Using the ASTERICS Framework for Rapid Prototyping and Education in Image Processing on FPGAs

Philip Manke, Michael Schäferling, Gundolf Kiefer
University of Applied Sciences Augsburg
Efficient Embedded Systems Research Group
Augsburg, Germany
Email: {philip.manke, micheal.schaeferling, gundolf.kiefer}@hs-augsburg.de

***Abstract*—Considering the ongoing surge of interest in embedded computer vision technology, a growing demand for quickly and easily implemented systems exists. ASTERICS is a framework for designing complex image processing systems on FPGAs. Among others, the ASTERICS framework has already been used to implement complete object recognition systems based on the Generalized Hough Transform or SURF feature detection, but also simpler systems for educational use.**

This paper presents ASTERICS with a focus on its new Python-based system generation tool. This tool allows image and video processing systems to be defined easily by a short textual description in Python syntax. Benefits considering the development effort and time of this approach are described alongside a design example demonstrating the development process.

***Keywords*—Computer Vision; Embedded Vision; Toolchains; Image Processing; FPGA; Python; VHDL**

I. Introduction

Image and video processing systems still require very powerful processors to satisfy many of the modern use cases, such as in autonomous vehicles, due to the large amounts of data that need to be processed and, in many cases, to satisfy real-time constraints. Since FPGAs are a better fit than CPUs or GPUs for many of the common algorithms in the area of image and video processing, many companies and researchers are choosing FPGAs to implement these systems. This choice comes with the drawback that the design process for hardware description languages (HDL) as well as the synthesis and verification processes are more time consuming compared to software development. However, the increased energy efficiency and speedup of the algorithms is often worth the effort.

This paper introduces *Automatics*, a generator tool for image processing systems using the ASTERICS framework. The framework comprises a collection of interface standards, processing modules and tools for image and video processing on FPGAs. ASTERICS aims to simplify the development of image processing systems for simple and complex image processing tasks. Among others, it has previously been used to implement object detection systems using the SURF algorithm [15] or a generalized version of the Hough Transform [10, 22] and a positioning system using a Hough Transform for curved lines and sophisticated lens distortion correction and rectification [15, 16, 23]. ASTERICS offers a transparent design process: All core components are open source [18], all automatically generated source files aim to be human readable, and the debugging process is supported through testbenches and editable source files wherever possible. *Automatics* follows the same principles, as it is extendable and written completely in Python.

Section II reviews related work with respect to FPGA-based computer vision frameworks. In Section III, the ASTERICS framework and its background are presented in more detail. Section IV summarizes two example systems previously implemented using the ASTERICS framework. In Section V the system generator *Automatics* is presented. Section VI details the tool's use for educational purposes. In Section VII first experimental results in context of *Automatics'* use for rapid prototyping are presented. Section VIII concludes the paper with a brief summary and planned future work.

II. Related Work

Computer Vision tasks, including image and video processing, require large amounts of data to be processed. Different technologies are used to accelerate these types of workloads. GPUs have been used for these tasks, as their hardware architecture allows for many pixels to be processed in parallel.

Several approaches towards modular image processing architectures on FPGAs can be found in literature. In [5] a set of common architectures for video processing systems is presented, to be used as templates to decrease development time. [6] and [4] propose two architectures that provide the user with a shell of infrastructure to be expanded with one or more custom modules, implementing the functionality to be developed. In [7], the researchers focus on the user input methodology, expanding the Khoros GUI [11], originally meant for image processing development in software, with a backend for HDL generation. A more modern example for this approach is the tool Visual Applets by the company Silicon Software [20].

To the best of our knowledge, the only published project that comes close to our approach, in terms of the functionality we strive to implement, is the HDL generator mentioned in [21]. Sahlbach et al. sought to automate much of the process of hardware development using software tools. The tools presented include a HDL generator for connecting processing modules and utilities for the verification of the generated hardware.

The industry is also developing ASICs for general purpose image and video processing. Google has developed the Pixel Visual Core [8, 25], accelerating matrix multiplications using weakly programmable processing elements. Renesas [19] has developed a dynamically reprogrammable processor technology, mainly for image and video processing [13, 14]. The coprocessor contains multiple fixed size processing elements, which can individually be reprogrammed fairly freely, though not as versatile as FPGAs.

III. ASTERICS Overview

The *Augsburg Sophisticated Toolbox for Embedded and Realtime Image Crunching Systems* (ASTERICS) framework is an open toolbox for developing image and video processing systems on FPGAs. The framework provides a number of processing modules for common tasks, all sharing the same open interfaces.

