5-2015

Design and Verification Environment for High-Performance Video-Based Embedded Systems

Michael Mefenza Nentedem
University of Arkansas, Fayetteville

Design and Verification Environment for High-Performance Video-Based Embedded Systems

Design and Verification Environment for High-Performance Video-Based Embedded Systems

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Engineering

by

Michael Mefenza Nentedem
Ecole Nationale Superieure Polytechnique
Bachelor of Science in Telecommunication Engineering, 2009
Ecole Nationale Superieure Polytechnique
Master of Science in Telecommunication Engineering, 2010

May 2015
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

Dr. Christophe Bobda
Dissertation Director:

Dr. David Andrews
Committee member

Dr. Scott Smith
Committee member

Dr. Pat Parkerson
Committee member

Abstract

In this dissertation, a method and a tool to enable design and verification of computation demanding embedded vision-based systems is presented. Starting with an executable specification in OpenCV, we provide subsequent refinements and verification down to a system-on-chip prototype into an FPGA-Based smart camera. At each level of abstraction, properties of image processing applications are used along with structure composition to provide a generic architecture that can be automatically verified and mapped to the lower abstraction level. The result is a framework that encapsulates the computer vision library OpenCV at the highest level, integrates Accelera's System-C/TLM with UVM and QEMU-OS for virtual prototyping and verification and mapping to a lower level, the last of which is the FPGA. This will relieve hardware designers from time-consuming and error-prone manual implementations, thus allowing them to focus on other steps of the design process. We also propose a novel streaming interface, called Component Interconnect and Data Access (CIDA), for embedded video designs, along with a formal model and a component composition mechanism to cluster components in logical and operational groups that reduce resource usage and power consumption.

This work was supported in part by the grant 1302596 from the National Science Foundation (N.S.F.)

Acknowledgements

I would like to extend my sincere gratitude to my advisor Dr. Christophe Bobda. His guidance and support have been invaluable and his belief in me unfaltering. I cannot ever hope to repay the time and efforts that he devoted to my growth both academically and as a person; I can only hope that I will have the opportunity to pay this debt forward to a colleague or a student. I would like to thank Dr. David Andrews, Dr. Pat Parkerson and Dr. Scott Smith for serving as members of my dissertation committee. Their questions, comments and suggestions have helped me distill the arguments and clarify the exposition. I appreciate the time and efforts that they put in reading and evaluating my dissertation and presentations. I would also like to acknowledge that this work was supported in part by the grant 1302596 from the National Science Foundation (N.S.F.). Last but not least, I would like to thank my family for supporting me and always being there when I needed help. I would like to thank my parents Mr. Nentedem Pierre and Mme Megni Louise for their support and encouragement.

Table of Contents

List of Figures

List of Tables

List of Papers

This dissertation is based on the following four papers:

Chapter 2 **Component Interconnect and Data Access Interface for Embedded Vision Applications**
Michael Mefenza, Franck Yonga and Christophe Bobda
In Journal of Real-Time Image Processing. Submitted for review.

Chapter 3 **A Framework for Rapid Prototyping of Embedded Vision Applications**
Michael Mefenza, Franck Yonga, Luca B Saldanha, Christophe Bobda and Senem Velipassalar
In Conference on Design & Architectures for Signal and Image Processing (DASIP), Oct 2014, Madrid, Spain. Accepted.

Chapter 4 **Automatic UVM Environment Generation for Assertion-based and Functional Verification of SystemC Designs**
Michael Mefenza, Franck Yonga and Christophe Bobda
In Conference on Microprocessor Test and Verification (MTV 2014), Dec 2014, Austin, Texas, USA. Accepted.

Chapter 5 **Interface Based Memory Synthesis Of Image Processing Chains In FPGA**
Michael Mefenza and Christophe Bobda
In International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART 2015). Submitted for review.

Other related papers:

A Hardware/software Prototyping System for Driving Assistance Investigations
Jacob Anders, Michael Mefenza, Christophe Bobda, Franck Yonga,

Z. Aklah and Kevin Gunn.

In Journal of Real-Time Image Processing, pages 1-11, May 2013. [Online]. Available: http://dx.doi.org/10.1007/s11554-013-0351-4. Contributed to chapter 2.

RazorCam: A Prototyping Environment for Video Communication

Michael Mefenza, Franck Yonga and Christophe Bobda

In International Workshop on Mobile Computing Systems and Applications, 17(3):13-14, February 2013. [Online]. Available: http://doi.acm.org/10.1145/2542095.2542103. Contributed to chapter 3

I Introduction

I.1 Video and image processing

The use of cameras, and thus visual detection, has become a promising alternative conventional range sensors due to their advantages in size, costs and accuracy. Robotics, driving assistance systems, autonomous driving cars and unmanned aerial systems (UAS) are few examples of critical areas of application. Progress in automotive is leading to the use of cameras in high-end cars for driving assistance, lane departure warning, autonomous cruise control and occupant pose analysis [1]. UAS are increasingly used in surveillance [2], [3], precision agriculture [3], search and rescue [4], [3] and communications relay [5], [3]. Many research efforts have also been devoted to building systems for vision-aided flight control [6], tracking [7], terrain mapping [8], and navigation [9]. Image processing can be described as a task which converts an input image into a modified output image or a task that extract information from the features present in an image. A typical set-up of an image processing system includes an image acquisition device and the image processing unit.

I.1.1 Image acquisition

Video images are captured by CMOS or CCD (charge-coupled device) image sensors. These are semiconductor devices comprising an array of light sensitive elements which convert photon intensity into electric charge. In most cases the sensing element responds to intensity only; color images are captured by passing the light through a mosaic of red, green and blue filters before sampling, such that each element captures one primary color only. The sensor produces a frame by spatially dividing a light sensitive region into an ordered array of picture elements referred to as pixels, which are aligned to a grid or lattice, with M rows and N columns. The size (M X N) of the image is defined as the number of pixels per frame. A common size for video derived from analogue sources is 720 x 576 whereas for digitally sourced video, it is typically a minimum of 640 x 480 in most applications.

I.1.2 Image processing algorithms

There are several types of image processing algorithms depending on the end use of the video stream. Algorithms range from low-level processing, whereby operations are performed uniformly across a complete image or sequence, to high-level procedures such as object tracking and identification. Low-level techniques are generally highly parallel, repetitive and require high throughput, making them attractive for implementation in hardware. Moreover, operations are generally a function of a localized contiguous neighborhood of pixels from the input frame, which can be exploited in data reuse schemes. Note that the serialization of video frames using raster-scanning means that significant portions of the video stream may need to be stored, despite the data locality of a particular algorithm. Examples of image processing algorithms include image segmentation, noise elimination and morphological erosion and dilation. These operations find applications in robot vision and machine vision. Image processing algorithms can be classified in 4 categories which are applied to alter a pixel of a set of values to make an image more suitable for subsequent operations:

- Point operations
- Global operations
- Neighborhood operations
- Temporal operations

These categories cover simple algorithm operations. Many higher-level algorithms may be formed from combinations of operations from these five categories. An important requirement of video processing is the real-time i.e. the processing should be fast enough so that the result is meaningful for the user. For example, the processing in a video-based autonomous car should be fast enough in detecting pedestrians so that the car can be able to avoid them. Real-Time is the term used to describe a class of video processing system in which the video signal is processed at the rate of video capture such that the rate of generating output pixels matches the rate of receiving input pixels. Real-time video processing is computationally demanding but often highly parallelizable, making it amenable to hardware implementations.

I.1.3 Image processing implementation

Image processing algorithms are usually written in high level languages such as OpenCV, C++ and executed within software based processors such as General Purpose Processors (GPP) and Digital signal processors (DSP). A GPP typically has a generic instruction set that is not optimized for any particular application [10]. There are several issues limiting their applicability in real-time processing systems. The main issue is due to the fact that a GPP does not offer any type of specialized hardware support for specific or repetitive operations found in digital processing algorithms. A DSP, on the other hand, has a specialized instruction set with dedicated hardware support for operations commonly used in digital signal processing algorithms [11]. As a consequence, this type of implementation platform has a better throughput but lacks the flexibility of programmable hardware processors such as Application Specific Integrated Circuits (ASICs) and Field programmable gate arrays (FPGA). ASICs are fabricated and made for special or dedicated applications. This means that their precise functions and performance are considered and fully analyzed before fabrication. The consequence is efficiency, reliability and high performance. However, changes in system requirements which might be due to an oversight or a changing system demands results in a complete replacement of the device because the architecture in ASICs cannot be altered. An FPGA is a reconfigurable implementation platform which typically consists of logic blocks, interconnects (routing), and I/O blocks [12]. An FPGA also offers the possibility of exploiting parallelism, resulting in an increased performance compared to GPPs and DSPs. Compared to ASIC, FPGA technology offers flexibility and rapid prototyping capabilities in favor of faster time to market. A design concept can be tested and verified in hardware without going through the long fabrication process of custom ASIC design. You can then implement incremental changes and iterate on an FPGA design. For these reasons, FPGA are used as alternative for video and image processing systems.

I.2 FPGA

Field programmable gate arrays (FPGA) are configurable integrated circuits containing programmable logic that can be used to design digital circuits. Modern FPGAs acting as true system-on-chip (SoC) devices with integrated memory, microprocessors, digital signal processing (DSP) elements, high-speed transceivers, clock management, and numerous other features. The basic elements of FPGAs are configurable logic blocks (CLB) connected together via a hierarchy of routing resources and programmable switch matrices. Each CLB contains a relatively small amount of memory and some logic resources that may be programmed to implement the desired function, with the memory acting as a look-up table (LUT), RAM, or a shift register. When configured as a LUT it may be used to replicate combinatorial and sequential logic, and CLBs may be chained together to implement logic functions of any size. LUTs provide the main resource for implementing logic functions. LUTs can also be configured as a Distributed RAM or as a 16-bit shift register [13]. The storage elements can be programmed as either a D-type flip-flop or a level-sensitive latch in order to provide a means of synchronizing data to a clock signal. Wide-function multiplexers effectively combine LUTs in order to permit more complex logic operations. The carry chain, together with various dedicated arithmetic logic gates, supports rapid and efficient implementations of mathematical operations. To enable connections between logic elements themselves and between logic elements and any other parts of the chip, the FPGA contains the interconnect. It is an important feature that ultimately determines device performance. It is essentially a network of interconnecting wires with switching matrices at crossover points comprised of pass-transistors and multiplexers programmable routing resources take up a large proportion [13]. Interconnects provide the mechanism for routing signals between logic cells, memory blocks, DSP blocks and I/O pins inside the FPGA. Interconnects are usually optimized for efficient signal transport based on the signal frequency and the distance between the signal source and the sink to ensure predictability, signal integrity and performance repeatability. In addition to these basic components, on-chip blocks of memory are also provided. Many of the FPGA designs require some kind of fast memory for temporary storage of intermediate results, data buffers and other. For this reason, the chip contains embedded memory blocks. These are hardened SRAM memory units, usually configurable for different memory sizes, data widths or single/dual port access. The reconfigurability feature as well as hardware parallelism of the FPGAs offers significant advantages in many applications. However, there are a number of challenges to system development particularly in the field of video and image processing.

I.3 FPGA for rapid prototyping of embedded video applications

Embedded video systems are usually composed of deeply integrated hardware and software components to achieve complexity and performance. Low-level repetitive computations on huge amounts of data are mapped into hardware, while complex reasoning parts are maintained in software. There are several challenges to embedded video system development in FPGAs. Some of these challenges include the design reuse, design flow, resources usage and design verification. They will be discussed in the following sections.

I.3.1 Design reuse

Design reuse is the use of a library of intellectual property (IP) cores to build a desired circuit. The library implement functions of high complexity (e.g., a DCT or a filter) and systems are built bottom-up by selecting designs from the library and connecting them together. Bottom-up block-based design reuse has limited scalability due to the design cost of integration. System-level interconnect and logic must be custom designed for each implementation, the complexity of which grows exponentially with block number. Functional and performance verification are difficult. These issues are addressed by using a standardized communication architecture, such as Open Core Protocol (OCP) [23], WISHBONE [8], AMBA [4]. Standardizing block interfaces precludes compatibility issues; while the communication architecture implementation is itself a parameterized circuit which can be reused. The block design process can be simplified as well by isolating the interaction between blocks from their functionality. By using pre-designed components, the main design task moves from designing components from scratch and interconnecting them to simple integration of existing IPs. Unfortunately, this process can be very difficult and integration burdens can quickly offset the benefit of IP-reuse. IPs are available with various interfaces, which limit data exchange among interface with different communication protocols and data access mechanism. Research in interface synthesis has sought to automatically generate glue-logic between components with different interfaces. The lack of formalism and standard in existing interfaces as well as the infinite number of potential protocols and communication mechanism makes it impossible for a single tool to target the general purpose case and provide a universal synthesis methodology. Standard interfaces such as Avalon-streaming [1] from Altera, AXI4-Stream [25] from Xilinx among others, lack the formalism required to capture all facets of the interface. They often lead to a poor timing, higher resource usage and higher power consumption; the main reason being their general purpose orientation.

I.3.2 Design flow

Software design environments, such as OpenCV, are very popular in the software community for the design of video-based systems. While those frameworks increase the productivity by providing a rich set of library function for image and video manipulation and machine learning, they are limited to target only general purpose processors. As a consequence, there is a need to map applications captured in those framework onto dedicated hardware/software architecture while performing verification tasks. Manual translations would be time consuming, error-prone, and would require hardware design skills not available in the image processing community, the bulk of which is made upon software designers. In some systems the hardware part is fixed, but the increasing demands on performance and quality requires more and more that the hardware fulfills very special requirements and that it is precisely adjusted to the embedded software. As a consequence, it is often no longer sufficient to use prefabricated hardware components. Traditionally, hardware devices are implemented by low-level coding in hardware description language (HDL). This approach is very remote from the high level specification tool and can be a very tedious task and need special expertise for image processing algorithms. An easier path is to use high level languages (HLL). These include C/C++, Java, MATLAB [8]. many works have focused on synthesizing hardware from C. De Micheli [14] summarized the major research contribution in the use of C/C++ for hardware modeling and synthesis while Edwards [15] provided in detail, challenges to hardware synthesis from C-based languages. It was observed that the approach generates inefficient hardware due to difficulties in specifying or inferring concurrency, time, type and communication in C and its variants. To these ends, modeling languages such as SystemC [1] have been optimized to efficiently overcome some of these shortcomings (for example, both handling concurrency through process-level parallelism) and are often employed to capture the system behavior in the form of executable specifications. The SystemC library has layers of increasing abstraction, enabling hardware to be modeled at different levels. In addition, since systems are usually composed of deeply integrated hardware and software components to achieve complexity and performance, these components must be designed together. This leads to hardware/software co-design. An essential component of co-design methods is HW/SW co-simulation, which is necessary to evaluate and compare different design alternatives. In a HW/SW co-simulation, hardware and software parts of a system are simulated together. This provides an integrated way to simulate the interactions between hardware and software. The main challenge in HW/SW co-simulation is that hardware and software designers talk in different languages. They use different abstraction levels, different models of computation, different programming languages and different tools. These differences make it complicated to bring the design processes together and to unify them in a single co-simulation framework.

I.3.3 Resources usage

The essential resources on FPGAs are arithmetic and logic resources, embedded memory and logic cells. They are available in an optimized form but in limited amounts [8]. It is necessary to have a balanced usage of these resources in an application in order to avoid a shortage of one type of resource while having an excess of others. For instance, an example temporal algorithm would be to detect motion by subtracting a frame from the previous one. This would require frame buffering, and it is clear that whenever processing is required that utilizes the temporal dimension of video data the storage requirements increase rapidly. Alternatively, global operations involve high speed processing, as multiple passes through the image data will usually be required; this will increase the memory requirement and pose implementation challenges in timely processing of the data. It is clear that on-chip memory is an important resource that can quickly become scarce. This kind of temporal processing, where data sets are comprised of elements that do not arrive in sequence but are distributed in time, is common in many video applications but because of high data rates can often involve a significant requirement for memory resources.

I.3.4 Verification

Video-based autonomous systems are used in critical areas such as unmanned aerial systems (UAS) and autonomous vehicles with high safety standards, which can only be provided by a sound verification process. As FPGA capabilities and design complexities increase, verification and simulation also become more complex. Verification of logic designs is at present carried out predominantly through RTL simulation, using event-driven HDL simulators. It is the responsibility of the designer to produce testbenches that correctly drive the simulation software and cover a sufficient range of test cases to ensure the original specification is being met. Due to the fact that testbenches are not synthesized the full extent of VHDL or Verilog instructions may be used, which provides a considerable number of additional capabilities over HDL that is to be synthesized, but designing the testbench and performing the simulation is still a lengthy and complicated process.

I.4 Objective

It is therefore imperative to provide an automatic approach for the translation process, while minimizing the resources involved and insuring correctness of design through verification. This can be addressed by exploiting:

- High-level system modeling: to define concepts necessary for modeling embedded vision-based systems through SystemC/TLM.
- Transformations: to develop necessary model to model transformation rules, in order to allow subsequent refinements down to the hardware/software implementation.
- Verification and analysis: to develop methods to verify and analyze models, in order to guarantee that the final implementation corresponds with the initial system specification.

