of the 8th International Workshop on Microfactories*, June 2012.

C. Diederichs and S. Fatikow. FPGA-Based Object Detection and Motion Tracking in Micro- and Nanorobotics. *International Journal of Intelligent Mechatronics and Robotics*, 3:27–37, 2013.

C. Diederichs, S. Zimmermann, and S. Fatikow. FPGA-based object detection and classification inside scanning electron microscopes. In *Proc. of the Second International Conference on Manipulation, Manufacturing and Measurement on the Nanoscale (3M-NANO)*, 2012.

C. Diederichs, M. Bartenwerfer, M. Mikczinski, S. Zimmermann, T. Tiemerding, C. Geldmann, H. Nguyen, C. Dahmen, and S. Fatikow. A Rapid Automation Framework for Applications on the Micro- and Nanoscale. In *Proc. of the Australasian Conference on Robotics and Automation 2013*, page 8, December 2013.

C. Edeler, D. Jasper, and S. Fatikow. Development, Control and Evaluation of a Mobile Platform for Microrobots. In *Proc. of the 17th IFAC World Congress*, pages 12739–12744, Seoul, Korea, July 2008.

C. Edeler, D. Jasper, C. Diederichs, and S. Fatikow. Fast and Accurate Pick-and-Place Automation with Nanorobots. In *Proc. of Intl. Conference on New Actuators*, pages 397–400, Bremen, June 2010.

P. Elfert, T. Tiemerding, C. Diederichs, and S. Fatikow. Advanced Methods for High-Speed Template Matching targeting FPGAs. In *Proc. of International Symposium on Optomechatronic Technologies (ISOT) 2014*, November 2014.

S. Fatikow, A. Buerkle, and J. Seyfried. Automatic control system of a microrobot-based microassembly station using computer vision. In *Photonics East'99*, pages 11–22. International Society for Optics and Photonics, 1999.

S. Fatikow, J. Seyfried, A Buerkle, and F Schmoeckel. A flexible microrobot-based microassembly station. *Journal of Intelligent and Robotic Systems*, 27(1-2): 135–169, 2000.

S. Fatikow, C. Stolle, C. Diederichs, and D. Jasper. Auto-configuration and self-calibration in fexible robot-based micro-/nanohandling cells. In *Proc: of the 18th IFAC World Congress*, 2011.

A. Ferreira, C. Cassier, and S. Hirai. Automatic microassembly system assisted by vision servoing and virtual reality. *Mechatronics, IEEE/ASME Transactions on*, 9(2):321–333, 2004.

PE. Forssén and G. Granlund. Robust multi-scale extraction of blob features. In *Image Analysis*, pages 11–18. Springer, 2003.

H. Freeman. Techniques for the digital computer analysis of chain-encoded arbitrary plane curves. In *Proc. Nat. Electronics Conf*, volume 17, pages 412–432, 1961.

Z Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, May 1964. ISSN 0001-0782. doi: 10.1145/364099.364331. URL `http://doi.acm.org/10.1145/364099.364331`.

J. Gause, P. YK. Cheung, and W. Luk. Reconfigurable shape-adaptive template matching architectures. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 98–107. IEEE, 2002.

K. T. Gribbon, D. G. Bailey, and C. T. Johnston. Using Design Patterns to Overcome Image Processing Constraints on FPGAs. In *DELTA '06: Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications*, pages 47–56, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2500-8. doi: http://dx.doi.org/10.1109/DELTA.2006.93.

T. Hasegawa, N. Ogawa, H. Oku, and M. Ishikawa. A new framework for micro-robotic control of motile cells based on high-speed tracking and focusing. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3964–3969. IEEE, 2008.

K. Hashimoto, Y. Ito, and K. Nakano. Template Matching using DSP slices on the FPGA. In *Computing and Networking (CANDAR), 2013 First International Symposium on*, pages 338–344. IEEE, 2013.

L. He, Y. Chao, and K. Suzuki. A linear-time two-scan labeling algorithm. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, volume 5, pages V–241. IEEE, 2007.

Y. He and A. Kundu. Shape classification using hidden markov model. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 2373–2376. IEEE, 1991.

H. Hedberg, F. Kristensen, and V. Owall. Implementation of a labeling algorithm based on contour tracing with feature extraction. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1101–1104. IEEE, 2007.

S. Hezel, A. Kugel, R. Manner, and DM. Gavrila. FPGA-based template matching using distance transforms. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 89–97. IEEE, 2002.

XH. Huang, XJ. Zeng, and M. Wang. SVM-based Identification and Un-calibrated Visual Servoing for Micro-manipulation. *International Journal of Automation and Computing*, 7(1):47–54, 2010.