A major focus in the development of ASTERICS lies in its modularity. According to [15] and [1], the individual image processing steps of a computer vision system can be grouped into four classes:

a) **Pixel-based**: Each result pixel requires only a *single input pixel*, e.g. contrast operations, color space conversions
b) **Window-based**: Each result pixel is calculated from a *delimited area* around the input pixel, e.g. edge filters
c) **Semi-global**: Each result value is dependent on a *variable section* of the input image, e.g. feature descriptors
d) **Global**: Each result is based on information collected from the *entire input image*, e.g. Hough-transform, descriptor matching

ASTERICS supports all of these classes. Operations of the classes a) and b) are generally best implemented as hardware modules in FPGA logic. Such modules are provided as VHDL source code using standard interfaces to communicate with each other and with software.

For operations of class c) the best implementation depends on the specific problem. On one hand, for example, the calculation of SURF feature descriptors is an algorithm not very well suited for the implementation in hardware. Therefore, as shown in [15], an array of softcore processors may be instantiated inside the system, with each processor calculating a descriptor in parallel to the others. On the other hand, for image rectification and undistortion, an implementation entirely in hardware has proven itself as feasible and efficient, as shown in [16].

Finally, operations of class d) are generally best implemented in software. For example, the matching of SURF descriptors is a task well suited for a general purpose processor, as implemented in [15].

The ASTERICS framework contains a module library, which is defined by the directory structure and contains VHDL source files, software drivers and metadata. Three common interfaces are defined to connect hardware modules with each other and to facilitate communication with the software: A streaming interface for pipeline architectures (`as_stream`), a pixel window interface for filter modules and a register interface for software communication. A software library ties control of all modules together, also allowing for ASTERICS to operate under Linux, using a kernel driver.

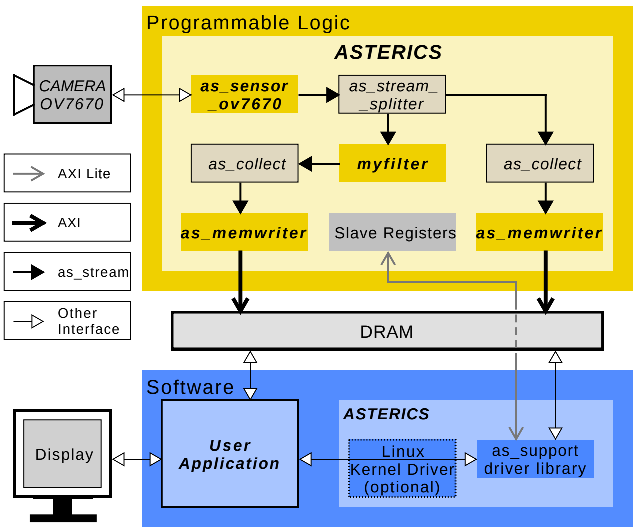


Figure 1. Representation of an example ASTERICS system including hardware and software components.

Figure 1 shows an example ASTERICS system, as it can be implemented on FPGA-SoC hardware. This example system uses an OmniVision7670 camera as a video source and writes two results into main memory, through the `as_memwriter` modules: The original camera image through the right path and a modified image, processed by some custom filter module (`myfilter`), through the left path. From main memory, the user application may directly use the results when running on bare metal or access them through the Linux driver using the `as_support` library.

IV. Example Systems

ASTERICS has been used to implement multiple complex systems. This section discusses two of these previously designed systems in more detail, to show the capability and modularity of the ASTERICS framework.

A. Object Detection on a Chip using the SURF Algorithm

For the general task of object detection, an ASTERICS system was implemented using point features, as presented in [15]. To locate objects in a captured scene, first point feature candidates need to be detected. For each candidate a descriptor is calculated in the next step of the algorithm. Finally, the descriptors are matched against a database of feature descriptors, each associated to a known object. The sophisticated SURF algorithm [3] was chosen as it provides strong point feature candidates and descriptors. Unfortunately, this algorithm is rather complex and demanding towards computational power and memory bandwidth so that FPGA-based hardware acceleration was necessary.