The objective of this dissertation is: *to demonstrate the feasibility of a high-level framework for the rapid design and verification of embedded vision-based systems. The approach allows capturing computer vision application at a highest abstraction-level with subsequent refinements and verification down to the hardware/software implementation.*

I.5 Contributions

The contributions of this dissertation are as follows:

- A novel streaming interface, called Component Interconnect and Data Access (CIDA), for embedded video designs, along with a formal model and a component composition mechanism to cluster components in logical and operational groups that reduce resource usage and power consumption.

- A design methodology for rapid prototyping of system-on-chip with emphasis of embedded video applications. It leverages existing tools and provides a means to facilitate their integration toward a semi-automatic mapping of software specification to hardware/software implementations.

- A verification flow to perform Assertion-based and coverage validation on SoC design at IP level.

These contributions will be discussed at a later stage together with the results obtained by their use. Tests on the performance of the proposed solutions and comparisons with other works are also discussed and were published.

I.6 Dissertation outline

The remainder of this dissertation is organized as follows. Chapter II presents an interface methodology to address design reuse problems and a comparison with other interfaces. In chapter III, our design methodology for rapid prototyping of embedded video applications is presented and the verification flow is described in chapter IV. In chapter V, memory synthesis approach is presented for resource optimization. Finally, chapter VI presents conclusions drawn from this research, as well as possible future work.

Bibliography

[1] A. Wilson, “Auto cameras benefit from cmos imagers,” 2009. [Online]. Available: http://www.vision-systems.com/display_article/228883/19/none/none/Feat/Auto-cameras-benefit-from-CMOS-imagers

[2] G. Cai, B. Chen, K. Peng, M. Dong, and T. Lee, “Modeling and control of the yaw channel of a uav helicopter,” *Industrial Electronics, IEEE Transactions on*, vol. 55, no. 9, pp. 3426–3434, 2008.

[3] O. Špinka, O. Holub, and Z. Hanzálek, “Low-cost reconfigurable control system for small uavs,” *Industrial Electronics, IEEE Transactions on*, vol. 58, no. 3, pp. 880–889, 2011.

[4] B. Ludington, E. Johnson, and G. Vachtsevanos, “Augmenting uav autonomy,” *Robotics Automation Magazine, IEEE*, vol. 13, no. 3, pp. 63–71, 2006.

[5] M. Campbell and W. Whitacre, “Cooperative tracking using vision measurements on seascan uavs,” *Control Systems Technology, IEEE Transactions on*, vol. 15, no. 4, pp. 613–626, 2007.

[6] N. Guenard, T. Hamel, and R. Mahony, “A practical visual servo control for an unmanned aerial vehicle,” *Robotics, IEEE Transactions on*, vol. 24, no. 2, pp. 331–340, 2008.

[7] F. Lin, K.-Y. Lum, B. Chen, and T. Lee, “Development of a vision-based ground target detection and tracking system for a small unmanned helicopter,” *Science in China Series F: Information Sciences*, vol. 52, no. 11, pp. 2201–2215. [Online]. Available: http://dx.doi.org/10.1007/s11432-009-0187-5

[8] M. Meingast, C. Geyer, and S. Sastry, “Vision based terrain recovery for landing unmanned aerial vehicles,” in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, vol. 2, 2004, pp. 1670–1675 Vol.2.

[9] J. Kim and S. Sukkarieh, “Slam aided gps/ins navigation in gps denied and unknown environments,” in *The 2004 International Symposium on GNSS/GPS, Sydney*, 2004, pp. 6–8.

[10] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier, 2012.

[11] P. Lapsley, J. Bier, A. Shoham, and E. Lee, “Dsp processor fundamentals,” *IEEE SPECTRUM*, 1998.

[12] W. J. MacLean, "An evaluation of the suitability of fpgas for embedded vision systems," in *Computer Vision and Pattern Recognition-Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on.* IEEE, 2005, pp. 131–131.

[13] S. Mirzaei, "Design methodologies and architectures for digital signal processing on fpgas," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA SANTA BARBARA, 2010.

[14] G. De Micheli, "Hardware synthesis from c/c++ models," in *Proceedings of the conference on Design, automation and test in Europe.* ACM, 1999, p. 80.

[15] S. A. Edwards, "The challenges of hardware synthesis from c-like languages," in *Design, Automation and Test in Europe, 2005. Proceedings.* IEEE, 2005, pp. 66–67.

II Component Interconnect and Data Access Interface for Embedded Vision Applications

Michael Mefenza, Franck Yonga and Christophe Bobda

Abstract— IP-based design is used to tackle complexity and reduce time-to-market in Systems-on-Chip with high-performance requirements. Component integration, the main part in this process, is a complicated and time-consuming task, largely due to interfacing issues. Standard interfaces can help to reduce the integration efforts. However, existing implementations use more resources than necessary and lack of a formalism to capture and manipulate resource requirements and design constraints. In this paper, we propose a novel interface, the Component Interconnect and Data Access (CIDA), and its implementation, based on the interface automata formalism. CIDA can be used to capture system-on-chip architecture, with primarily focus on video-processing applications, which are mostly based on data streaming paradigm, with occasional direct memory accesses. We introduce the notion of component-interface clustering for resource reduction and provide a method to automatize this process. With real-life video processing applications implemented in FPGA, we show that our approach can reduce the resource usage (#slices) by an average of 20% and reduce power consumption by 5% compared to implementation based on vendor interfaces.
Keywords— Interface Formalism, Computer Vision, Functional Verification, FPGA.

II.1 Introduction

Embedded vision applications are increasingly complex, in part because of the huge amount of functionality required by customers, the huge amount of data delivered by high-density sensors and complex computation to apply on those data in real-time. With the well established component-based design approach, complexity can be tackled and time-to-market of embedded vision applications reduced by using pre-designed and pre-verified components to assemble large and complex systems. Using off-the-shelf components, the main design task moves from designing components from scratch and interconnecting them to simple integration of existing IPs. Unfortunately, this process can be very difficult and integration burdens can quickly offset the benefit of IP-reuse. IPs are available with various interfaces, which limit data exchange among interface with different communication protocols and data access mechanism. One way to address this issue is to insert protocol transducers or wrappers between IPs with incompatible protocols. The result is an increase in resource usage and design effort, with a negative impact on time-to-market. Research in interface synthesis has sought to automatically generate glue-logic between components with different interfaces. The lack of formalism and standard in existing interfaces as well as the infinite number of potential protocols and communication mechanism makes it impossible for a single tool to target the general purpose case and provide a universal synthesis methodology. Standardization is the path adopted by companies to address the interfacing issue. It forces IP designers to use a well described and implemented interface, thus making the integration easier. However, standard interfaces such as Avalon-streaming[1] from Altera, AXI4-Stream[25] from Xilinx among others, lack the formalism required to capture all facets of the interface. They often lead to a poor timing, higher resource usage and higher power consumption; the main reason being their general purpose orientation.

To address these issues, we propose a novel streaming interface, called Component Interconnect and Data Access (CIDA), for embedded video designs, along with a formal model and a component composition mechanism to cluster components in logical and operational groups that reduce resource usage and power consumption. Even though CIDA can be used for the design of any Systems-on-Chip, design efforts were made for the use in video processing applications, which usually require access to image data from various sizes and various sources. Local access is performed to access local buffers, neighbor processed data, and global memory data, all of which represent part of, or entire images at different levels of the processing chain. With real-life video processing applications, our approach was able to achieve a reduction of 20% in resource consumption and 5% in power consumption with a considerable reduction in the overall design time.

The rest of the paper is organized as follows: Section II.2 introduces a motivation example for a better understanding of the problem addressed in this work. Related research regarding streaming interfaces and component-based design methodology is discussed in section II.3. We present our proposed interface formalism in section II.4, followed by its implementation in section II.5. Section II.6, section II.7 and section II.8 respectively presents system integration, resource optimization and functional verification support, Experimental results with real-life applications implemented in FPGAs are provided in section II.9. Finally, section II.10 concludes the paper.

II.2 Motivation example

Generally, interfaces are separated from the core functions, so the IP core can be easily and quickly integrated into different system platforms utilizing different protocols, by simply changing the interface logic wrapper without altering the core logic function. Our main objective is to provide a formalized streaming interface with a means to minimize logic resources and the total area by appropriately composing components using that interface. The formalism is used to describe and manipulate the constraints under which independently developed components can properly operate together. Furthermore, it helps specifying resource optimization and verification objectives, thus simplifying the use of external tools for optimization.

Consider the example of figure II.1 where 3 components with registered-input and output, using the same interface with different component-logic. Because component-based design sees the components as separated entities with separate interface, the 3 components can be connected serially, as in figure II.1-a, thus using three times the same interface. Clustering 2 of the 3 components (Figure II.1-b) leads to less resource usage. The main reason is that the interface of a clustered group of components usually uses less amount of resources than the separated implementation. Reducing resource usage will lead to a reduction in power consumption and sometimes to an improvement of the timing, as many of our experiments have confirmed. Figure II.1-b and figure II.1-c show different clustering possibilities, all leading to different results. We intend in this paper to study the impact of different configurations on the overall system.

II.3 Related Work

The bandwidth requirements of video processing applications can be achieved using a stream processing model where applications data are organized as streams of data which flow through a composition of producer/consumer components using a streaming interface. A streaming interface description concentrates on the input/output behavior abstracting it from the component's internal structure.

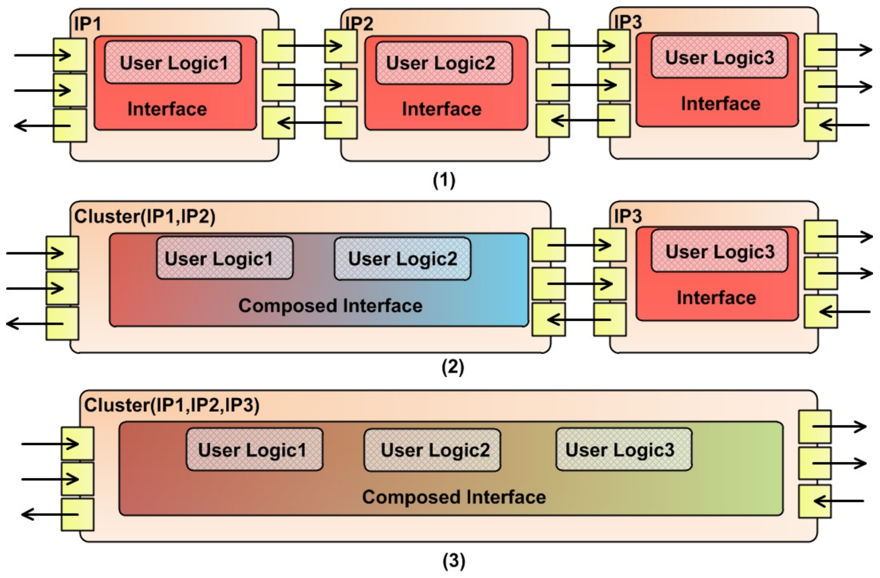


Figure II.1: Motivation example. In (a) the 3 components are connected serially while in (b) and (c) they are clustered.

There are few streaming interfaces available for FPGA. Xilinx provides AXI4-Stream [25] and Altera respectively the Avalon-streaming [1] for their FPGAs. However, those interfaces use more resources than the user usually needs. Moreover, they do not provide the formalism needed to capture interface properties and devise input for synthesis and verification tools. Interface formalism provides a means to unambiguously describe and manipulate constraints under which independently developed components can work properly together. Streaming architectures can also be organized in structures in which neighbors communicate directly through dedicated FIFOs [17] [24] [27]. The work in [20] presents the SIMPPL model that uses asynchronous FIFOs to connect different Computing Elements (CEs). FSL (Fast Simplex Link) [24] interface are implemented as 32-bit x 16-deep FIFOs, which helps to decouple the timing of the FSL master from the FSL slave. FERP (Full/Empty Register Pipe) [27] presents a similar architecture as the FSL, but with additional information to coordinate multiple streams of data. The use of FIFOs limits the model to streaming only, with no possibility of global data access as required when entire pictures are stored in the main memory. Furthermore, blind use of FIFOs increases resource usage between components that do not require intermediate storage in their communication link. Several works have focused on generating hardware/software interfaces [15] [12] [21] [11], other have provided BUS interconnect mechanism such as Open Core Protocol (OCP) [23], WISHBONE [8], AMBA [4] for integration of customized peripherals. The goal is to convert peripheral interface operations into packets that adhere a bus-specific protocol by inserting wrappers in the peripheral. This introduces unnecessary cost in streaming-oriented architectures. In contrast, we target direct communication models and use point-to-point interconnect structure for all on-chip communications. [28] [13] propose a method to generate interface circuits. The proposed solution produces flexible micro architectures from FSM descriptions of the two interfaces to be connected. Knowledge of the protocols of both the sender and the receiver is required. With the infinite number of available protocols and communication paradigms, it is nearly impossible for a single tool to target the general purpose case and provide a universal synthesis methodology.

Different formalisms exist for modeling interfaces, among which are relational interfaces [22], Assume/Guarantee contract [18], Interface Automata [2], and I/O automata [14]. [22] presents a theory of relational interfaces, that is, interfaces that specify relations between inputs and outputs, which is not in the scope of our work, which is IP reusability by a separation of concerns between interface and user-logic. In Assume/Guarantee contracts [18], the assumptions made on the environment and the guarantees provided by the system are modeled as separate sets of behaviors, whereas in interface theories the two are merged into a single model, called an interface. I/O automata and interface automata are formalisms that provide a single model for the input and output actions of a component. The main difference between the two formalisms is that I/O automata are required to be input-enabled, meaning they must be receptive at every state to each possible input action while for interface automata, some inputs may be illegal in particular states, i.e., the component is not prepared to service these inputs in those states. We are using a formalism that bears syntactic similarities to Interface Automata but significant differences arise as we use a lower level of abstraction. While actions in Interface Automata models represent methods and procedure calls, our model uses actions to describe the behavior of hardware signals.

Component-based design techniques are important for mastering design complexity and enhancing re-usability. Components are designed independent from their context of use and may be glued together through their interfaces. This view has led some authors, e.g. [3], [16], [5] to consider a component as a black box and to concentrate on the combination of components using a syntactic interface description of the components. However, none of these works have provided an implementation and optimization mechanism aimed at reducing metrics such as resource usage and power.

We address the limitations of the previous works with a novel data access interface that reduced IP interactions to the description of data source and destination. The proposed formal model captured the properties of our interface and allows seamless optimization through component clustering and functional verification.

II.4 Interface Model

In this section, we present an interface formalism based on Interface Automata(IA) [2], which is used to model interaction between system components and their environment. This interaction is performed by means of input and output actions. Input actions describe the behavior that the component expects (or assumes) from the environment. Output actions represent the behavior it communicates (or guarantees) to the environment.

Definition 1 (Interface Automaton) *An Interface Automaton is a tuple* $S =< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow>$ *where:*

- *Q is a finite set of states with $q^0 \in Q$ being the initial state and $Q^f \subseteq Q$ being the set of final states;*
- *$A^I, A^O and A^H$ are pairwise disjoint finite sets of input, output, and hidden actions, respectively, A is the set of all actions i.e. $A = A^I \cup A^O \cup A^H$;*
- *$\rightarrow \subseteq Q \times A \times Q$ is the transition relation that is required to be input deterministic (i.e. $(q, a, q_1), (q, a, q_2) \in \rightarrow$ implies $q_1 = q_2$ for all $a \in A^I$ and $q, q_1, q_2 \in Q$).*

An interface is a shared boundary across which two separate components of a system exchange information. Interface automata are light-weight models that capture the temporal behavior of an interface. The following examples of interface automata are borrowed from [9]. The automaton (Fig. II.2-a) receives and transmits messages over a lossy communication channel. The input actions msg, ack, and nack (resp., send, ok, and fail) are depicted by incoming (resp., outgoing) arrows to the enclosing box, and question (resp., exclamation) marks on edge labels. For the sake of optimization in image processing, we consider special cases of interface automata (input-only, output-only and input-output).

Definition 2 (Input-only) *An input-only interface is an interface that can only receive streams or data information.*

Input-only interfaces are used in components at the end of image processing chains such as displays, storage or image transfer.

Definition 3 (Output-only) *An output-only interface is an interface that can only transmit streams or data information.*

Output-only interfaces are used in components in front of image processing chains, including cameras, storage and image collectors. Figure II.2-b presents an example of output-only interface which can be used to generate messages for the interface of Fig. II.2-a.

Definition 4 (Input-Output) *An input-output interface is an interface that can receive and transmit streams or data information.*

Input-output interface can be used anywhere in an image processing chain, particularly between two components. Figure II.2-a presents an example of input-output interface that receives streams through its *msg* port and transmits them through its *send* port.