ISO/IEC. Road vehicles – Controller area network (CAN) – Part 1 & 2. Technical report, International Organization for Standardization, Geneva, Switzerland., 15. October 2003.

D. Jasper. *SEM-based motion control for automated robotic nanohandling*. PhD thesis, Oldenburg University, 2011.

D. Jasper and C. Edeler. Characterization, Optimization and Control of a Mobile Platform. In *Proc. of 6th Int. Workshop on Microfactories (IWMF)*, pages 143–148, Evanston, IL, USA, October 2008.

D. Jasper, C. Diederichs, and S. Fatikow. Hardware-based, Trajectory-Controlled Visual Servoing of Mobile Microrobots. In *Proc. of IFAC Symposium on Mechatronic Systems*, pages 565–570, Cambridge, MA, USA, September 2010.

D. Jasper, C. Diederichs, C. Edeler, and S. Fatikow. High-speed nanorobot position control inside a scanning electron microscope. *ECTI Transactions on Electrical Eng., Electronics, and Communications*, 9(1):177–186, Feburary 2011a.

D. Jasper, C. Diederichs, C. Stolle, and S. Fatikow. Automated robot-based separation and palletizing of microcomponents. In *Assembly and Manufacturing (ISAM), 2011 IEEE International Symposium on*, pages 1 –6, may 2011b. doi: 10.1109/ISAM.2011.5942339.

C.T. Johnston and D.G. Bailey. FPGA implementation of a Single Pass Connected Components Algorithm. In *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, pages 228 –231, jan. 2008. doi: 10.1109/DELTA.2008.21.

M. Kharboutly and M. Gauthier. High speed closed loop control of a dielectrophoresis-based system. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1446–1451. IEEE, 2013.

M. Klaiber, L. Rockstroh, Zhe Wang, Y. Baroud, and S. Simon. A memory-efficient parallel single pass architecture for connected component labeling of streamed images. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 159–165, Dec 2012. doi: 10.1109/FPT.2012.6412129.

M.J. Klaiber, D.G. Bailey, S. Ahmed, Y. Baroud, and S. Simon. A high-throughput FPGA architecture for parallel connected components analysis based on label reuse. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 302–305, Dec 2013. doi: 10.1109/FPT.2013.6718372.

A. Krupa, J. Gangloff, C. Doignon, M. F. de Mathelin, G. Morel, J. Leroy, L. Soler, and J. Marescaux. Autonomous 3-D positioning of surgical instruments in robotized laparoscopic surgery using visual servoing. *Robotics and Automation, IEEE Transactions on*, 19(5):842–853, 2003.

YS. Lee and CS. Koo, HS.and Jeong. A straight line detection using principal component analysis. *Pattern Recogn. Lett.*, 27:1744–1754, October 2006. ISSN 0167-8655. doi: 10.1016/j.patrec.2006.04.016. URL `http://dl.acm.org/citation.cfm?id=1195775.1195792`.

X Liang and JS. N. Jean. Mapping of generalized template matching onto reconfigurable computers. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(3):485–498, 2003.

N. Ma, D.G. Bailey, and C.T. Johnston. Optimised single pass connected components analysis. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 185–192, Dec 2008. doi: 10.1109/FPT.2008.4762382.

Z. Ni, A. Bolopion, J. Agnus, R. Benosman, and S. Régnier. Asynchronous event-based visual shape tracking for stable haptic feedback in microrobotics. *Robotics, IEEE Transactions on*, 28(5):1081–1089, 2012.

N. Ogawa, H. Oku, K. Hashimoto, and M. Ishikawa. Microrobotic visual control of motile cells using high-speed tracking system. *Robotics, IEEE Transactions on*, 21(4):704–712, 2005.

NP. Papanikolopoulos, B. Nelson, and PK. Khosla. Full 3-D tracking using the controlled active vision paradigm. In *Intelligent Control, 1992., Proceedings of the 1992 IEEE International Symposium on*, pages 267–274. IEEE, 1992.

I. Pappas and A. Codourey. Visual control of a microrobot operating under a microscope. In *Intelligent Robots and Systems' 96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on*, volume 2, pages 993–1000. IEEE, 1996.

E. Persoon and KS. Fu. Shape discrimination using fourier descriptors. *Systems, Man and Cybernetics, IEEE Transactions on*, 7(3):170–179, 1977.

C. Poynton. *Digital video and HD: Algorithms and Interfaces*. Elsevier, 2012.

R. V. Rachakonda, P. M. Athanas, and A. L. Abbott. High-speed region detection and labeling using an FPGA-based custom computing platform. In Will Moore and Wayne Luk, editors, *Field-Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-60294-1. doi: 10.1007/3-540-60294-1_101. URL http://dx.doi.org/10.1007/3-540-60294-1_101.