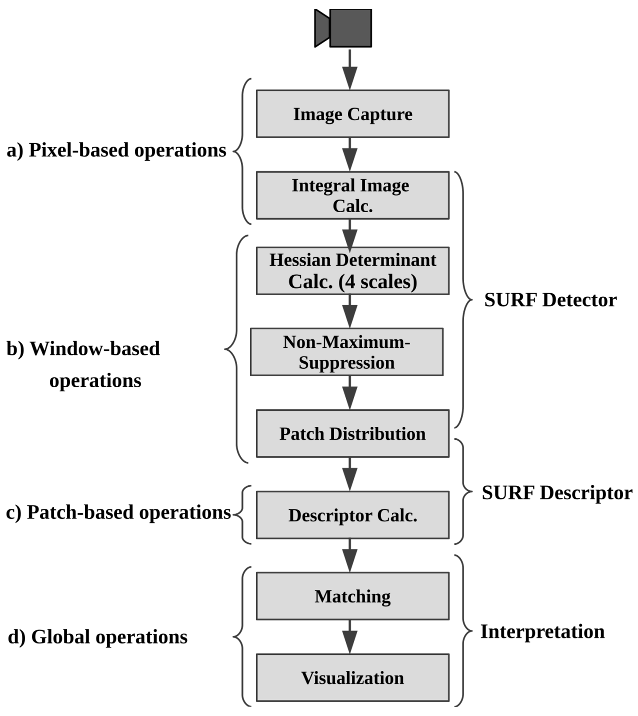


Figure 2. Steps of the SURF algorithm, assigned to image processing operation classes.

Figure 2 details the individual processing steps and how they are mapped to the four classes of operations introduced in Section III.

Figure 3 shows the structure of the resulting image processing system. A demonstrator application was built for this system in terms of a mobile museum guide for the *Augsburg Puppet Theatre Museum "die Kiste"*. The object database was built using just six still images in total of four museum exhibits as test objects.

In terms of execution speed, the system is able to calculate SURF descriptors at 18 FPS for a resolution of 640x480 pixels while operating at only 50 MHz, limited mainly by the

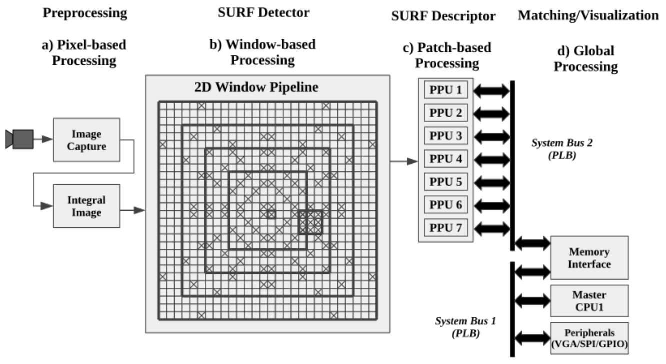


Figure 3. Hardware architecture of the ASTERICS system implementing a SURF-based object detection museum guide.

descriptor calculation, as the determinant calculation is able to run at up to 232 FPS. This makes the detector stage the most efficient and customizable at the time of its publication in [15].

B. Shape Recognition Using a Costumizable Hardware Implementation of the Generalized Hough Transform

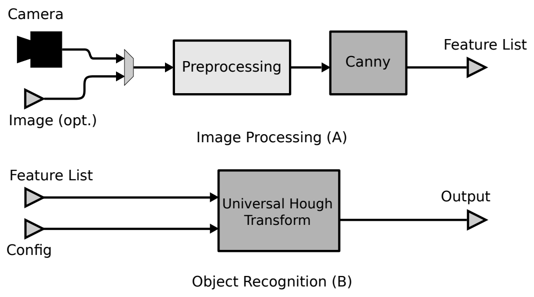


Figure 4. Hardware architecture to perform the Generalized Hough Transform

To perform efficient shape recognition tasks, a series of similar, customized image processing systems have been implemented using ASTERICS, as presented in [10] and [22]. Figure 4 shows the general structure of these systems which consist of various modules for image preprocessing, edge detection and perform variants of the Hough transform [2, 9].

The `Canny` module implements a 2D Window Pipeline which provides the edge features, weight and direction for the following Hough transform. The `Universal Hough transform` module can operate in General Hough Transform (GHT) mode, for finding arbitrary shapes in images, and in Line Hough Transform (LHT) mode, for finding straight lines. The *Universal Hough Transform (UHT)* module is described in detail in [22].

With this architecture, it is possible to perform a wide range of different shape recognition tasks, including the following examples:

- Groyne detection was performed in a GHT-based analysis of aerial images very efficiently [10].
- Detection of parts at construction sites, such as locating flanges of pipes, using GHT mode [22].

- Traffic sign detection is an application which can be efficiently performed using GHT mode (see Figure 5).
- In a race car application, cone detection is performed using GHT mode (see Figure 6).
- Lane detection can be performed using the UHT module in LHT mode (see Figure 7).

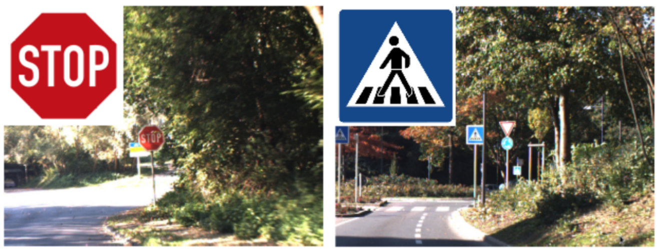


Figure 5. Examples for traffic sign detection (UHT in GHT mode) [10].