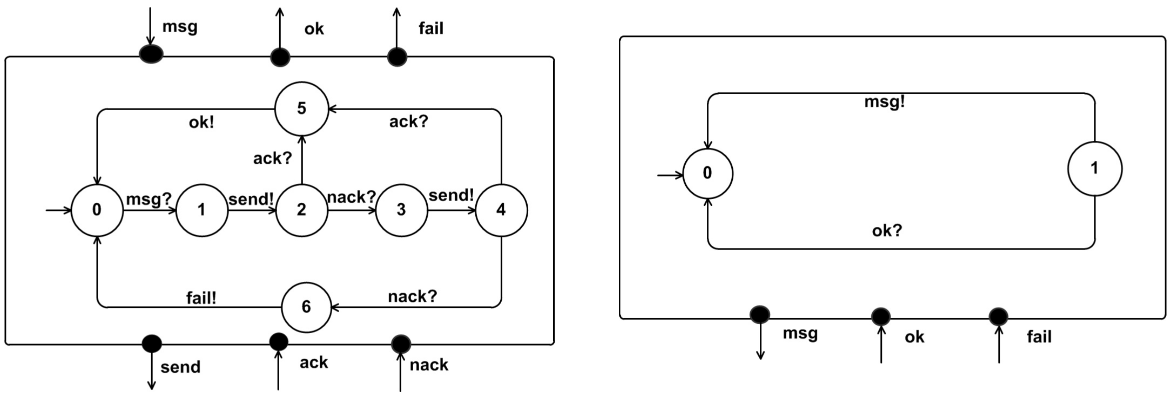


Figure II.2: Interface automata. The automaton is enclosed in a box, whose ports correspond to the input and output actions. The names of the actions are appended with the symbol ”?” (resp. ”!”, ”;”) to denote that the action is an input (resp. output, internal) action. An arrow without source denotes the initial state of the automaton.

Definition 5 (Compositionality) *Let S and T be two IA, and let shared(S,T) =* $A_S \cap A_T$ *be the set of shared actions. We say that S and T are composable whenever shared(S,T) =* $(A_S^I \cap A_T^O) \cup (A_S^O \cap A_T^I)$.

The interpretation is that variables in $shared(S, T)$ are outputs of S which are connected to inputs of T. Note that we allow $shared(S, T)$ to be empty, in which case serial composition reduces to parallel composition (where no connections between the two interfaces exist).

Definition 6 (Product of interfaces) *Let S and T be composable IA. The product* $S \bigotimes T$ *is the interface automata defined by:*

- $Q_{S\bigotimes T} = Q_S X Q_T$ *with* $q^0_{S\bigotimes T} = (q^0_S, q^0_T)$*;*
- $A^I_{S\bigotimes T} = A^I_S \cup A^I_T - shared(S,T), A^O_{S\bigotimes T} = A^O_S \cup A^O_T - shared(S,T),$ *and* $A^H_{S\bigotimes T} = A^H_S \cup A^H_T \cup shared(S,T)$
- $(q_S, q_T) \xrightarrow{a} S \bigotimes T(q_S^{,}, q_T^{,})$ *if any of the following holds:*
  - $a \in A_S - shared(S,T), q_S \xrightarrow{a} Sq_S^{,}$*, and* $q_T = q_T^{,}$
  - $a \in A_T - shared(S,T), q_T \xrightarrow{a} Tq_T^{,}$*, and* $q_S = q_S^{,}$
  - $a \in shared(S,T), q_S \xrightarrow{a} Sq_S^{,}$*, and* $q_T \xrightarrow{a} Tq_T^{,}$*,*

There may be reachable states on $S \bigotimes T$ for which one of the components, say S, may produce an output shared action that the other is not ready to accept (i.e. its corresponding input is not available at the current state). Those states are called error states. The composition of 2 IAs is defined as their product without the error states. To describe and perform resource reduction through component clustering, a formalism must be provided for the single components.

Definition 7 (Component) *A component, C, is a tuple (U, I) where*

- *U is the core function of C.*
- $I =< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow>$ *is an interface through which C interacts with other components, for instance, a messaging interface or a procedural interface.*

Components are executable units that read data and write data to ports. They are composed of a core that represents the task's functionality and an interface that establishes data transmission on the input/output ports. The component model (Definition 7) provides a framework for representing components and composing them. The interface I of a component can be an input-only or an output-only or an input-output, depending on where it is used in a system.

Definition 8 (Component clustering) *A component, C = (U, I), can be composed from a set of simpler components, $C_1(U_1, I_1), ..., C_n(U_n, I_n)$, as follows:*

- *U is constructed from $U_1, ..., U_n$ by connecting $U_1, ..., U_n$ through their interfaces.*
- *I is derived from the composition of $I_1, ..., I_n$.*

The goal of component clustering it to combine a set of simpler components into a unique component with equivalent core logic and an equivalent and resource optimized interface. Using the above formalism, we propose a new streaming interface and a component-composition mechanism to cluster components.

II.5 Component Interconnect and Data Access Interface

The need for a new interconnect mechanism was motivated by the desire to allow designers of image processing IP to focus on the functions of their IPs and let the data supply and collection mechanism be taken care of by the interconnect implementation. Data needed by image processing IP can be local, usually small part of images stored in buffer nearby, global with entire images stored in the main memory, or direct, as computation result from neighbor modules. Furthermore, local, direct and global data can be shared by several modules, which requires a coordination. The purpose of CIDA is to provide a simple interface, which would be used by designers to specify the source and destination of their data in a very abstract manner, regardless of the implementation. The designer would then be freed from the implementation of the orchestration and dataflow mechanism, which will be entirely handled in the interface. We have therefore designed the Component Interconnect and Data Access (CIDA), that fulfills the requirement of the interface model presented in section II.4 and provides a set of common features that IP cores can use.

CIDA features include a streaming interface for exchanging data among hardware and DMA for exchanging data with global memories. CIDA can be parameterized to efficiently accommodate different peripherals. To manage local, direct and remote data access, we differentiate between the two main properties of CIDA: CIDA Streaming for streaming-based design and CIDA-DMA for streaming design that incorporate global memory access.

II.5.1 CIDA streaming:

CIDA-Streaming is based on a handshaking protocol with the different configurations and the inputs, outputs and internal signals used to wrap a user logic function as shown in Figure II.3. The data signals are made generic to handle different data width during the streaming.

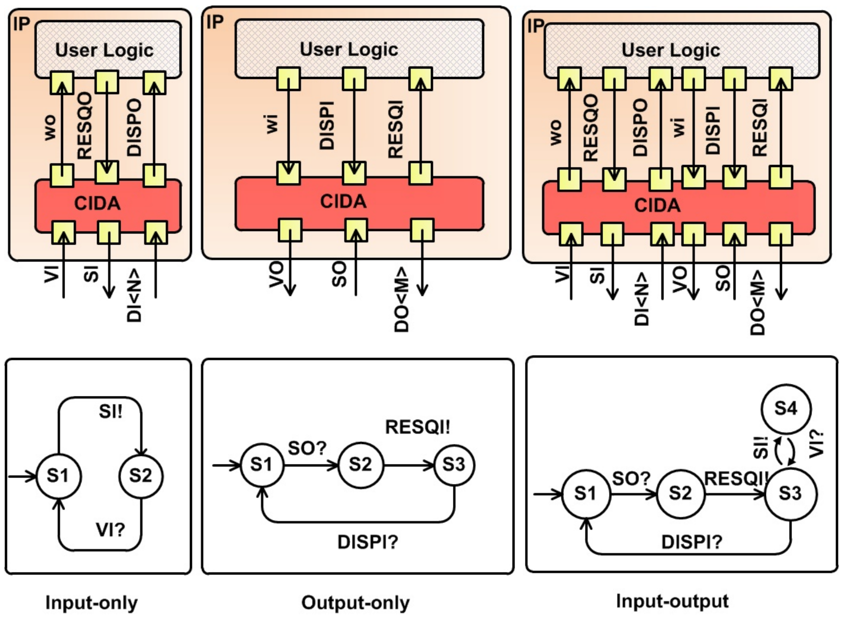


Figure II.3: CIDA streaming. On top, interface description; on bottom, interface automaton.

DI<N> represents the input data with width N, *VI* a valid data at the input, *SI* a request for data at the output. *DO<M>* describes the output data with width M, *VO* a valid data at the output, *SO* a request for input data, *Wo* data sent to User Logic with width N and *RESQO* a request for data to the User Logic, active low. *DISPO* signals valid data sent to the User Logic. *WI* is the data from User Logic, width N, *DISPI* a valid data from User Logic, *RESQI* a request for data from User Logic, active low. The number of signals needed is very limited in the CIDA implementation. When used as input-only, the interface is in state *S1* until the User Logic is ready to receive data. The User Logic sets *RESQO* low to signal it is ready to receive data, the interface sets *SI* low and waits for a valid data. Upon reception of a valid data, the data are transmitted to the User Logic and the interface returns to the state *S1*. In the output-only configuration, the interface is in state *S1* until it receives a request data. The interface transmits that request to the User Logic and waits for a valid data. Upon reception of a valid data, the data is sent out and the interface returns to the state *S1*. When used as input-output, the interface is in state *S1* until it receives a request for data. The interface transmits that request to the User Logic and waits for a valid data. In order to produce a valid data, the User Logic sets *RESQO* low to signal it is ready to receive data, the interface sets *SI* low and waits for a valid data. Upon reception of a valid data at the input, the data are transmitted to the User Logic, processed by the User Logic to produce a valid data at the output. Upon receiving a valid data from the User Logic, the data is sent out and the interface returns to the state *S1*. The User Logic can request as much data as it needs to produce a valid output. Figure II.3 shows the states and transitions for each configuration. To model the CIDA streaming as interface automaton $< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow >$ defined in section II.4, we consider the three cases input-only, output-only and input-output separately :

- Input-only : $I = < Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow > \; with$
  - Q={*S1, S2*}, q^0={*S1*}, Q^f={*S2*},
  - A^I={*VI, DI<N>, RESQO*},
  - A^O={*SI, wo, DISPO*},
  - A^H={$\oslash$},
  - $\rightarrow = \{S1 \xrightarrow{RESQO?} S2, S2 \xrightarrow{VI?} S1\}$.
- output-only : $I = < Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow > \; with$
  - Q={*S1, S2, S3*}, q^0={*S1*}, Q^f={*S3*},
  - A^I={*wi,DISPI, SO*},
  - A^O={*VO, DO<M>, RESQI*},
  - A^H={$\oslash$},
  - $\rightarrow = \{S1 \xrightarrow{SO?} S2, S2 \xrightarrow{RESQI!} S3, S3 \xrightarrow{DISPI?} S1\}$.
- Input-output : $I = I =< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow > \; with$

- Q={$S1$, $S2$, $S3$, $S4$}, q^0={$S1$}, Q^f={$S3$},
- A^I={VI, $DI<N>$, $RESQO$, wi,$DISPI$, SO},
- A^O={SI, wo, $DISPO$, VO, $DO<M>$, $RESQI$},
- A^H={$\emptyset$},
- $\rightarrow$= {$S1 \xrightarrow{SO?} S2, S2 \xrightarrow{a} S3, S3 \xrightarrow{b} S1, S3 \xrightarrow{c} S4, S4 \xrightarrow{VI?} S3$} *where a = 'RESQI!', b= 'DISPI?' and c= 'RESQO?'.*

II.5.2 CIDA DMA:

CIDA DMA is made of a streaming interface and a memory-mapped interface. CIDA DMA is used to write incoming streams into a memory frame buffer without processor intervention.

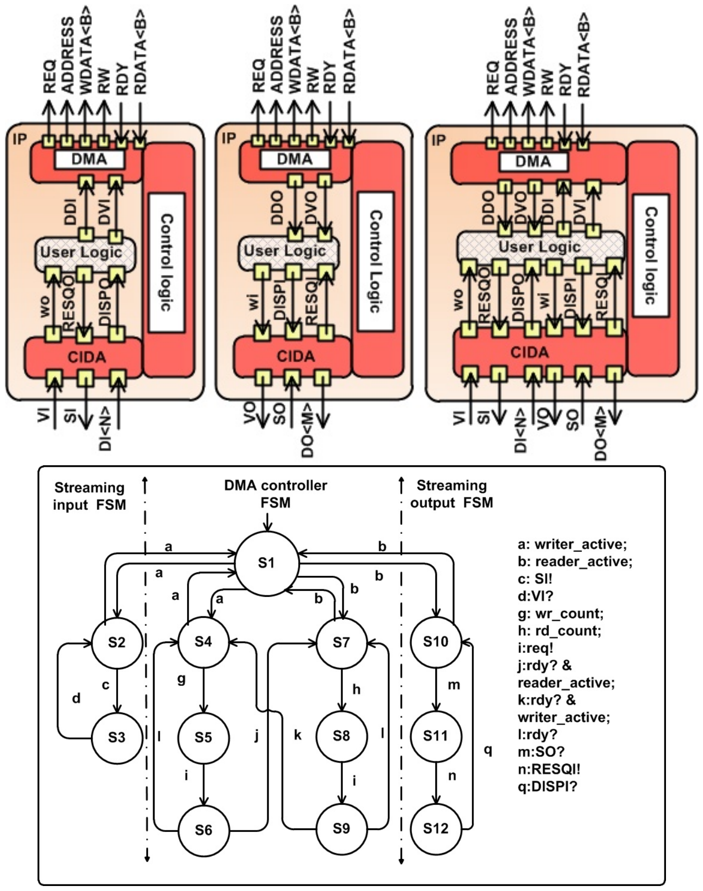


Figure II.4: CIDA DMA. On top, interface description; on bottom, interface automaton.

Reciprocally, images can be read from the frame buffer and streamed out. Figure II.6 shows an implementation of the DMA interface where incoming/outgoing streams are in FIFOs for synchronization and a scheduler is used to coordinate memory reads/writes and prevent collisions. The scheduling policy is currently based on round robin, but more complex scheduling mechanisms will be implemented in the future. Different configurations are possible as shown in Figure II.4. The corresponding FSM of a configuration is obtained by removing the missing streaming interface from the FSM in figure II.4. *DDI* represents the DMA input data, *DVI* a valid DMA input data, *DDO* the DMA output data, *DVO* a valid DMA output data. {*REQ, ADDRESS, WDATA, RW, RDY, RDATA*} are memory-mapped signals, which go through a BUS to allow interconnection between multiples DMA interfaces and multiple memories. DMA operations (read/write) can be done in single word or in burst mode. We used FIFOs to store streams for burst operations. Wr_count is the number of elements in the Write FIFO (Figure II.6) and *rd_count* is the number of elements in the Read FIFO. $Writer\_active$, respectively *reader_active*, is used to initialize writing and reading frames to and from memory. Reading and writing frame buffers can be done simultaneously, in which case a scheduler is used to order single-word or burst read/write operations. Figure II.4 shows the states and transitions for each configuration. The control logic contains memory-mapped registers that can be used to reset or start/stop execution of DMA operations from the software. It also contains a set of registers for configuring and gathering the status of IPs. This relieves the designer from implementing memory-mapped registers and bus interface in each IP which needs a configuration from the software. To model the CIDA DMA as interface automaton, we use $< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow>$ defined in section II.4 we consider the three cases input-only, output-only and input-output separately :

- Input-only : $I = < Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow> \ with$
  - *Q={S1, S2, S3, S4, S5, S6, S7, S8, S9}*, q^0*={1}*, Q^f*={S3}*,
  - A^I*={VI, DI<N>, RESQO, DDI, DVI, RDY, RDATA}*,
  - A^O*={SI, wo, DISPO, REQ, ADDRESS, WDATA, RW}*,
  - A^H*={writer_active, reader_active, wr_count, rd_count}*,
  - $\rightarrow = \{S1 \xrightarrow{a} S2, S2 \xrightarrow{a} S1, S2 \xrightarrow{b} S3, S3 \xrightarrow{VI?} S2\} \cup \rightarrow_{ctrl}$ *where a = 'witer_active;' and b ='RESQO?'.*

  .

- output-only : $I = < Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow> \ with$
  - *Q={S1, S4, S5, S6, S7, S8, S9, S10, S11, S12}*, q^0*={S1}*, Q^f*={S12}*,
  - A^I*={wi,DISPI, SO, RDY, RDATA}*,

- A^O={*VO, DO<M>, RESQI, DDO, DVO, REQ, ADDRESS, WDATA, RW*},
- A^H={*writer_active, reader_active, wr_count, rd_count*},
- $\rightarrow = \{S1 \xrightarrow{a} S10, S10 \xrightarrow{a} S1, S10 \xrightarrow{SO?} S11, S11 \xrightarrow{b} S12, S12 \xrightarrow{c} S10\} \cup \rightarrow_{ctrl}$ *where* a = *'reader_active;', b = 'RESQI!' and c = 'DISPI?'.*

.