R. Rodrigo, W. Shi, and J. Samarabandu. Energy Based Line Detection. In *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, pages 2061 –2064, may 2006. doi: 10.1109/CCECE.2006.277538.

A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM (JACM)*, 13(4):471–494, 1966.

V. Salapura, M. Gschwind, and O. Maischberger. A Fast FPGA Implementation of a General Purpose Neuron. In *Proceedings of the 4th International Workshop on Field-Programmable Logic and Applications: Field-Programmable Logic, Architectures, Synthesis and Applications*, FPL '94, pages 175–182, London, UK, 1994. Springer-Verlag. ISBN 3-540-58419-6. URL `http://dl.acm.org/citation.cfm?id=647921.740703`.

L. Schomaker and M. Bulacu. Automatic writer identification using connected-component contours and edge-based features of uppercase western script. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(6):787–798, 2004.

J. Serra. *Image analysis and mathematical morphology, v. 1.* Academic press, 1982.

T. Sievers and S. Fatikow. Real-time object tracking for the robot-based nanohandling in a scanning electron microscope. *Journal of Micromechatronics*, 3(3/4): 267, 2006.

Y. Sun and B. J. Nelson. Biological cell injection using an autonomous microrobotic system. *The International Journal of Robotics Research*, 21(10-11):861–868, 2002.

S Suzuki and K Abe. Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing*, 30(1):32–46, 1985.

B. Tamadazte, E. Marchand, S. Dembélé, and N. Le Fort-Piat. CAD model-based tracking and 3D visual-based control for MEMS microassembly. *The International Journal of Robotics Research*, 2010.

B. Tamadazte, N. Piat, and S. Dembélé. Robotic micromanipulation and microassembly using monoview and multiscale visual servoing. *Mechatronics, IEEE/ASME Transactions on*, 16(2):277–287, 2011.

B. Tamadazte, N. Le-Fort Piat, and E. Marchand. A direct visual servoing scheme for automatic nanopositioning. *Mechatronics, IEEE/ASME Transactions on*, 17 (4):728–736, 2012a.

B. Tamadazte, M. Paindavoine, J. Agnus, V. Pétrini, and N. Le-Fort Piat. Four DOF Microgripper Equipped With a Smart CMOS Camera. *Microelectromechanical Systems, Journal of*, 21(2):256–258, 2012b.

T. Tiemerding, C. Diederichs, C. Stehno, and S. Fatikow. Comparison of different Design Methodologies of Hardware-based Image Processing for Automation in Microrobotics. In *Proc. of International Conference on Advanced Intelligent Mechatronics (AIM), 2013 IEEE/ASME*, 2013.

K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163 – 1169, 1995.

J Trein, A. T. Schwarzbacher, B Hoppe, K. Noffz, and T. Trenschel. Development of a FPGA based real-time blob analysis circuit. In *Proceedings of the 2007 international conference on system and safety*, 2007.

J Trein, A. T. Schwarzbacher, and B Hoppe. FPGA implementation of a single pass real-time blob analysis using run length encoding. In *MPC-Workshop, February*, 2008.

W. H. Wang, X. Y. Liu, and Y Sun. High-throughput automated injection of individual biological cells. *Automation Science and Engineering, IEEE Transactions on*, 6(2):209–219, 2009.

T. Wortman and S. Fatikow. Carbon Nanotube Detection by Scanning Electron Microscopy. In *Proc. of the Eleventh IAPR Conference on Machine Vision Applications (MVA)*, 2009.

T. Wortmann, C. Dahmen, R. Tunnell, and S. Fatikow. Image processing architecture for real-time micro-and nanohandling applications. In *MVA*, pages 418–421, 2009.

K. Wu, E. Otoo, and A. Shoshani. Optimizing connected component labeling algorithms. In *Medical Imaging*, pages 1965–1976. International Society for Optics and Photonics, 2005.

Declaration of original work

I, Claas Diederichs, hereby declare to have written this thesis only based on the sources listed and without the help of others. I have not submitted or prepared the submission of this or any other doctoral thesis at the Carl von Ossietzky University Oldenburg or any other university.

I have honored the German Research Foundation guidelines for safeguarding good scientific practice in the completion of this work.

Hiermit erkläre ich, Claas Diederichs, diese Arbeit ohne fremde Hilfe und nur unter Verwendung der angegebenen Quellen verfasst zu haben. Ich habe bis dato weder an der Carl von Ossietzky Universität Oldenburg noch an einer anderen Universitiät die Eröffnung eines Promotionsverfahrens beantragt oder anderweitig eine Promotion vorbereitet.

Ich habe die Regeln guter wissenschaftlicher Praxis entsprechend der Richtlinien der Deutschen Forschungsgemeinschaft eingehalten.

(Claas Diederichs)