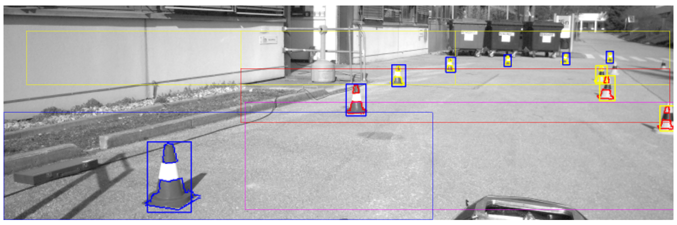
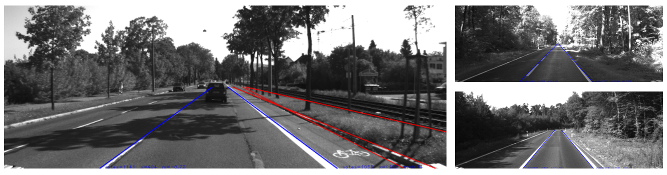


Figure 6. Example for cone detection in the race car (UHT in GHT mode) [22].



(a) (b, c)

Figure 7. Examples for lane detection (UHT in LHT mode) [22].

All of these shape recognition systems have been implemented on Xilinx Zynq XC7Z020 FPGA-SoCs, resulting in very cost and energy efficient systems. In the driverless race car of the University of Applied Sciences Augsburg, the SoC runs Linux where a complex software stack with additional *OpenCV* routines and a *ROS* (Robot Operating System) interface controls the ASTERICS subsystem.

In all these systems, the Hough Transform requires just a few milliseconds. For example, in the traffic sign recognition system (Fig. 5), a single GHT run required on average 11.0 ms for images with a resolution of 640x480 pixels with an average of 24.1×10^3 edge points. This is faster than the used image sensor modules which were operated at 30 frames per second.

V. Designing ASTERICS Systems using Automatics

The process of designing image and video processing systems is usually done using hardware description languages (HDL) like VHDL or Verilog or using High Level Synthesis tools. The ASTERICS system generator, *Automatics*, operates on a higher abstraction level. A first version of *Automatics* is introduced in brief in [12].

A. *Automatics Script*

Automatics uses a textual description of image processing systems on a processing module level. This description is written in Python syntax and consists mostly of single line method calls. The system description script, we believe, is simple enough to be understood and modified by users without knowledge of any programming language, while users experienced in programming can leverage the possibilities offered by the Python language.

Listing 1 shows a simple example of a system description script. The described system uses a camera as a pixel source, inverts all pixels, packages four pixels to 32 bit words using a collector module and writes the results to main memory using the `as_memwriter` module.

```
 1  # Setup ASTERICS Automatics
 2  import asterics
 3  chain = asterics.new_chain()
 4
 5  # Add processing modules
 6  camera = chain.add_module("as_sensor_ov7670")
 7  inverter = chain.add_module("as_invert")
 8  collect = chain.add_module("as_collect")
 9  writer = chain.add_module("as_memwriter")
10
11  # Configure "as_memwriter" module
12  writer.set_generic_value("MEMORY_DATA_WIDTH", 32)
13  writer.set_generic_value("DIN_WIDTH", 32)
14
15  # Describe module connections
16  camera.connect(inverter)
17  inverter.connect(collect)
18  collect.connect(writer)
19
20  # Start generation process
21  chain.write_system("inverter_system")
```

Listing 1. Description script for a simple pixel inverter system

Lines 2 and 3 import the ASTERICS library including *Automatics* and initialize the generation environment by creating a `chain` object. In lines 6 to 9, the four processing modules are added to the chain. The `as_memwriter` is then customized by setting two configuration options in lines 12 and 13. Lines 16 to 18 connect the modules in the order described above. Finally, in line 21, the `chain.write_system` method is called, which starts the generation process.

Notice how the setup and management process of importing the ASTERICS framework and starting the generation process requires just three lines of Python code. If the default configuration of a processing module is used, adding and connecting a module is just one line each, while changing a configuration value is one line for each value. Besides the methods shown in Listing 1, *Automatics* provides further configuration methods to allow the user to connect modules down to a port by port basis and configure all generic values. A detailed account of functionalities of *Automatics* and ASTERICS in general can be found in the ASTERICS Manual [24].