- Input-output : $I = I =< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow > \; with$
  - Q={*S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12*}, q^0={*S1*}, Q^f={*S3,S12*},
  - A^I={*VI, DI<N>, RESQO, wi,DISPI, SO, DDI, DVI, RDY, RDATA*},
  - A^O={*SI, wo, DISPO, VO, DO<M>, RESQI, DDO, DVO, REQ, ADDRESS, WDATA, RW*},
  - A^H={*writer_active, reader_active, wr_count, rd_count*},
  - $\rightarrow = \{S1 \xrightarrow{a} S2, S2 \xrightarrow{a} S1, S2 \xrightarrow{b} S3, S3 \xrightarrow{VI?} S2, S1 \xrightarrow{d} S10, 10 \xrightarrow{d} S1, S10 \xrightarrow{SO?} S11, S11 \xrightarrow{e} S12, S12 \xrightarrow{c} S10\} \cup \rightarrow_{ctrl}$ *where* a = *'writer_active;', b = 'RESQO?', c = 'DISPI?', d = 'reader_active;' and e ='RESQI!'.*

.

where $\rightarrow_{ctrl} = \{S1 \xrightarrow{a} S4, S4 \xrightarrow{a} S1, S1 \xrightarrow{b} S7, S7 \xrightarrow{b} S1, S4 \xrightarrow{c} S5, S5 \xrightarrow{req!} S6, S6 \xrightarrow{rdy?} S4, S6 \xrightarrow{rdy?\&b} S7, S7 \xrightarrow{d} S8, S8 \xrightarrow{req!} S9, S9 \xrightarrow{rdy?} S7, S9 \xrightarrow{rdy?\&a} S4\}$ *where a = 'writer_active;', b ='reader_active;', c = 'wr_count;' and d = 'rd_count;'.*

II.6 System Integration with CIDA

Data management is one of the most important aspects of hardware/software systems. The huge amount of image data collected must be supplied in real-time to heterogeneous computing components that need to process them to avoid computation delays. System designers usually handle this step manually, by defining communication interfaces and protocols followed by low-level implementations. The complexity and versatility of protocols and algorithms increase the challenges and limit the portability of designs. To solve this issue, we provide, through CIDA, a mechanism which allows algorithm development to be decoupled from data transfer handling. For each module in the design, the user just needs to specify the source of data and result target. The system takes care of the data transfer, buffer instantiation and scheduling of memory access for shared memory components. Figure II.5 shows an abstract component composition using CIDA-Interface modules. Incoming image data are temporally stored into an input buffer.

A processing unit PU_1 reads images from the input buffer, performs some processing and copies the results to the memory. From there PU_2 reads the processed image and performs further computations. The result is sent to PU_3 where it is mixed with stored data in memory to produce the output. All components use CIDA at their boundary to handle the data transfers and memory scheduling. This allows designers to focus on implementing their components and let the system take care of data exchange.

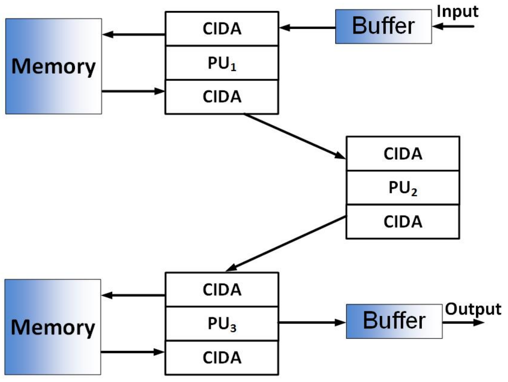


Figure II.5: Organization of data access. Designer specify only the source and sink of data using the CIDA interface. The system coordinates the data transfer and access to share memories.

II.7 Resource Optimization

In this section, we present the algorithm used for component clustering, which allows simpler components to be combined and produce more complex component with reduced resource requirements. Understanding the composition of 2 CIDA interfaces is key to understand this construction.

Consider the 2 input-output streaming interfaces I1 and I2 represented in figure II.7. Their serial composition i.e. (SO1,DO1,VO1) will coincide with (SI2,DI2,VI2).

[I1]: $I1 =< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow> \ with$

Q={*1, 2, 3, 4*}, q^0={*1*}, Q^f={*3*},

A^I={*VI1, DI1, RESQO1, wi1,DISPI1, SO1*},

A^O={*SI1, wo1, DISPO1, VO1, DO1, RESQI1*},

A^H={$\emptyset$},

$\rightarrow = \{1 \xrightarrow{SO1?} 2, 2 \xrightarrow{RESQI1!} 3, 3 \xrightarrow{DISPI1?} 1, 3 \xrightarrow{RESQO1?} 4, 4 \xrightarrow{VI1?} 3\}$.

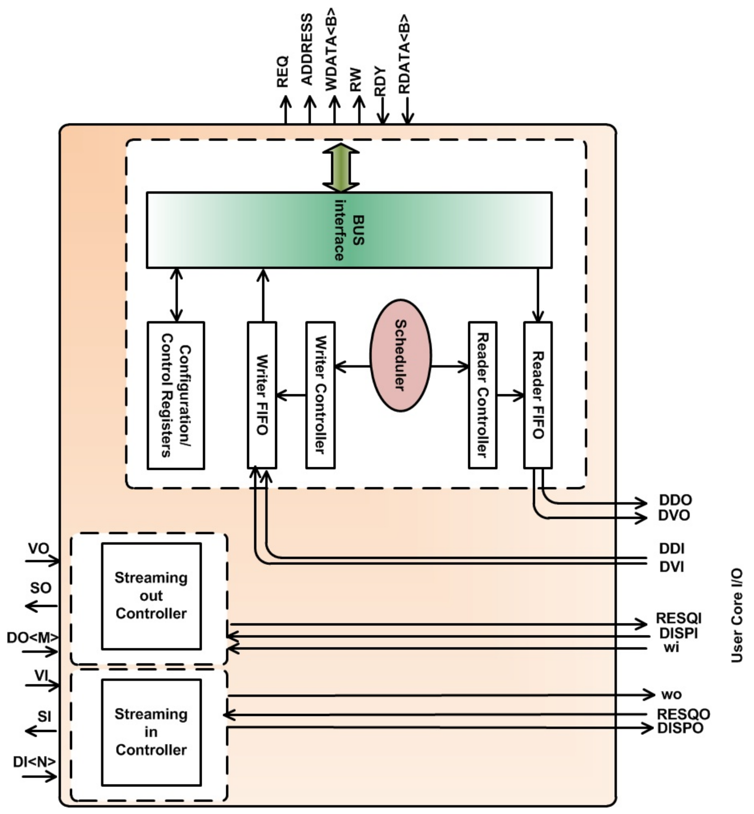


Figure II.6: System integration with CIDA; concrete FPGA-Implementation

[I2]: $I1 =< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow> \ with$

Q={*1, 2, 3, 4*}, q^0={*1*}, Q^f={*3*},

A^I={*VO1, DO1, RESQO2, wi2,DISPI2, SO2*},

A^O={*SO1, wo2, DISPO2, VO2, DO2, RESQI2*},

A^H={$\oslash$},

$\rightarrow = \{1 \xrightarrow{SO2?} 2, 2 \xrightarrow{RESQI2!} 3, 3 \xrightarrow{DISPI2?} 1, 3 \xrightarrow{RESQO2?} 4, 4 \xrightarrow{VO1?} 3\}$.

Shared(I1,I2) = {*SO1, VO1, DO1*}.

[I1 $\bigotimes$ I2]: The product I1 $\bigotimes$ I2 is obtained from definition 6 as follows:

- $A^I_{I1 \bigotimes I2} = A^I_{I1} \cup A^I_{I2} - shared(I1, I2)$ ={*VI1, DI1, RESQO1, wi1,DISPI1, RESQO2, wi2,DISPI2, SO2*}
- $A^O_{I1 \bigotimes I2} = A^O_{I1} \cup A^O_{I2} - shared(I1, I2)$ = {*SI1, wo1, DISPO1, RESQI1, wo2, DISPO2, VO2, DO2, RESQI2* }
- $A^H_{I1 \bigotimes I2} = A^H_{I1} \cup A^H_{I2} \cup shared((I1, I2)$ = {*SO1, VO1, DO1*}

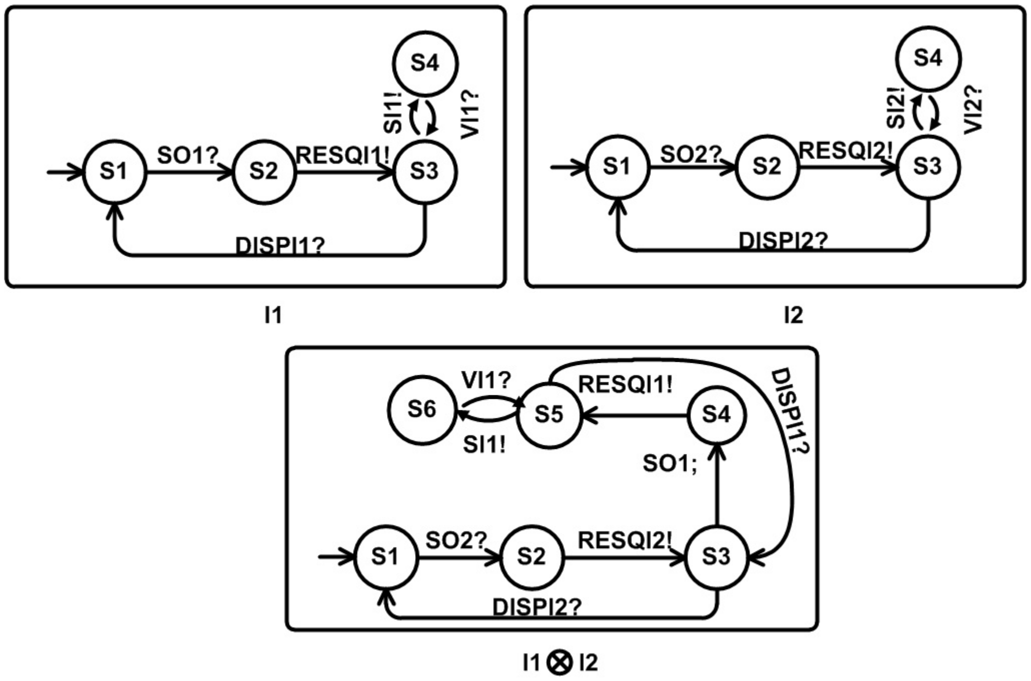


Figure II.7: Product of 2 CIDA interfaces.

Figure II.7 gives the result of the product I1 $\bigotimes$ I2. There is no error state, thus the composition of 2 CIDA interfaces is their product. From this example, we conclude that:

- The composition of 2 CIDA interfaces is their product.
- The composition of 2 CIDA interfaces serially connected is obtained by merging the 2 FSMs at the final state of the second one.

Based on the previous conclusions, we proposed the algorithm 1 for clustering components. Lines 3-24 define the inputs, outputs and hidden actions of the resulting component by looking at shared actions between 2 consecutive components. Lines 3-10 search for the inputs of component C_i that are outputs of the component C_{i-1}. These inputs are hidden actions in the final component C(U,I). Lines 11-17 search for the outputs of component C_i that are inputs of the component C_{i-1}. These outputs are hidden actions in the final component C(U,I). Lines 25-35 identify serially connected components and combine their FSMs by merging the 2 FSMs at the final state of the second one as shown in Figure II.7. In Line 37, the user logics $U_1, ..., U_n$ of all the component are instantiated in the final component. We wrote a program to parse the VHDL description of a set of components to obtain their input and output data and

FSMs. Our algorithm is then applied on those data to produce the VHDL description of the final component.

II.8 Functional Verification Support

Verification is an integral part of the system-on-chip design process and consumes up to 80% of the design efforts. A reduction of the verification time would substantially reduce costs and improve system reliability. We provide support for functional verification, which is the main approach currently used in the industry. We have integrated Universal Verification Methodology (UVM) to allow designs specified in RTL to be verified using the capabilities (random and constrained stimuli, coverage, assertions) of the UVM environment. From a module described in CIDA, we automatically extract information needed to generate drivers, monitors, assertion and coverage metrics for the functional verification. The verification can then be performed with signals generated in the UVM to feed designs in RTL. The results are gathered back in UVM using the monitor. The generated assertion component then checks for bugs, while the coverage module measures the quality of verification. Figure II.8 illustrates this approach. From an RTL description of a hardware, a *netlist extractor* is used to gather information on the component interface. In case of a CIDA interface, we automatically generate assertions and coverage metric. Otherwise, the user can provide assertion and metrics for coverage separately. Our framework then generates the components (sequencer, driver, monitor, coverage and assertion evaluation) needed for the verification within the UVM environment. The *sequencer* is used to generate the inputs for the system and the *driver* mimics protocol of real-life communication components such as UART, USB. The *monitor* gathers the output signals from the system under test which are then evaluated for correctness by the *assertion checker.* The *coverage checker* is used to measure the verification coverage, a measure of quality used in the industry. Because the design under test is described in RTL and UVM is based on System-Verilog, we use the SystemVerilog interface component to bridge the two descriptions. The Verification is done in two phases:

- In a first phase, the UVM environment is generated. A parsing step produces all the necessary information to generate a class packet in UVM. A packet is a data container combining the inputs and the outputs of the DUT. The class packet in

Algorithm 1 Component_clustering

Input: list of Ordered components $C_1(U_1, I_1), ..., C_n(U_n, I_n)$ With $I_i =< Q_i, q_i^0, Q_i^f, A_i^I, A_i^O, A_i^H, \rightarrow_i>$. Ordered means the data flows from C_1 to C_n

Output: C(U,I): component resulting from the clustering.

```
$q^0 \leftarrow q_n^0$
$Q \leftarrow q^0$
for $i$ = n to 2 step -1 do
    for $x \in A_i^I$ do
        if $x \in A_{i-1}^O$ then
            $A^I \leftarrow x$
        else
            $A^H \leftarrow x$
        end if
    end for
    for $x \in A_i^O$ do
        if $x \in A_{i-1}^I$ then
            $A^O \leftarrow x$
        else
            $A^H \leftarrow x$
        end if
    end for
    for $x \in A_1^O$ and $\notin A^H$ do
        $A^O \leftarrow x$
    end for
    for $x \in A_1^I$ and $\notin A^H$ do
        $A^I \leftarrow x$
    end for
end for
$Q \leftarrow q \in Q_n$
$\rightarrow \leftarrow \rightarrow_n$
for $i$ = n-1 to 1 step -1 do
    if $\exists x \in A_{i+1} \cap A_i$ then
        combine $q_i^0$ and $q_{i+1}^f$ in Q
        $Q \leftarrow q \in Q_i$ and $q \neq q_i^0$ and $q \neq q_i^f$
        combine $q_i^f$ and $q_{i+1}^f$ in Q
    else
        $Q \leftarrow q \in Q_i$
    end if
    $\rightarrow \leftarrow \rightarrow_i$
end for
$U \leftarrow U_1 \cup ... \cup U_n$
return C(U,I).
```

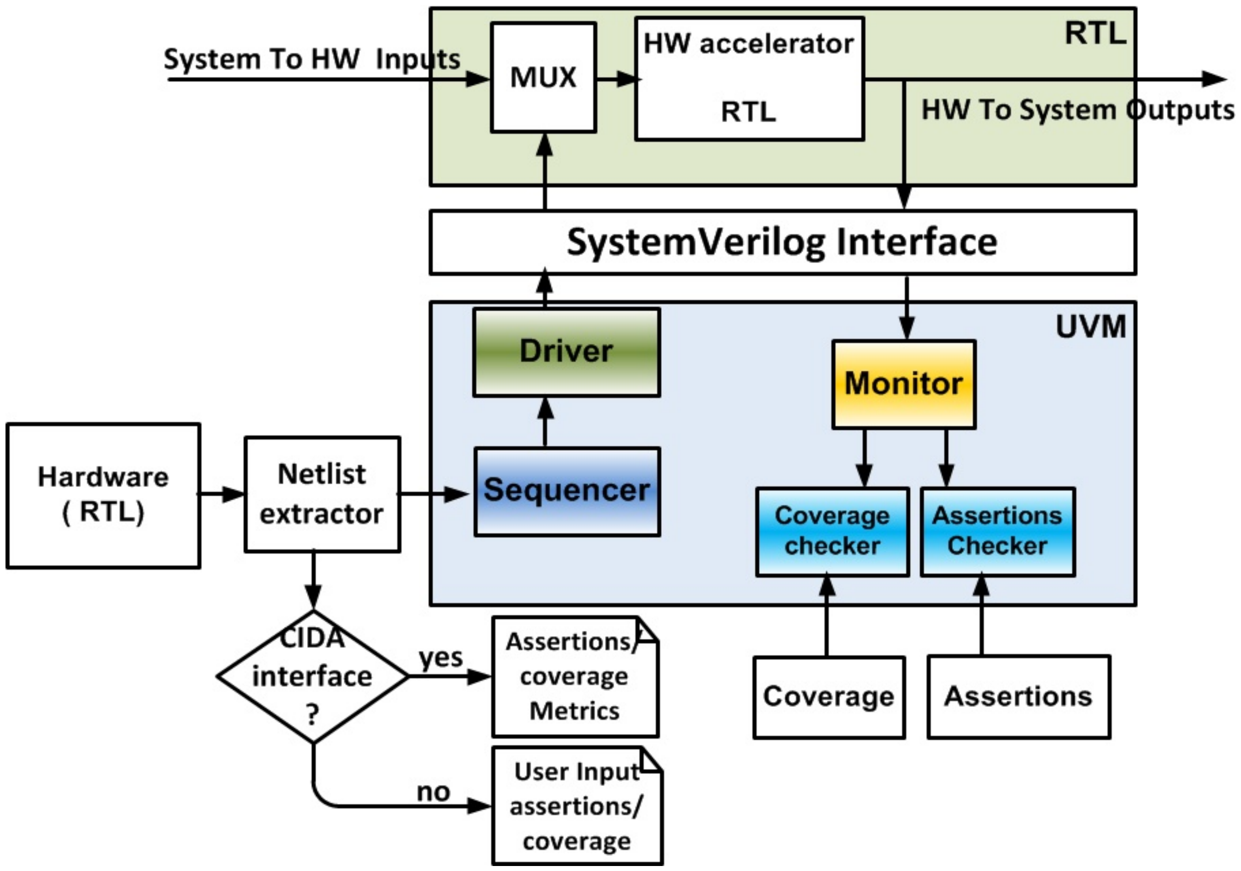


Figure II.8: UVM verification envrionment

UVM is then used to generate the UVM components. The Sequencer generates random and constrained packets. The Driver sends a packet to the DUT ports. The monitor is responsible for receiving the DUT's response to the stimulus packet from the driver, collecting coverage and performing assertion checking. The coverage is integrated into the environment using SystemVerilog coverage properties. Assertions are implemented using SystemVerilog assertions.

- In the second phase, the UVM environment is compiled and simulation is done using a simulation tool. During the simulation, Assertions are checked and coverage is collected for the specified DUT. Checking typically consists of verifying that the DUT Output meets the protocol specification.

Coverage metrics for CIDA are specified using SystemVerilog covergroup construct. Each covergroup can include a set of coverage points, cross coverage between coverage points and coverage options. A coverage point can be a variable or an expression. Each coverage point includes a set of bins associated with its sampled values or its value-transitions. A sc_logic signal should have only two possible values: 0 or 1. Therefore, the coverage of sc_logic signal such as VI is done with a bin [0:1]. A sc_logic_vector(m-1 downto 0) signal should have only 2^m values, the coverage of a sc_logic_vector signal such as DO<M> is done with a bin [0:2^m-1].

Assertion-Based Verification: Assertions for CIDA include the transitions between different states, liveness, and progression of subsystems. They are expressed using Temporal Logic [6].
Assertion-Based Verification is used to prove or discard some design properties or assertions during the simulation. Assertions are implemented to execute protocol or timing checking. They can be used to implement simple or complex property/sequence checks for an interface or a protocol. Temporal Logic can be used for the specification of assertions. It provides a means for reasoning about properties over time, for example the behavior of a finite-state system. Temporal logic is an extension of conventional logic. While conventional logic is useful for specifying combinational circuits, temporal logic is used for the specification of sequential circuits representing processes.

Temporal Logic: While the traditional logic uses operators such as $\vee, \wedge, \implies, \neg$, temporal logic introduces additional operators for dealing with temporal sequences. The Operator ∇: The expression ∇A means that the assertion A will be true at some future time, possibly the present time, but not necessarily remain true. The next Operator: The expression B $\implies$ next A means that if B is true at the present time, then A will be true at the "next" instant of time to be considered. As an example, if we consider a handshake protocol where a slave grants data (grant) on the next clock cycle after it receives a request (red) from a master, and the master sends an acknowledgment(ack) after receiving the data. This can be expressed using temporal logic as follows: req next grant => ∇ ack. Using Temporal Logic, we can discuss the properties of CIDA including liveness and progress of subsystems.
Liveness: Informally, an interface is alive if it eventually does something interesting, for example an input-only interface that eventually requests an input and receives a valid input. We define the liveness properties of CIDA interfaces as follows:

- Input-only : $\neg RESQO \implies (\nabla VI \wedge \nabla DISPO)$

- ouput-only : $\neg SO \implies (\nabla DISPI \wedge \nabla VO)$
- Input-ouput : (i) $\neg SO \implies (\nabla \neg RESQO \wedge \nabla DISPI \wedge \nabla VO)$ and (ii) $\neg RESQO \implies (\nabla VI \wedge \nabla DISPO)$

Progress of subsystems: In a subsystem S composed from connected components $C_1, C_2, ..., C_n$, we say that there is progression in S if a data at the output of C_1 reaches the input of C_n going through all the intermediate components i.e.
$VO_{C_1} \implies (\nabla VO_{C_2} \wedge ... \wedge \nabla VO_{C_{n-1}} \wedge \nabla VI_{C_n})$.
Progression is an important tool to verify that events such as an action generated in a component C_1 is properly received in a subsystem C_n, whose behavior depends on the generated event.
As an example, consider figure II.9 that illustrates the structure of a processing chain for a driving assistance [2].

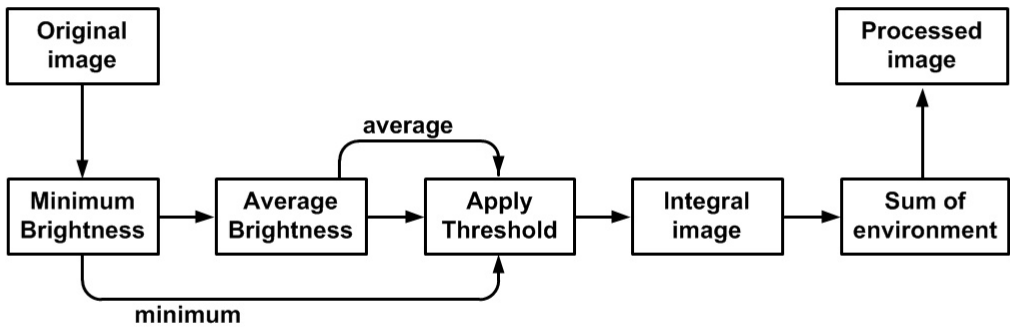


Figure II.9: Processing chain for driving assistance proposed in [2].

For the subsystems (Average Brightness, Apply Threshold, Integral Image), the progress of subsystems allows to check that when the valid minimum value is received by the Average Brightness module, a valid output is eventually produced by the Integral Image module.

II.9 Experimental Results

II.9.1 Interface comparison

Experimental Setup

Our target FPGA platform is the Zynq XC7Z020 CLG484-1 for Xilinx and Cyclone II EP2C35F672C6 for Altera. Our goal is to compare CIDA Streaming, AXI4-Stream and Avalon Streaming; but also CIDA DMA, AXI Video Direct Memory Access (AXI VDMA) and Scatter-Gather Direct Memory Access (SGDMA). The AXI VDMA core is a soft Xilinx IP core for high-bandwidth direct memory access between memory and AXI4-Stream-video type target peripherals. The SGDMA is available in Altera SOPC Builder and allows to transfer between Avalon memory-mapped and streaming interfaces.

Results

Table II.1 shows the results of the comparison. The used resources and the bandwidth are obtained after synthesis and the power is obtained after implementation using Xpower on Xilinx platform and PowerPlay on Altera platform. The advantages of CIDA compared to existing interfaces are not only the usage of less resources and less power as shown in Table II.1. CIDA DMA uses 2X less resources and power for almost the same bandwidth than AXI VDMA. The comparison with Altera is difficult since Altera and Xilinx use different logic unit element in their FPGAs, the relationship between the 2 logic units needs to be established. However, from the power information we can assume that CIDA DMA uses 2X less resources and power than SGDMA.

Interfaces	Streaming			DMA		
	Resources	Timing	Power	Resources	Timing	Power
AXI4-Stream	1-8 2-11 3-11	673.85 Mhz	0.010 mW	1-4176 2-3887 3-5452 4-6	229.31 Mhz	35.15 mW
Avalon-ST	6-1	N.A.	0.010 mW	5-1029 6-1507 7-2548	N.A.	32.84 mW
CIDA Streaming	1-34 2-2 3-35	1317.854 Mhz	0.001 mW	1-1646 2-1789 3-2484 4-1	173.04 Mhz	14.23 mW

Xilinx	Altera
1-Slice Registers	5-LC Combinational
2-Slice LUTs	6-LC Registers
3-LUT Flip Flop	7-Memory Bits
4-BRAM	

Table II.1: Interface comparison CIDA VS AXI4-stream VS Avalon-ST.

II.9.2 Resource Optimization

Experimental Setup

Our target FPGA platform is the Xilinx Zynq XC7Z020 CLG484-1 and the test cases include a set of real-life computation-intensive programs: hand follower, segmentation, Harris corner, and Xilinx Zynq base reference design, raytracer, and JPEG encoder. Those test cases feature image processing algorithms where data are streamed between the used IPs. The hand follower project identifies skin like colors (like hands) in the image and uses mean shift to follow it while moving around. The steps within the processing chain for this task are: grabbing images from the camera, convert Bayer to RGB, convert RGB to HSV and filter the skin like colors, perform mean shift algorithm to follow the skin colors (initial window in the middle of the screen), add the current window as a red rectangle to the video stream, the results are put into memory and sent to a display. The segmentation does the foreground/background segmentation. It stores an initial image as reference background and defines a pixel as foreground if it has a significant different color value compared to its corresponding background pixel. Two CIDA DMA are used. One CIDA DMA reads the calculated foreground and stores it to the memory where it is read for the display. The second CIDA DMA handles reading the old background frame from the memory and storing the new background in the memory. The central IPCore is the segmentation core which reads two image streams (old background frame, current image) and gives back also two image streams (new background frame, foreground). The input image is converted from Bayer pattern to RGB after streamed from the camera and before entering the segmentation core. Harris corner test case calculates the Harris corner points, a well known interest point descriptor. The incoming image is converted into a RGB image. Then the discrete approximation of the horizontal and vertical derivative is calculated, using 3x3 Sobel operators. Harris points are then found as points where both the horizontal and vertical derivative are significantly large. The results are put into memory and sent to a display. In this project we also make use of the Serialization cores. Because the image grabber core outputs 80 bits data wide while the Bayer to RGB converter needs 10 bits data wide. The Xilinx Zynq Base TRD from [26] is an embedded video processing application designed to showcase various features and capabilities of the Zynq Z-7020 AP SoC device for the embedded domain. The Base TRD consists of two elements: The Zynq-7000 AP SoC Processing System (PS) and a video processing pipeline implemented in Programmable Logic (PL). The AP SoC allows the user to implement a video processing algorithm that performs edge detection on an image (Sobel filter) either as a software program running on the Zynq-7000 AP SoC based PS or as a hardware accelerator inside the AP SoC based PL. The raytracer is a working implementation of a tracing processor obtained from [10]. The JPEG encoder is an implementation of the JPEG compression available on [9].

Results

The number of components to cluster affects the quality of our algorithm. The tradeoff with the number of components to cluster is shown in Figure II.10 for the Harris corner example. The resource usage is computed for different numbers of components clustered. The Segmentation and hand follower test cases showed similar behavior as the Harris corner example. In Figure II.10, the minimal resource usage is obtained when up to 6 components are clustered. The number of components to cluster is assigned to be 6 in the subsequent experiments. Table II.2 shows the comparison of our proposed component clustering algorithm compared to the flow without component clustering. The first column lists the names of test cases. For each test case, the second and third columns are the flip flop usage of the implementation result without and with component clustering respectively; the fourth column is the comparison between the implementation without and with component clustering. Similarly, the fifth to seventh columns are the numbers and comparison of Slice usage; the eighth to tenth columns are the numbers and comparison of maximum frequency. Overall, our component clustering flow can achieve about an average of 40% reduction over the implementation without component clustering. The algorithm performed poorly on the Xilinx Zynq base design because most the IPs used in that design are encrypted i.e. it is not possible to access the user logic core of these IPs. We were able to cluster only up to 3 components instead of 6 like in the other test cases. Table II.3 shows the comparison of the power consumption. The Power is evaluated using the Xilinx Xpower analyzer. We note a 5% average power reduction on the test cases excluding Xilinx Zynq Base design.

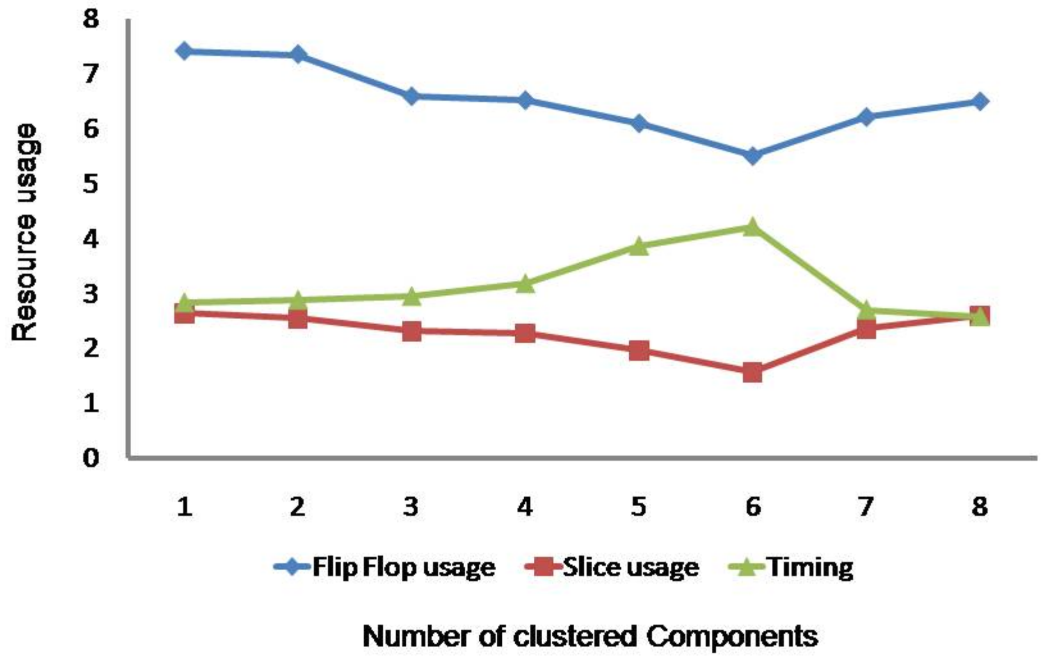


Figure II.10: Resource usage with different number of component clustered

II.10 Conclusions

In this paper, a streaming interface, called Component Interconnect and Data Access (CIDA), based on interface automata formalism, for modeling video processing architectures has been presented. The notions of interface composition and component clustering were formalized and their applications to resource reduction were demonstrated. CIDA has successfully been used to design several video applications such as Harris corner, segmentation, hand follower and more. Further, the component clustering is applied to solve the resource optimization problem on FPGA platforms. Our experiments show the efficacy of the component clustering algorithm. Our future work includes the use of CIDA for memory synthesis and optimization in video applications.

Case study	FF	FF (p)	CMP	SLICE	SLICE (p)	CMP	TIMING	TIMING (p)	CMP
Harris Corner	7,416	5,512	25.67%	2,648	1,567	40.82%	58.377 MHz	82.210 MHz	40.82%
Hand Follower	7,241	6,283	13.23%	2,105	1,776	15.62%	71.058 MHz	71.989 MHz	1.31%
Segmentation	10,799	7,602	29.60%	3,980	2,199	44.75%	72.606 MHz	78.027 MHz	7.46%
JPEG Encoder	6,617	6,125	7.43%	2,051	1,744	14.96%	101.491 MHz	106.202 MHz	4.64%
Ray Tracer	10,439	9,734	6.75 %	2,944	2,625	10.83%	77.597 MHz	79.434 MHz	2.36%
Xilinx Zynq Base	27,243	27,064	0.65 %	8,806	8,722	0.95%	70.822 MHz	75.769 MHz	6.98%

Table II.2: Resource reduction on all testcases.

Case study	Power	Power(p)	CMP
Harris Corner	128 mW	120 mW	6.25%
Hand Follower	128 mW	123 mW	3.90%
Segmentation	135 mW	125 mW	7.40%
JPEG Encoder	112 mW	107 mW	4.46%
Ray Tracer	142 mW	137 mW	3.52%
Xilinx Zynq Base	393 mW	393 mW	0%

Table II.3: Power reduction on all testcases.

Bibliography

[1] Altera. Avalon interface specifications. URL: `http://www.altera.com/literature/manual/mnl_avalon_spec.pdf.`, 2010.

[2] Jakob Anders, Michael Mefenza, Christophe Bobda, Franck Yonga, Zeyad Aklah, and Kevin Gunn. A hardware/software prototyping system for driving assistance investigations. *Journal of Real-Time Image Processing*, pages 1–11, 2013.

[3] Farhad Arbab. Abstract behavior types: A foundation model for components and their composition. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 33–70. Springer Verlag, 2003.

[4] ARM. Advanced microcontroller bus architecture specification. URL: `http://www.arm.com/armtech/AMBA_spec.`, 1996.

[5] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, and Manfred Broy. A formal model for componentware. In *IN FOUNDATIONS OF COMPONENT-BASED SYSTEMS. MURALI SITARAMAN AND GARY LEAVENS*, pages 189–210. University Press, 1999.

[6] G.V. Bochmann. Hardware specification with temporal logic: An example. *Computers, IEEE Transactions on*, C-31(3):223–231, March 1982.

[7] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[8] Changlei Dongye. Design of the on-chip bus based on wishbone. In *Electronics, Communications and Control (ICECC), 2011 International Conference on*, pages 3653–3656, Sept 2011.

[9] Michael Emmi, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Assume-guarantee verification for interface automata. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 116–131, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Josh Fender. A ray tracing engine. URL: `http://www.eecg.toronto.edu/~jayar/benchmarks/bench.html`, 2003.

[11] Sumit Gupta, Manev Luthra, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Hardware and interface synthesis of fpga blocks using parallelizing code transformations, 2003.

[12] M.A Hasamnis and S.S. Limaye. Custom hardware interface using nios ii processor through gpio. In *Industrial Electronics and Applications (ICIEA), 2012 7th IEEE Conference on*, pages 1381–1385, July 2012.

[13] S. Ihmor, T. Loke, and W. Hardt. Synthesis of communication structures and protocols in distributed embedded systems. In *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pages 3–9, June 2005.