B. *Automatics Output Products*

Multiple methods are available to generate output products, including `write_system` as used in Listing 1. In general, *Automatics* generates system specific hardware and software source files on a system-by-system basis. With each generation process it copies or creates symbolic links to their respective hardware and software driver source files for all processing modules included in the system.

The following is a detailed list of available targets and their output products:

- `write_hw`: Generate the hardware (VHDL) source files.
- `write_sw`: Generate the software driver source files.
- `write_asterics_core`: Generate all source files required to build the ASTERICS IP-Core.
- `write_ip_core_xilinx`: Generate all source files and package the chain as an IP-Core for Xilinx Vivado. To accomplish this, TCL scripts are generated. At the time of writing, only the Xilinx toolchain is supported in this way, others may follow in the future.
- `vears`: Copy or create a symbolic link to the VEARS IP-Core. VEARS is an IP-Core for video output via HDMI, DVI and VGA and part of the ASTERICS framework.
- `write_system`: Generate an ASTERICS IP-Core and place it into an example system folder structure with the VEARS IP-Core. This may be used as a starting point for a new project with ASTERICS.
- `write_system_graph`: Generate a graphical representation of the described system as a vector graphic. Figure 10 shows an example.
- `list_address_space`: Print a list of addresses used by ASTERICS processing modules to the terminal.

Among the VHDL source files, *Automatics* generates two toplevel files used to define the interface of the ASTERICS IP-Core and to connect the processing modules with each other. The generated files aim to be human readable, with readable code formatting and signal and port names reflecting their origin in the generator script. Typically, a prefix is added to the existing port names, signifying their origin and association to a new entity.

Within the software driver, *Automatics* generates the main C header file. The architecture of the software driver is shown in Figure 8. The driver consists of the aforementioned header file, `asterics.h`, invoking all individual drivers of the systems various processing modules. Additionally, it contains hardware-specific details, depending on the used processing modules, such as their slave register addresses. The *ASTERICS Support Library* implements the most basic functionalities required by ASTERICS. Depending on whether the Linux kernel driver is included, it either accesses the processing modules directly through register access or using kernel function calls.

Figure 10 shows an example of the graphical representations of ASTERICS systems that *Automatics* can generate. The figure shows the graph of the simple invert system described

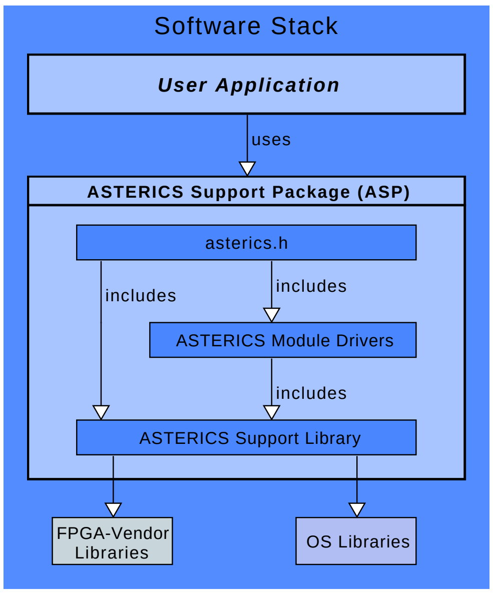


Figure 8. Software stack of ASTERICS drivers.

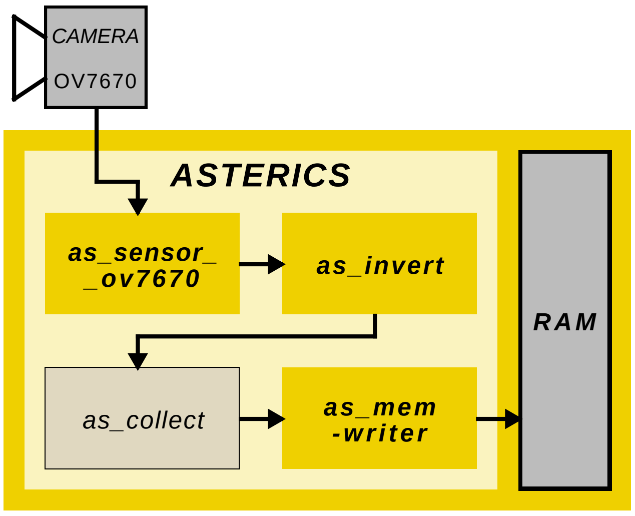


Figure 9. Block graph of the pixel inverter system described by Listing 1.

in Listing 1, also shown as a block graph in Figure 9. This functionality allows developers to quickly verify the *Automatics* script. Besides only showing the user-added processing modules, the graph output can be enriched by management components added by *Automatics*, external inputs and outputs and port names, useful for debugging or when working on HDL level on an ASTERICS system.

C. *Defining Custom Processing Modules*

Besides the processing modules available with ASTERICS, developers may also add their own processing modules. For each execution of *Automatics*, all available modules are analysed and imported, using a short specification script written

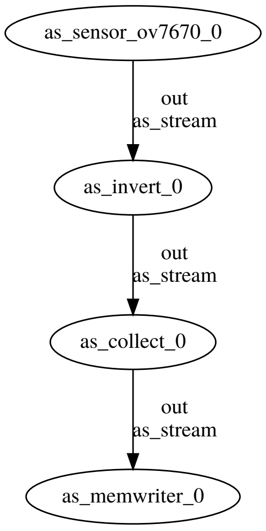


Figure 10. Vector graphic graph representation of the pixel inverter system generated by *Automatics*.

in Python. As an example, Listing 2 shows the specification script for the module `as_memwriter`.

Essentially, the developer has to specify the following three things:

- **Line 7, 8:** The main VHDL file of the module, which defines its VHDL entity (toplevel file).
- **Line 9-11:** Other HDL files this module depends on.
- **Line 12, 13:** Other modules this module depends on.

In lines 16 and 17 the method `discover_module` is called, starting the VHDL analysis of the toplevel file specified for this module. In this step, all other metadata used later by *Automatics* is generated automatically, mainly a list of all VHDL ports of the module. From the list of ports interfaces are inferred using a user-extendable list of interface templates. The basis of this operation is the names of the VHDL ports, which are split into a base name, prefixes and suffixes and the port direction and data type.

```
1   # Import Automatics
2   from as_automatics_module import AsModule
3
4   # Module definition function
5   def get_module_instance(module_dir):
6     module = AsModule()
7     toplevel_file = \
8       "hardware/hdl/vhdl/as_memwriter.vhd"
9     module.files = \
10      [("hardware/hdl/vhdl/"
11        "as_mem_address_generator.vhd")]
12    module.dependencies = \
13      ["as_regmgr", "helpers", "fifo_fwft"]
14
15    # Run analysis and return the module object
16    module.discover_module( \
17      module_dir + "/" + toplevel_file)
18    return module
```

Listing 2. The module specification script of the `as_memwriter` module.

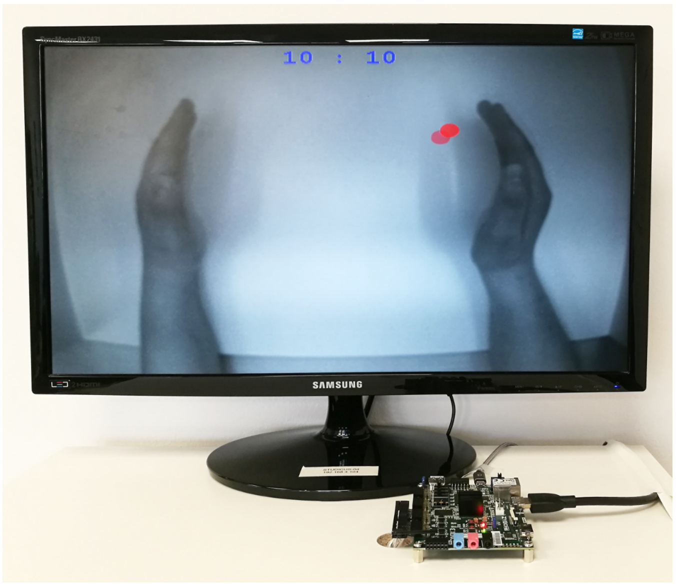


Figure 11. Demonstration of an augmented reality ASTERICS System.

VI. Automatics in Education

ASTERICS in combination with *Automatics* is used actively to convey concepts of hardware design and image processing in bachelor courses at the University of Applied Sciences Augsburg.

For example, ASTERICS is used in a system and logic design course where, within three lab exercises of four hours each, students build an augmented reality game "Pong-on-a-Chip", similar to the system shown in Figure 11. The system includes an ASTERICS chain to generate edges from the camera image, at which the virtual pong ball is reflected.

The lab project is implemented using a Zybo board with a Xilinx Zynq-7010 SoC device [17]. The system comprises an ASTERICS chain which the students extend with their own module for edge detection. The students write their own software to animate the ball and let it change direction based on the edge data delivered by the edge detection module. For this, the VEARS visualization module and its graphics library, which is also part of the ASTERICS framework, is used, running on the ARM processor of the SoC.

Within this course the students learn:

- How to create SoC designs using the Xilinx Vivado toolchain.
- How to do and practice hardware- software co-design, implementation and debugging techniques using SoCs.
- The basics of hardware accelerated image processing.
- How to integrate custom hardware into a larger project.
- How to use and reuse existing hardware and software components.