[14] D.K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed i/o automata: a mathematical framework for modeling and analyzing real-time systems. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 166–177, Dec 2003.

[15] S. Kohara, N. Tomono, J. Uchida, Y. Miyaoka, Nozomu Togawa, M. Yanagisawa, and T. Ohtsuki. An interface-circuit synthesis method with configurable processor core in ip-based soc designs. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6 pp.–, Jan 2006.

[16] S. Moschoyiannis and M. W. Shields. Component-based design: towards guided composition. In *Application of Concurrency to System Design, 2003. Proceedings. Third International Conference on*, pages 122–131, June 2003.

[17] Stephen Neuendorffer and Kees Vissers. Streaming systems in fpgas. In *Proceedings of the 8th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '08, pages 147–156, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] Pierluigi Nuzzo, Antonio Iannopollo, Stavros Tripakis, and Alberto L. Sangiovanni-Vincentelli. From relational interfaces to assume-guarantee contracts. Technical Report UCB/EECS-2014-21, EECS Department, University of California, Berkeley, Mar 2014.

[19] OPencores. Jpeg encoder :: Overview. URL: `http://www.opencores.org`, 2009.

[20] L. Shannon, B. Fort, S. Parikh, A Patel, M. Saldana, and P. Chow. Designing an fpga soc using a standardized ip block interface. In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 341–342, Dec 2005.

[21] J. Thiel and R.K. Cytron. Splice: A standardized peripheral logic and interface creation engine. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[22] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. On relational interfaces. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 67–76, New York, NY, USA, 2009. ACM.

[23] Chih-Wea Wang, Chi-Shao Lai, Chi-Feng Wu, Shih-Arn Hwang, and Ying-Hsi Lin. On-chip interconnection design and soc integration with ocp. In *VLSI Design, Automation and Test, 2008. VLSI-DAT 2008. IEEE International Symposium on*, pages 25–28, April 2008.

[24] Xilinx. Fast simplex link channel. URL: `http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf`, 2004.

[25] Xilinx. Axi4-stream interconnect. URL: `http://www.xilinx.com/products/intellectual-property/axi4-stream_interconnect.htm.`, 2011.

[26] Xilinx. Zynq base trd 14.5. URL: `http://www.wiki.xilinx.com/Zynq+Base+TRD+14.5`, 2013.

[27] Jeff Young and Ron Sass. Ferp interface and interconnect cores for stream processing applications. In LaurenceT. Yang, Minyi Guo, GuangR. Gao, and NirajK. Jha, editors, *Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 291–300. Springer Berlin Heidelberg, 2004.

[28] Chang Ryul Yun, DongSoo Kang, Younghwan Bae, Hanjin Cho, and Kyoung-Son Jhang. Automatic interface synthesis based on the classification of interface protocols of ips. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 589–594, March 2008.

APPENDIX

I, certify that Michael Mefenza Nentedem is the first author of the article *"Component Interconnect and Data Access Interface for Embedded Vision Applications"* and have completed at least 51% of the work in the article.

Name:____________________ Signature: ____________________ Date: ____________________

III A Framework for Rapid Prototyping of Embedded Vision Applications

Michael Mefenza, Franck Yonga, Luca B Saldanha, Christophe Bobda and Senem Velipassalar

Abstract— We present a framework for fast prototyping of embedded video applications. Starting with a high-level executable specification written in OpenCV, we apply semi-automatic refinements of the specification at various levels (TLM and RTL), the lowest of which is a system-on-chip prototype in FPGA. The refinement leverages the structure of image processing applications to map high-level representations to lower level implementation with limited user intervention. Our framework integrates the computer vision library OpenCV for software, SystemC/TLM for high-level hardware representation, UVM and QEMU-OS for virtual prototyping and verification into a single and uniform design and verification flow. With applications in the field of driving assistance and object recognition, we prove the usability of our framework in producing performance and correct design.

Keywords— Vision, Verification, SystemC, SoC, UVM, FPGA.

III.1 Introduction

Due to their advantages in size, cost, and programmability, computer vision-based systems have become a promising alternative to conventional sensors such as RADAR and LIDAR for gathering information on the surrounding environment. Robotics, autonomous driving cars and unmanned aerial systems (UAS) are few examples of application areas that can benefit from the use of computer vision. UAS are increasingly used in surveillance, precision agriculture, search and rescue and communications relay. Image processing applications are increasingly complex, in part because of the huge amount of functionality required by customers and the huge amount of data that high-density sensors can deliver. Besides the computational performance, many systems require additional constraints such as SWAP (Size Weight and Power), which can be satisfied only with a combination of hardware and software, where complex low-level repetitive computations on huge amounts of data is done in hardware, while control dominated parts used for reasoning are kept in software. This path is followed by the industry, which provides domain specific processors with hardware built-in capabilities tailored for dedicated low-level image processing. Those processors perform well only on algorithms that follow their implementation patterns. For more complex application, a hardware/software system must be sought for real-time computation. The design of hardware/software systems for embedded video applications is a difficult task that requires hardware and software skills along with deep knowledge in image processing. It usually involves a manual partitioning of the application followed by separate implementation of software and hardware parts. While the software is implemented in imperative languages such as C/C++, hardware parts require hardware description languages such as Verilog or VHDL. The process is complicated, error prone and time consuming for hardware designers, left alone software developers who build the bulk of image processing community. As a consequence, there is a need to map software applications onto dedicated hardware/software architecture for performance improvement. In critical environments such as UAS and cars, verification must be performed to insure the correctness of design and avoid run-time malfunction that can damage properties or human life.

To address these issues, we propose a framework for rapid prototyping of system-on-chip with emphasis of embedded video applications. The proposed framework leverage existing tools and provide a means to facilitate their integration toward a semi-automatic mapping of software specification to hardware/software implementations. There are several tools involved in the design and evaluation of embedded vision applications. At the highest level, computer vision libraries such as OpenCV are used by to implement executable specifications and quickly prove concepts. Hardware/Software partitioning is then done, mostly manually using profiling data of the executable. Hardware functions are then implemented along with interconnect and memory management components using VHDL or Verilog compilers. In many cases SystemC/TLM is used to provide a high level system-on-chip representation that can be quickly simulated and verified with tools such a UVM (Unified Verification Methodology) or SCV (SystemC Verification). While each of these tools is used for a dedicated part of the design, there is currently no glue that connects them to provide a unified design flow with migration of specifications and data from one tool to another. Translation of designs is manually performed by designers with a considerable amount of recoding and errors.

Our main contribution in this work is to provide a seamless integration of all tools and methods involved in the design and verification of system-on-chips for vision-based applications. This is done by leveraging common structures of image processing to provide configurable processing and data exchange components across all levels of specification, along with a semi-automatic mapping and verification of high-level specifications into hardware/software implementations. With driving assistance and object recognition applications, we demonstrate the importance of our design framework.

The rest of the paper is organized as follow: In section III.2 relevant work in high-level design and verification of video-based embedded systems is presented. In section III.3 a conceptual view of the proposed methodology is presented. Section III.4 is devoted to evaluate performance of the methodology. Finally, section III.5 concludes the paper.

III.2 Related work

A review of embedded video processing systems in [14] shows that computation in current systems is performed in software on a general purpose processor, sometimes optimized for multimedia computation. The system in [6] uses several digital signal processors on different PCI-boards, while the CITRIC [7] relies on the Intel XScale PXA270 processor. FPGAs are used in systems such as [5] mostly as co-processor just for accelerating a single function. In [5] a Xilinx Spartan-3E FPGA is used as a co-processor. In [17], a hardware/software implementation on a Xilinx FPGA platform is presented for a 3D facial pose tracking application; the most-computationally intensive part is implemented in hardware while the remaining is implemented in the soft-core processors. Various architectures use programmable DSPs with additional resources, such as special graphics controllers and reconfigurable logic devices, as shown in [15]. These implementations are done manually using low-level languages. This approach is tedious and error-prone as we mentioned. In [20], Xilinx presents a limited library of synthesizable OpenCV functions. However, native OpenCV functions must be manually replaced by functions from the synthesizable library. This limitation is made to overcome the handling of OpenCV dynamic memory allocation with non synthesizable objects such as cv::Mat. Furthermore Xilinx OpenCV function library uses a streaming pixel approach rather than the native array pixels. As a result, random access is not supported for images. Many OpenCV functions such as at () method and cvGet2D () have no correspondence in the Xilinx library, which prevent interoperability and porting across platforms. Functions are only provided for AXI4 Streaming Video, thus preventing its usage for other FPGAs. Intel Integrated Performance Primitives (Intel IPP) [3] was designed to optimized OpenCV on Intel processor featuring Streaming SIMD (Single Instruction Multiple Data) Extensions (SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2, and Intel AVX). There is no support for dedicated hardware acceleration beyond the SIMD extension. Verification for embedded imaging systems is addressed in [19][1], which present an automatic environment for the verification of the image signal processing IPs. However, this work is limited to hardware accelerators. In [19], the verification environment is based on Specman language while [1] uses reusable eVC and C-Model. In [13], a specific application system with the co-simulation of SystemC and RTL is presented. The co-simulation of SystemC TLM is used in a surveillance camera system using a one-bit motion detection algorithm for portable applications. The host controller interface (HCI) and the motion detection sensor (MDS) are implemented by SystemC TLM. API program is implemented by C++ program and the other blocks of this system are implemented by RTL HDL. To verify co-simulation, HCI, MDS, and API program are operated at a PC workstation. Co-simulation is used to accelerate the simulation of the system. In [18] video data and synchronization signals are generated as testbench for the simulation of video processing IPs. However, simulation without emulation is limited since it does not catch every error in the RTL systems, especially timing errors. The scope in [16] is limited to algorithms by formally verifying complex loop transformations like loop folding, loop distribution, typically applied in the design trajectory of data dominated signal and data processing applications. Important part of the systems like interfacing, communication and dataflow component, and memory are not addressed.

The presented related works do not address the high-level design and verification of video-based embedded systems in tandem. Our goal in this work is to provide software designers a mean to capture designs at high-level with subsequent refinements and verification down to the hardware/software implementation and the emulation in the FPGA.

III.3 Proposed Design Approach

We propose the design environment of Figure III.1 which consists of the following steps: system specification, high-level hardware/software partitioning, register transfer implementation, and emulation. At each level, the corresponding abstract representation of the target architecture is used with the required amount of details to verify and refine the representation to the lower level.

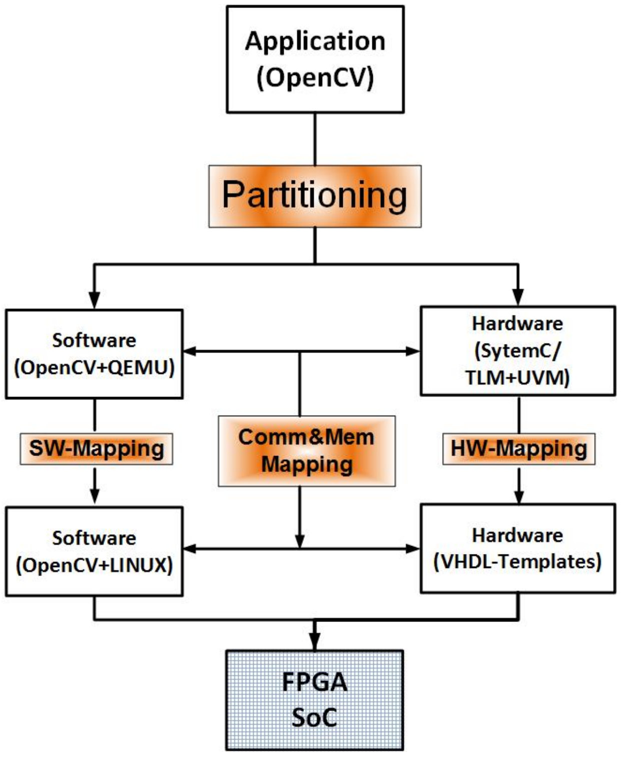


Figure III.1: The Proposed Design Flow

System specification: The first step in system specification is to describe the application, regardless of the target architectures. At this stage, only image processing skills are required and applications are defined in executable form in C/C++ in the OpenCV environment. The verification is done by means of simulation just to validate the application. Synthetic videos or a webcam can be used to feed video to the application, and a normal computer screen to visualize the results.

High-Level Hardware/Software Partitioning: Executable specification produced in OpenCV in the first step is refined to hardware/software architecture. The partitioning is done manually, either on the base of profiling information or user knowledge.

SystemC/TLM is used to model the behavior of the entire system in a transactional way. Transaction Level Modeling (TLM) is becoming increasingly popular in the industry as the ultimate tool to capture and verify systems consisting of several software processes and hardware components. To speed-up the simulation, details of the communication are left out. Our framework integrates OpenCV and SystemC/TLM in the same environment through a set of classes and functions to access image data from both end and allow communications among tasks running in OpenCV and those described in SystemC. Designs at this level reflect the final hardware/software implementation with SystemC describing the hardware architecture as a set of blocks with abstract communication among them and all reasoning on extracted features as software in OpenCV. Figure III.2

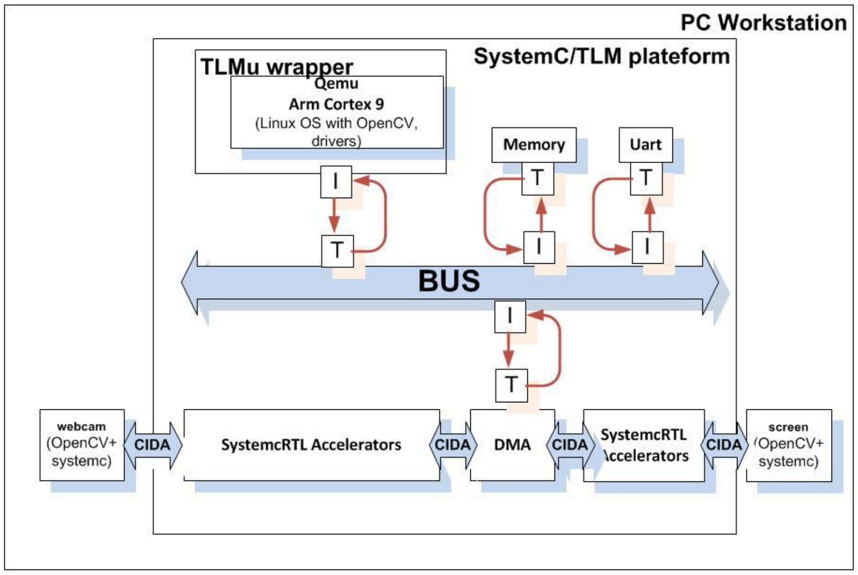


Figure III.2: SystemC/TLM abstract representation of the target platform at system-level. Initiator (I) and Target (T) interfaces are automatically included at the boundary of components of an abstract system-on-chip description. CIDA is used to handle data access from hardware modules.

presents an abstract representation of a system-on-chip for the Xilinx Zynq-FPGA. The ARM target processor running Linux-OS is modeled using QEMU, a generic and open source hypervisor. QEMU is integrated into the OpenCV + SystemC/TLM environment using TLMu [2], a TLM wrapper for QEMU that allows to communicate with the CPU core using TLM2.0 sockets. A hardware accelerator implemented in SystemC is used after image capture to perform low-level segmentation on input images captured by a webcam. The result is DMA into the system memory from where it can be further evaluated by the processor. A second hardware accelerator is used at the output to improve the output image stored by the processor in memory. At this level, integration of components into an abstract system-on-chip architecture is done by implementing initiators (I) and targets (T) as well as the communication logic. This step is simplified in our framework with configurable interface modules to connect components as initiator or target. Data exchange is performed according to the protocol of the target system. Currently, we support simple bus communication protocol, but other paradigms like network-on-chip can easily be included. The resulting system (OpenCV + SystemC/TLM + QEMU) is then used to simulate the abstract SoC with software and OS at the transaction-level, and inputs and output provided by the OpenCV. The video input used for simulation is the same as in the first stage. As we will explain later, the refinement from high-level is done by mapping high-level descriptions of image processing functions and structures to their low level counterparts. Automatic extraction of hardware designs from sequential code is increasingly and partially supported by vendor tools such as Xilinx Vivado, Mentor HandelC and Synopsys SympohonyC. These tools can be used to generate and map image processing functions available in software into hardware description, thus enriching the framework. This part is however not in the scope of this work. To allow for a seamless integration of hardware and software modules with a system-wide transparent data exchange, we have designed a *Component Interconnect for Data Access (CIDA).* CIDA is a portable interface module used for data exchange between software and hardware components in a system-on-chip. A SystemC-TLM level, CIDA uses direct memory access based on TLM-Socket to allow for communication among hardware and software components on the chip. As shown in Figure III.2, users just need to understand the CIDA-interface to be able to connect very complex representation. More explanation on CIDA is provided in section III.3.

Register-Transfer Level: The abstract description of the previous step is further refined into a final structure that can be synthesized by hardware compilers. The refinement includes the pin and cycle accurate implementation of the communication interface between software and hardware, a detailed description of the bus model, and a detailed implementation of buffers and memory.

We leveraged the structure of image processing systems to provide a generic and configurable set of components that will allow for a smooth refinement from the TLM to the RTL description. Our framework provides RTL implementations of image processing components and functions available at the SystemC/TLM level, some of which are described in more detail in section III.3.1. The SystemC/TLM description is automatically mapped into an RTL implementation that can be synthesized, while the software part is kept on the embedded processor. Also, the CIDA TLM implementation is mapped to its RTL counterpart and used as in the previous step for component integration and memory access management. Since all communications among software and hardware tasks go through the CIDA, which is hardware module, it must be made accessible from the software. The mapping automatically generates a driver, using the properties of the implementation.

***Emulation**:* The last step in the design process is the emulation of the system specified in the previous level. For this step, we have designed a versatile FPGA-based smart camera, the *RazorCam* to allow for testing in a real-life environments. Our platform implements image processing directly inside the camera, instead of propagating the image to a workstation for processing. Its compact size and performance facilitates the integration in embedded environments like cars and UAS. The processing module consists of: one Xilinx Zynq FPGA, a flash drive, connectors for an infrared camera, a digital camera sensor, and an analog camera sensor. A TFT display can be connected to the platform, so the user can check the results of its applications in real-time in the field.

System Integration with CIDA

Data management is one of the most important aspects in hardware/software systems. The huge amount image data collected must be supplied in real-time to heterogeneous computing components that need to process them to avoid computation delays. System designers usually handle this step manually, by defining communication interface and protocols followed by low-level implementations. The complexity and versatility of protocols and algorithm increase the challenges and limit the portability of designs. To solve this issue, we provide a mechanism which allows algorithm development to be decoupled from data transfer handling. The resulting data access module called Component Interconnect and Data Access (CIDA) is based on the interface model, whose formalization fulfills the interface automata [2] [4] definition. Using this model, we can

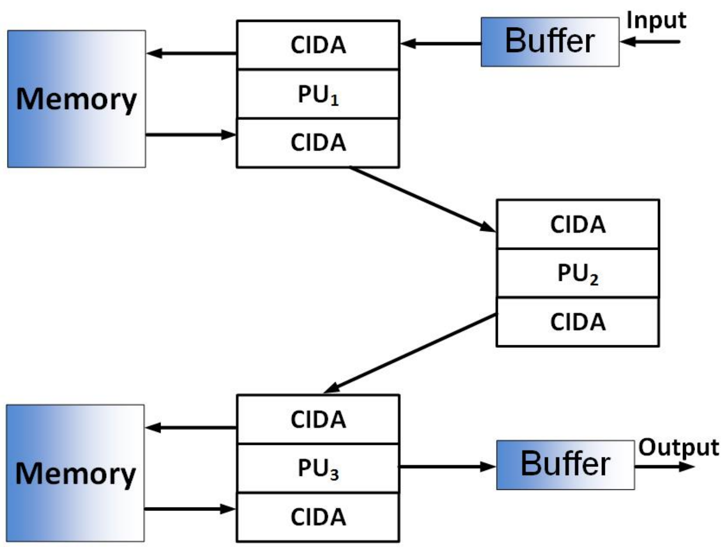


Figure III.3: Organization of data access. The designer specifies only the source and sink of data using the CIDA interface. The system coordinates the data transfer and access to share memories.

check for compatibility of component interaction protocols and devise the entire data management infrastructure. For each module in the design, the user just needs specifying the source of data and result target. The system takes care of the data transfer, buffer instantiation and scheduling of memory access for shared memory components. Figure III.3 shows an abstract component composition using CIDA-Interface modules. Incoming image data are temporally stored into an input buffer. A processing unit PU_1 reads image from the input buffer, performs some processing and copy results in memory. From there PU_2 reads the processed image and performs further computations. The result is sent to PU_3 where it is mixed with stored data in memory to produce the output. All components use CIDA at their boundary to handle the data transfer and memory scheduling. This allows designers to focus on implementing their components and let the system take care of data exchange. The example in Figure III.4 shows an automatic RTL mapping of an abstract specification that uses CIDA for data access of hardware modules. Data from DDR memory supplies the first module of a processing chain. Subsequent computations are done in the chain and the last module copies the result back into the DDR memory. The use of CIDA allows for the generation of the two input and output buffer as well as temporary buffers among the module to meet the timing requirements. A scheduler is automatically devised to arbitrate the transactions between CIDA interfaces and memories.

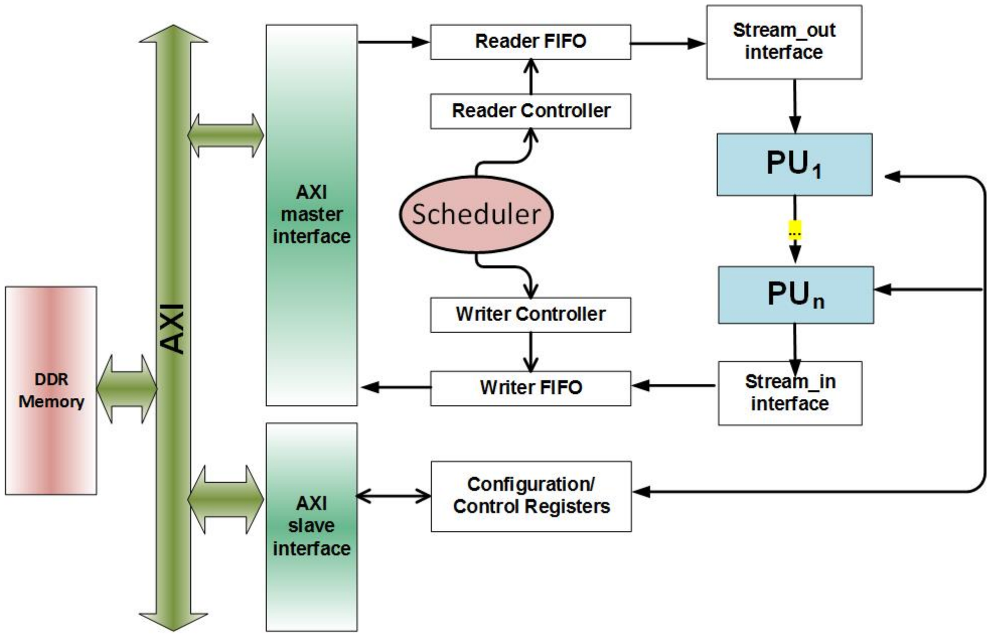


Figure III.4: System integration with CIDA; abstract representation (left) and concrete FPGA-Implementation (right)

III.3.1 Basic Hardware Modules

One of the main goals of our design environment is to reduce design time of system-on-chip for video applications. Leveraging common computing structure of those applications to provide a generic implementation of components commonly used along with a way of binding them will reduce the design time. Many applications can simply be composed and verified using templates available at various levels of hierarchy. We have populated our design environment with generic components (functions, data access managers, communication components) needed to build most image processing applications. The functions are fully parameterizable according to the picture size, the lighting conditions, applied threshold, etc. New and more complex functions can be used to populate the framework. For each module a SystemC/TLM and an equivalent RTL-version is available to allow a seamless mapping from SystemC/TLM descriptions into RTL ones. This step is currently conducted by hand, but our future work will seek to automate this process, with efficient design space exploration strategies. In the next section, we provide a brief explanation of the modules currently available in our library.

Buffers: Buffers are very important in image processing for temporally holding data needed by components not having direct access to memories. In systems in which pixels are delivered sequentially from image sensors, buffers can be used to capture neighborhoods of incoming pixels to perform some low-level operations such as convolution or simple filtering. Buffers can also be used to control timing or to multiplex memory access between components. Our implementation provides a wide range of buffers that can be configured for various purposes, including synchronization, temporary data storage, and timing. The configuration defines the number, size and type ports, as well as the buffer size.

Convolution Filters: Many low-level morphological operations in image processing rely on convolution. Noise reduction, smoothing, edge detection, median, and averaging all operate with convolution filters. Our framework provides a generic description of the buffer, including the number of lines, the sliding window structure which captures the neighborhood the current pixel and the pixel function to be applied in the convolution.

Minimum and Maximum Brightness: To calculate the image's minimum brightness, a generic module providing the lowest in a series of values is implemented. So, mathematically speaking, this means calculating

$$brightness_{min} = min(P) \tag{III.1}$$

where P is the set of all pixels generated by

$$\begin{aligned} P := \{p_{x,y} \mid \forall x \in [0, imagewidth - 1]\,, \\ \forall y \in [0, imageheight - 1]\}. \end{aligned} \tag{III.2}$$

A buffer is used to store the image pixel. The computation is done either sequentially by streaming all elements through the module to compute the minimum or maximum value.

Average Brightness: The general approach is to sum the values and then divide the overall sum by the number of values. For the set of all pixels P, this means calculating

$$brightness_{average} = \frac{\sum\limits_{p \in P} p}{|P|}. \tag{III.3}$$

Due to the complexity and cost of hardware implementation, the division is replaced by subtracting the number of values from the overall sum whenever possible. If subtracted, a separate counter is incremented to keep track of the number of performed subtractions. The counter then holds the result of the division at the end of the computation.

Thresholding: Taking both minimal and average brightness from the last frame as input, this module calculates and applies the current threshold. The result is an inverted binary image corresponding to the original image and the threshold. The set B of all binarized pixels is calculated by

$$B := \left\{ p'_{x,y} \;\middle|\; \begin{matrix} p'_{x,y} = 0 & \forall p_{x,y} < threshold \\ p'_{x,y} = 1 & \forall p_{x,y} \geq threshold \end{matrix}, \quad p_{x,y} \in P \right\}. \tag{III.4}$$

For any value streaming through the module, the calculated threshold based on minimum and average brightness is applied. Any value below the threshold is converted to 1, any value above is converted to 0.

Integral Image: This module is responsible for converting the pixel stream from binary pixels to the corresponding integral image. It calculates the set I using the equation:

$$I := \left\{ p'_{x,y} \;\middle|\; p'_{x,y} = \sum p_{m,n}, \quad m \in [0, x], n \in [0, y], p \in B \right\}. \tag{III.5}$$

It buffers one line of the image, which is initialized with zeroes. In addition, the sum of original pixels left of the pixel under consideration is calculated. With these two sources, the module calculates the integral image pixel by adding the processed pixel directly above it to the sum of original pixels left of it (including its own value in the original image). The result is buffered to form the basis of the subsequent line and then sent to the output stream.

Sum of Environment: Although usually in fixed positions from the center of the rectangular environment, the position of the coefficients might change due to the environment overlapping the image borders. This module calculates the set S using

$$S := p'_{x,y} \middle| p'_{x,y} = \sum p_{m,n},$$

$$\begin{cases} m & \in [max(0, x - env_{width}), min(x + env_{width}, env_{width})] \\ n & \in [max(0, y - env_{height}), min(y + env_{height}, env_{height}] \\ p & \in B \end{cases} \tag{III.6}$$

The process is divided into several stages. In general, a number of lines are buffered as required by the size of the mask. As a first step, the relative position of the coefficients is calculated by checking for overlaps. Afterwards, the resulting positions are retrieved from the buffer using $Result = A - B - C + D$

Camera Calibration: Camera calibration is an important part of image processing, in particular in distance measurement and 3-D reconstruction. Because the projected image in the camera does not have any linear relation with the real world, we need to transform the image in order to remove these non-linearities by performing an inverse perspective mapping. This mapping consists of removing the distortion of the camera lens, rotating the image to reach a perfect horizontal level in relation to the ground and performing a translation to adjust the scale. These operations can be expressed by the following equation:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = KTR \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{III.7}$$

Where $(u, v, 1)^T$ and $(x, y, z, 1)^T$ denote respectively the image expressed in homogeneous coordinates and the real world expressed also in homogeneous coordinates. K is the camera intrinsic parameter matrix, T is the translation matrix and R is the rotation matrix. Our framework implements perspective mapping of [11], which makes it easier to deploy in systems where automatic calibration are required. To calibrate a car for driving assistance for instance, a quadrilateral pattern can be used on the road with known dimension. With the fixed position of the camera on the car, the calibration can be done and intrinsic camera parameter extracted to compute the linearity in future images. Figure III.5 shows the inverse perspective mapping for a camera setup that we used on a RC self-driving car. Many more components such as Harris corner and segmentation

Figure III.5: Inverse perspective mapping. The Original image (left) and transformed image (right)

whose description will consume too much room are available in our framework. Figure III.6 presents the compilation results of the previous described modules for the Xilinx Zynq-FPGA.

Module	Slices	Flip Flops	RAMB16s	Max Freq.
Serializer	96	104	0	234.71 MHz
Minimum	37	49	0	245.27 MHz
Average	110	81	0	144.97 MHz
Threshold	44	59	0	221.02 MHz
Integral	210	244	2	177.97 MHz
Integral Sum	521	458	16	133.79 MHz
Deserializer	87	97	0	202.36 MHz
Sobel	263	965	0	198.33 MHz
Harris Corner	742	287	0	179.63 MHz

Figure III.6: Synthesis results for each implemented module. All modules are capable of running well over 100 MHz. The *Serializer* and *Deserializer* modules are connected to a 64-bit-wide CIDA for pixel-wise data communication. This way, each intermediate module deals with single pixels on the bus, rather than having to redundantly deal with the extraction / combination individually.

III.4 Case Studies

We used 3 case studies to demonstrate the capability of our framework. The first case study is a complete driving assistance system, which was developed entirely in just in a couple of days, including software/hardware partitioning. Our experiences from similar projects show that without our framework, this project would have taken at least 3 months to complete. The remaining two projects are used to demonstrate the simulation speed and the correctness of result across all levels of the design flow.

III.4.1 Driving Assistance System

The goal of this case study is to detect obstacles on the road and their distance to the car, and provide warning to the driver in real-time. The method used is based on optical flow computation, which can be estimated by matching points across images. Given point (u_x, u_y) in image I_1, the goal is to find the point $(u_x + \sigma_x, u_y + \sigma_y)$ in image

I_2 that minimizes ε with :

$$\varepsilon(\sigma_x, \sigma_y) = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (I(x,y) - I(x+\sigma_x, y+\sigma_y)) \tag{III.8}$$

The procedure first finds features in an image and track them in subsequent images. The version implemented in OpenCV consists of the following steps: 1) features detection, 2) tracking, and 3) processing.
Feature Selection searches for feature points, mainly corners in images, usually based on the Harris corner approach. This function computes the minimal eigenvalue for every source image pixel and rejects the corners with eigenvalue below a predefined threshold.
Optical flow computes the optical flow vector between a point in two consecutive images, the location of this point must be determined in the sequence of frames. This is done with iterative Lucas-Kanade method with pyramids.
Processing: Prior to the depth estimation of obstacles, a camera calibration is needed to transform the image in order to remove the non-linearity by performing an inverse perspective mapping. Using the transformed image it becomes easy to make distance measurements with the relation pixels/meter, which is well defined by the transformation. Combined with the optical flow, we now have a method to detect obstacles as well as their distance to the camera. The proposed algorithm was implemented in software (SW) using OpenCV in our emulation platform. The implemented architecture consists of reading images in Bayer-format from the camera, convert them into RGB and store them into memory where they can be accessed and processed by the OpenCV algorithm running on one of the embedded ARM processor present in a Zynq-FPGA. The system works at 100 MHz. We profiled the algorithm using the Oprofile[10] profiler running on the embedded ARM processor. The total run-time for the algorithm in SW only was 354 ms, 70% of which was devoted to compute the corners. Using this profiling information, the partitioning was obvious. The Harris corner was moved into hardware and the rest unchanged on software, leading to a 12X performance improvement of the system.

III.4.2 Line Segment Detection Using Weighted Mean Shift(LSWMS)

In this case study, a new algorithm for line segment detection using weighted mean shift procedures [12] is prototyped. The processing chain consists of image acquisition, color to gray conversion followed by a Sobel edge detection, and weighted

Device utilization Zynq 7z020clg484-1			
Case study	Slices Registers	Slices LUTs	BRAM
Driving A. SW	5347 (5%)	6080 (11%)	34 (24%)
Driving A. HW/SW	5601 (5%)	6930 (13%)	38 (27%)
LSWMS	4747 (4%)	5380 (10%)	34 (24%)
Segmentation	606 (0%)	11710 (22%)	7 (5%)

Table III.1: Resources utilization of our case studies in the emulation platform. The overall percentage of utilization is less than 28% which gives enough resources for further acceleration.

mean shift segment detection, all of which was instantiated from our framework and verified across all design levels. Table III.2 and Figure III.7 show the results of the computation at different levels of specification. The original image (left) is used as test pattern across all design level. The pure software execution on a 1.2 GH dual-core processor results in 0.27 second per frame (spf). The hardware/software partitioning at TLM level is at 672 spf very closed to the RTL simulation at 770 spf. This is due mostly to the QEMU simulator in which the hardware modules are represented in RTL-manner, but only the transaction in TLM makes the difference. The emulation in FPGA with just 50MHz clock speed has a better performance (0.23 spf) than the high-level simulation. As shown in Figure III.7, the results of the segmentation are the same across all levels, with a much faster emulation in FPGA.

Timing information				
Case study	OpenCV	SystemC-TLM	SystemC-RTL	FPGA
LSWMS	0.27s	672s	770s	0.23s
Segmentation	0.02s	0.27s	2.7s	0.01s

Table III.2: Duration in seconds for processing a frame at different levels.