Throughout the course, *Automatics* helps to hide some of the organizational tasks that would have to be done to configure and build the ASTERICS IP-Core, enhancing the learning experience and making the ASTERICS framework a more valuable tool for education.

VII. Experimental Results

For a tool to be useful to build prototypes of systems, it must have a short enough runtime, where acceptable runtimes vary from application to application. Synthesis tools for FPGAs and ASICs have rather long execution times, increasing with the complexity and size of the project, but generally range from a few minutes to one or more hours.

The execution times of *Automatics* and the Xilinx Vivado toolchain (version 2017.2) have been measured and are shown in Table I. All runtime measurements are made on the same hardware platform: A notebook running an Intel Core i7-5500U mobile dual-core processor with SMT and all data stored on an SSD. The test project includes the ASTERICS system shown in Figure 1 and a VEARS IP-Core. The `myfilter` module used for this systems contains a pipelined 7x7 box filter. Furthermore, the system comprises two AXI management IP-Cores, an AXI IIC master and three AXI GPIO IP-Cores. The hardware target is the low-end Zybo development board integrating a XC7Z010 FPGA-SoC. The entire system uses 42% of all available slice LUTs, 22% of all slice registers and 2.5% of embedded RAM (Block RAM).

Table I shows average runtimes for the various synthesis and compilation steps. The entire build process is run in a terminal, without opening the Vivado GUI. The software project setup is done using the Xilinx *Hardware Software Interface* (HSI) tool in terminal mode. The Board Support Package for the hardware target and a minimal bare-metal user application are compiled for the project using an ARM GCC cross compiler of version 4.9.3.

Table I
Runtime measurements building the "myfilter" system.

Tool	**Build Step**	**Runtime [s]**
Automatics	Generate ASTERICS output products	0.18
Vivado	ASTERICS IP-Core Packaging	10.8
Vivado	Build Block Design	19.3
Vivado	Synthesize IP-Cores & System	461
Vivado	Implementation	130
HSI & GCC	Setup and compile Software Project	15.7

The results show that even for a small FPGA design the runtime of *Automatics* is negligible compared to the execution times of the rest of the toolchain. Therefore, *Automatics* is well suited to accelerate the design and development process.

Furthermore, *Automatics* is able to verify some configuration options of processing modules in the image processing system. Specifically, options that pertain to data vector widths of processing modules, can be verified and mismatches are reported. In cases where the solution to the mismatch is unambiguous, *Automatics* can automatically apply a fix. In all other cases, the process is stopped before the output products are generated and all encountered errors are reported. Catching these errors early on, instead of in the middle of a lengthy synthesis run, can greatly reduce the time spent debugging the hardware design and further speed up development.

VIII. Conclusion and Future Work

ASTERICS is a framework for image processing on FPGAs, introduced in concept and practice. The new system generator *Automatics* enables developers to describe image processing systems on a higher abstraction level via a concise textual input method using Python syntax. The input method is simple enough to allow rapid prototyping of new systems with little effort. This has enabled the ASTERICS framework to be used for interactive teaching on image and video processing on FPGAs and embedded systems. Developers are able to use new custom modules with the generator by adding a Python script to the hardware description, providing only some basic pieces of metadata. *Automatics* has a very short runtime and allows developers to catch certain errors in the hardware configuration early in the development process, thus contributing to a more rapid development cycle.

Ongoing and future work concentrates on extending *Automatics* towards window filter modules and support for artifical neural networks. The range of build targets available in *Automatics* is planned to be expanded by support for Intel FPGAs and an optional Linux driver. Likewise, the ASTERICS framework is continuously expanded by support for more FPGA and FPGA-SoC platforms and additional image processing modules, as well as more example and reference systems.

Acknowledgment

Part of this work has been supported by the German Federal Ministry for Economic Affairs and Energy, grant number ZF4102001KM5.

References

[1] D. G. Bailey, C. T. Johnston, and K. T. Gribbon. "Implementing Image Processing Algorithms on FPGAs". In: *Proceedings of the Eleventh Electronics New Zealand Conference*. Citeseer, 2004, pp. 118–123.

[2] D. H. Ballard. "Generalizing the Hough Transform to Detect Arbitrary Shapes". In: *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 714–725. ISBN: 0934613338.

[3] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool. "SURF: Speeded Up Robust Features". In: *Computer Vision and Image Understanding (CVIU)* 110.3 (2008), pp. 346–359.

[4] C. Desmouliers, E. Oruklu, and J. Saniie. "FPGA-based design of a high-performance and modular video processing platform". In: *2009 IEEE International Conference on Electro/Information Technology* (2009), pp. 393–398. ISSN: 2154-0357. DOI: 10.1109/EIT.2009.5189649.

[5] N. Faroughi. "An image processing hardware design environment". In: *Proceedings of 40th Midwest Symposium on Circuits and Systems. Dedicated to the Memory of Professor Mac Van Valkenburg*. Vol. 2. 1997, pp. 1225–1228. DOI: 10.1109/MWSCAS.1997.662301.

[6] E. Gudis, P. Lu, D. Berends, et al. “An Embedded Vision Services Framework for Heterogeneous Accelerators”. In: *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops* (2013), pp. 598–603. ISSN: 2160-7516. DOI: 10.1109/CVPRW.2013.90.

[7] J. Hammes, B. Rinker, W. Bohm, et al. “Cameron: high level language compilation for reconfigurable systems”. In: *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425).* 1999, pp. 236–244. DOI: 10.1109/PACT.1999.807557.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. Chap. 7, pp. 540–544, 557–606. ISBN: 0128119055, 9780128119051.

[9] P. V. C. Hough. “Method and means for recognizing complex patterns”. U.S. pat. 3069654A. Dec. 1962.

[10] G. Kiefer, M. Vahl, J. Sarcher, and M. Schaeferling. “A configurable architecture for the generalized hough transform applied to the analysis of huge aerial images and to traffic sign detection”. In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016, pp. 1–7. DOI: 10.1109/ReConFig.2016.7857143.

[11] K. Konstantinides and J. R. Rasure. “The Khoros software development environment for image and signal processing”. In: *IEEE Transactions on Image Processing* 3.3 (1994), pp. 243–252. ISSN: 1057-7149. DOI: 10.1109/83.287018.

[12] P. Manke and G. Kiefer. “Software Tool for the Automated Generation of Image Processing Systems for FPGAs Using the ASTERICS Framework”. In: *Applied Research Conference 2019*. 2019. ISBN: 978-3-96409-182-6.

[13] M. Motomura. “A Dynamically Reconfigurable Processor Architecture”. In: *Proc. 2002 Microprocessor Forum* (2002), pp. 2–4.

[14] M. Motomura. “STP Engine, a C-based Programmable HW Core featuring Massively Parallel and Reconfigurable PE Array: Its Architecture, Tool, and System Implications”. In: *Proc. Cool Chips XII* (2009), pp. 395–408.

[15] M. Pohl, M. Schaeferling, and G. Kiefer. “An efficient FPGA-based hardware framework for natural feature extraction and related Computer Vision tasks”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927463.

[16] M. Pohl, M. Schaeferling, G. Kiefer, et al. “An efficient and scalable architecture for real-time distortion removal and rectification of live camera images”. In: *2012 International Conference on Reconfigurable Computing and FPGAs* (2012), pp. 1–7. ISSN: 2325-6532. DOI: 10.1109/ReConFig.2012.6416730.

[17] Digilent Inc. *Zybo Development Board Reference*. Dec. 2019. URL: https://reference.digilentinc.com/reference/programmable-logic/zybo/start.

[18] EES research group. *Efficient Embedded Systems Homepage for ASTERICS*. Dec. 2019. URL: https://ees.hs-augsburg.de/asterics.

[19] Renesas Electronics Corporation. *Renesas Homepage*. Dec. 2019. URL: https://www.renesas.com.

[20] Silicon Software GmbH. *Silicon Software Homepage*. Dec. 2019. URL: https://silicon.software.

[21] H. Sahlbach, D. Thiele, and R. Ernst. “A system-level FPGA design methodology for video applications with weakly-programmable hardware components”. In: *Journal of Real-Time Image Processing* 13.2 (2017), pp. 291–309. ISSN: 1861-8219. DOI: 10.1007/s11554-014-0403-4. URL: https://doi.org/10.1007/s11554-014-0403-4.

[22] J. Sarcher, C. Scheglmann, A. Zoellner, et al. “A Configurable Framework for Hough-Transform-Based Embedded Object Recognition Systems”. In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2018, pp. 1–8. DOI: 10.1109/ASAP.2018.8445086.

[23] M. Schaeferling, M. Bihler, M. Pohl, and G. Kiefer. “ASTERICS - An Open Toolbox for Sophisticated FPGA-Based Image Processing”. In: *embedded world Conference 2015 - Proceedings*. 2015.

[24] M. Schaeferling, J. Sarcher, A. Zoellner, P. Manke, and G. Kiefer. *The ASTERICS Book*. 2019. URL: https://ees.hs-augsburg.de/asterics.

[25] Wikipedia. *Pixel Visual Core*. Dec. 2019. URL: https://en.wikipedia.org/wiki/Pixel_Visual_Core.