III.4.3 Segmentation

Segmentation is at the center of many computer vision applications. The segmentation of Kim and Chalidabhongse [9] is used and implemented entirely in hardware. Table III.2 and Figure III.8 illustrate the results at various level of the design flow. The simulation of the hardware/software system at TLM performs at 0.01 spf on a dual core 1.2 GHz processor, while the emulation in FPGA results in 0.01 spf on a 50MHz

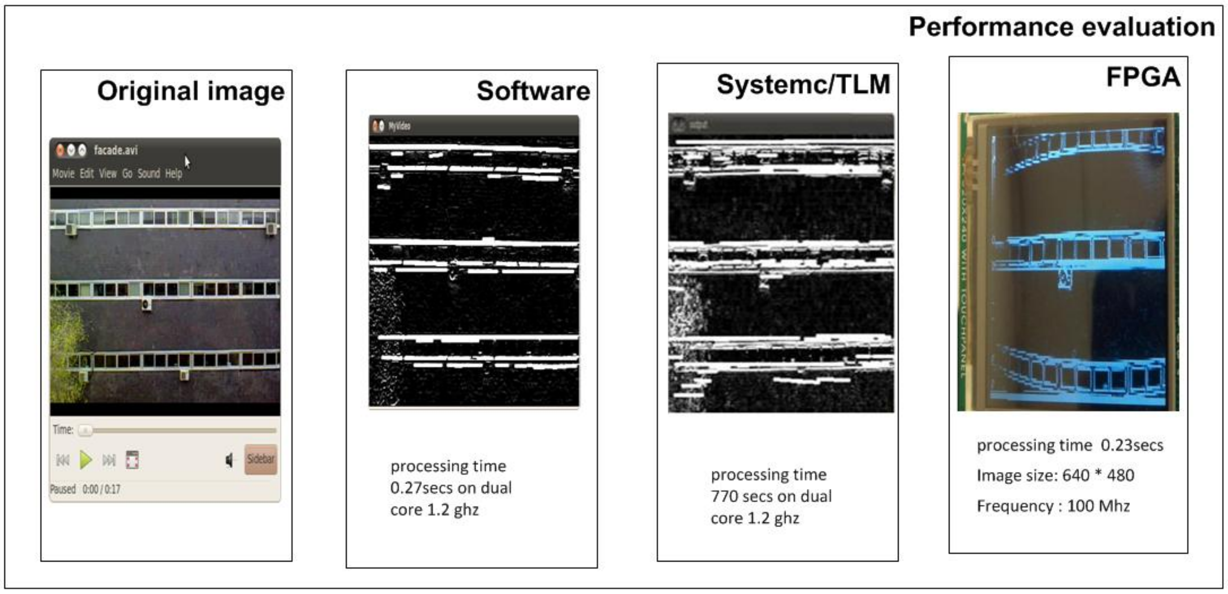


Figure III.7: Line segment detection using weighted mean shift. From left to right we have the input image, the output of the OpenCV algorithm (initial specification) running on a workstation, the output of the SystemC SoC prototype and the output of the emulation into FPGA.

clock system. The simulation at RTL-level is 10 times faster than the RTL, in this case due to the lower complexity of the segmentation. Once again the computations at all levels produce the same results. The two previous examples were implemented and verified in less than a day by an undergraduate student with limited experience in hardware design.

III.5 Conclusion

In this paper, we presented a framework to facilitate the design of system-on-chip solutions for video processing application. By leveraging structures of video processing applications and populating our framework at various level of design hierarchy with basic components, we showed that it was possible to considerably reduce design time for very complex systems. We provided the necessary glue (CIDA) to connect a software for image processing designs (OpenCV) with a system-on-chip specification tool (SystemC), and verification and emulation frameworks (QEMU, UVM). The result is a framework that can considerably reduce time-to-market not only of video-based

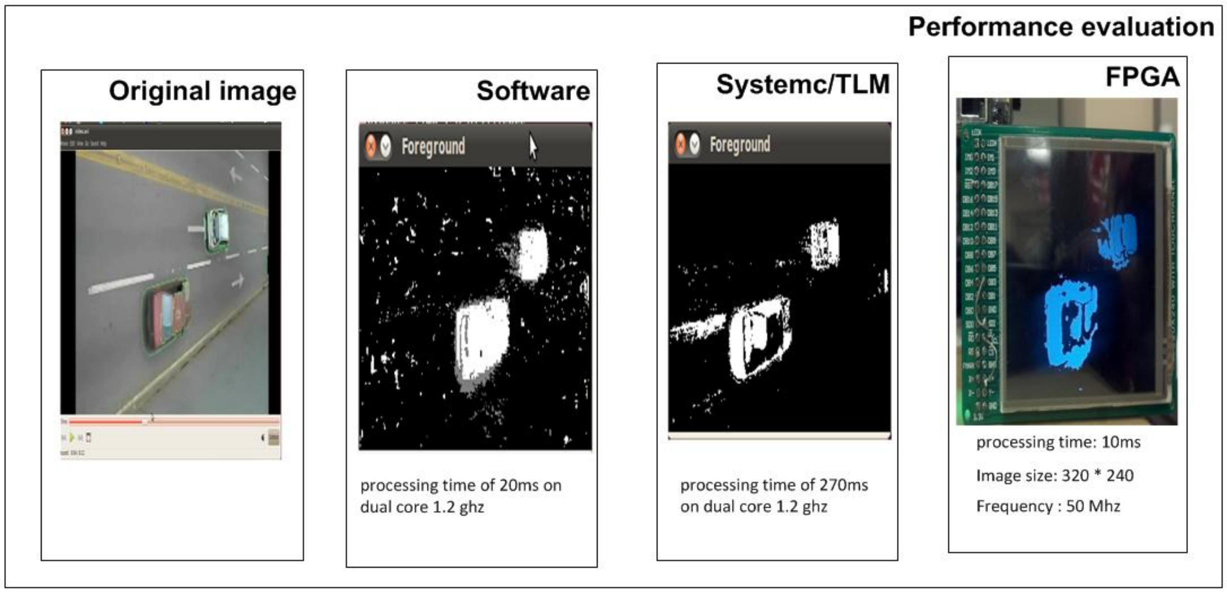


Figure III.8: Segmentation implementation. From left to right we have the input image, the output of the OpenCV algorithm running on a PC station, the output of the SystemC SoC prototype and the output of the emulation into FPGA.

hardware/software systems, but systems-on-chip in general. Our future work will provide formalism for CIDA along with design space exploration for optimal buffer and data access synthesis from a formal specified design.

Bibliography

[1] Generic and automatic specman based verification environment for image signal processing ips. URL: `design-reuse.com/articles/20907/specman-verification-image-signal-processing-ip`.

[2] Tlmu. URL: `edgarigl.github.io/tlmu.`

[3] Using intel ipp with opencv 2.1. URL: `http://software.intel.com/en-us/articles/using-intel-ipp-with-opencv.`

[4] L. Alfaro and T. Henzinger. Interface-based design. 195:83–104.

[5] K. Appiah, A. Hunter, T. Kluge, P. Aiken, and P. Dickinson. Fpga-based anomalous trajectory detection using sofm. In *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ARC '09, pages 243–254, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] M. Bramberger, A. Doblander, A. Maier, B. Rinner, and H. Schwabach. Distributed embedded smart cameras for surveillance applications. *IEEE Computer Society*, 39:68–75, 2006.

[7] P. Chen, P. Ahammad, C. Boyer, S.-I. Huang, L. Lin, E. Lobaton, M. Meingast, S. Oh, S. Wang, P. Yan, A. Y. Yang, C. Yeo, L.-C. Chang, D. Tygar, and S. S. Sastry. Citric: A low-bandwidth wireless camera network platform. *Second ACM/IEEE International Conference on Distributed Smart Cameras*, pages 1–10, 2008.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[9] K. Kim, T. H. Chalidabhongse, D. Harwood, and L. Davis. Real-time foreground-background segmentation using codebook model. *Real-Time Imaging*, 11(3):172–185, June 2005.

[10] J. Levon. Oprofile - a system profiler for linux. URL: `http://oprofile.sourceforge.net/news/.`

[11] A. Muad, A. Hussain, S. Samad, M. Mustaffa, and B. Majlis. Implementation of inverse perspective mapping algorithm for the development of an automatic lane tracking system. In *TENCON 2004. 2004 IEEE Region 10 Conference*, volume A, pages 207–210 Vol. 1, Nov 2004.

[12] M. Nieto, C. Cuevas, L. Salgado, and N. Garcia. Line segment detection using weighted mean shift procedures on a 2d slice sampling strategy. *Pattern Anal. Appl.*, 14(2):149–163, May 2011.

[13] J. Park, B. Lee, K. Lim, J. Kim, S. Kim, and Kwang-Hyun-Baek. Co-simulation of systemc tlm with rtl hdl for surveillance camera system verification. In *Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on*, pages 474–477, 2008.

[14] B. Rinner, T. Winkler, W. Schriebl, M. Quaritsch, and W. Wolf. The evolution from single to pervasive smart cameras. In *Distributed Smart Cameras, 2008. ICDSC 2008. Second ACM/IEEE International Conference on*, pages 1 –10, 2008.

[15] S. Saha. *Design methodology for embedded computer vision systems.* PhD thesis, 2007.

[16] H. Samsom, F. Franssen, F. Catthoor, and H. De Man. System level verification of video and image processing specifications. In *System Synthesis, 1995., Proceedings of the Eighth International Symposium on*, pages 144–149, 1995.

[17] J. Schlessman, C.-Y. Chen, W. Wolf, B. Ozer, K. Fujino, and K. Itoh. Hardware/software co-design of an fpga-based embedded tracking system. In *Computer Vision and Pattern Recognition Workshop, 2006. CVPRW '06. Conference on*, pages 123–123, 2006.

[18] A. Trost and A. Zemva. Verification structures for design of video processing circuits. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 222–227, 2012.

[19] S. Wu. Implementing an automated checking scheme for a video-processing device. URL: `www.cadence.com/rl/Resources/conference_papers/1.13Paper`.

[20] Xilinx. Accelerating opencv applications with zynq using vivado hls video libraries. URL: `http://www.xilinx.com/support/documentation/application_notes/xapp1167`.

APPENDIX

I, certify that Michael Mefenza Nentedem is the first author of the article *"A Framework for Rapid Prototyping of Embedded Vision Applications"* and have completed at least 51% of the work in the article.

Name:____________________ Signature: ____________________ Date: ____________________

IV Automatic UVM Environment Generation for Assertion-based and Functional Verification of SystemC Designs

Michael Mefenza, Franck Yonga and Christophe Bobda

Abstract— This paper presents an approach for reducing testbench implementation effort of SystemC designs, thus allowing an early verification success. We propose an automatic Universal Verification Methodology (UVM) environment that enables assertions-based, coverage driven and functional verification of SystemC models. The aim of this verification environment is to ease and speed up the verification of SystemC IPs by automatically producing a complete and working UVM testbench with all sub-environments constructed and blocks connected. Our experimentation shows that the proposed environment can rapidly be integrated to a SystemC design while improving its coverage and assertion-based verification.
Keywords— UVM, Verification, SystemC/TLM.

IV.1 Introduction

SystemC [1] is widely used for system level modeling of systems-on-chips. It is a C++ library that provides additional constructs to capture concurrency, time, and hardware data types, thus allowing the language to be used for system architecture modeling, hardware/software co-design with early integration of hardware and software. After the SystemC model of a circuit is developed, test data are generated for simulation of the model and the simulation results are observed to verify the functionality of the model. The general practice is to simulate a model with a very large amount of test data given by a designer. The generation of this test data is a very time-consuming and a labor-intensive task Simulation is very time consuming and does not usually captures run-time behavioral properties of systems as functional coverage and assertion checking do. Code coverage is mostly used to check how far verification has reached the functions of the design, while assertion check is used to prove or discard some design properties. Assertion-Based Verification (ABV) has proven to enhance design quality and verification time tremendously [2]. While simulation, even though slow, is well supported in SystemC, implementing coverage and assertions in SystemC is an error prone process due to the limited assertion capabilities of the class library. This is especially true for IP-integration, the complex interfaces and protocols, implemented in IPs require advanced assertions not directly supported in SystemC. This limitation can be overcome by using a SystemC extension such as SCV (SystemC Verification) or by integrating an independent function verification environment such as the universal verification methodology (UVM) [3]. SCV provides constructs for test-pattern generation, randomization, input constraints and automatic result evaluation, but lacks a framework for assertion-based verification. UVM is a framework that facilitates testbench generation and provides key features for functional coverage and assertion-based verification through SystemVerilog support. A combination of SystemC and UVM would provide a solid framework for efficient design specification with seamless and efficient verification, leading to reduced time-to-market of electronic systems. While the integration of UVM with SystemC is covered by Electronic Design automation (EDA) tools, verification engineers are required to design, identify how to connect, connect manually UVM components for a SystemC model. Some works have tried to adapt UVM to SystemC by re-implementing UVM concepts in SystemC. [4] and [5] introduce the System Verification Methodology (SVM) Library as an advanced TLM library for SystemC, which is based on a SystemC implementation of a limited Open Verification Methodology (OVM) subset. However, OVM was enhanced to the Universal Verification Methodology (UVM) with many improvements compared to OVM such as more control over the simulation phases. Verification engineers are still required to design, connect manually verification components and these works are not available and have not been tested on several designs as UVM; that makes a potential comparison difficult.

In this paper, we present an automatic UVM-Verification environment generation system that exploits the hierarchy information in a SystemC model, automatically produces a UVM testbench with coverage and assertions, and relieves the modeler of the time-consuming task of test data development. The system creates a Design Under Test (DUT)-specific UVM template testbench for SystemC DUT. Our approach intends to be generic and covers all kinds of SystemC DUT. Our experimentation shows that the proposed environment can easily and rapidly be integrated to a SystemC design while improving its coverage and assertion-based verification.

The rest of the paper is organized as follows: In section IV.2, relevant works in the verification of SystemC-based designs are presented. A conceptual view of the proposed UVM environment is presented in section IV.3. Section IV.4 presents the model adopted for the integration of UVM capabilities into SystemC/TLM designs. In section IV.5, the verification flow is explained. The evaluation of the proposed environment on several SystemC designs is done in section IV.6. Finally, section IV.7 concludes the paper.

IV.2 Related work

This section presents related work in the verification of SystemC-based designs. These works can be classified as coverage-driven verification and assertion-based verification.

Regarding ABV in SystemC, [6] presents the implementation of a SystemC assertion library, but the assertion library exists only for *bit* and *sc_logic* signal types. The SystemC Verification library (SCV)[7] was introduced, in order to support constrained-random stimuli techniques for RTL verification. Some works have extended the SCV library by improving or providing missing features for assertions in the SystemC language [8] [9]. The system in [8] extends the SystemC library with Assertion Based Verification (ABV) using SystemVerilog language. To do so, constructs, with the same syntax and semantics of SVA, are translated into external SystemC modules connected to the original design. However, this work supports only a partial subset of SVA.

Some works have tried to extend coverage in the SCV library. In [10], the Design Under Test (DUT), is simulated as usual in the verification environment and a Value Change Dump (VCD) file is generated. The VCD and a Coverage Input File (CIF) are passed to a coverage calculation module. CIF defines coverage groups using a syntax similar to the SystemVerilog language. In their approach, the generation of the test bench and the ability to stop the simulation only after reaching certain coverage is not possible because the coverage is done after simulation. Several simulation runs could be necessary to achieve a specific coverage. In [11], [12] and [13], the SystemC SCV library is extended to achieve functional coverage. However, the proposed approach does not include all features of SystemVerilog coverage such as cross coverage or illegal bins.

Other works have tried to adapt UVM to SystemC by re-implementing UVM concepts in SystemC. [4] and [5] introduce the System Verification Methodology (SVM)

Library as an advanced TLM library for SystemC, which is based on a SystemC implementation of a limited Open Verification Methodology (OVM) subset. However, OVM was enhanced to the Universal Verification Methodology (UVM) with many improvements compared to OVM such as more control over the simulation phases.

Although verification extensions for standard SystemC are meaningful, a library offering efficient and interoperable components for testbench development, like in UVM, is a suitable alternative to facilitate the verification of SystemC designs. Our goal in this work is to provide an approach for using UVM testbenches to ease the verification of SystemC/TLM designs using assertions and coverage.

IV.3 Proposed UVM environment for SystemC/TLM designs

In order to test the design model, the test must drive a stimulus into the design under test, and examine the results of that stimulus. The stimulus and checking can be done at the boundaries of a design, or it can examine the internals of design. This is referred to as black box versus white box testing. A white box test relies on the internal structure of the logic. Any changes to the device may affect the test; it tends not to run on a design in multiple abstraction levels, since the internal workings of different abstraction level may be different. In the black box case, the test is limited to examining the input and output signals, and determining if the model is working correctly based only on information gathered from the outputs; therefore, it may run on a block that is designed as a behavioral or gate-level model without modifications. It is re-usable as the scope of the design changes. In this work, we introduce the integration of UVM capabilities into SystemC/TLM based on a black-box approach. We present the automatic generation of the UVM verification environment, which supports functional coverage and assertion evaluation. As show in Figure IV.1, from a SystemC description of a hardware, a SystemC parser is used to gather information on the component interface. The user can provide assertion and metrics for coverage separately. Our framework (environment generator) then generates the components (sequencer, driver, monitor, coverage and assertion metrics for the functional verification) needed for the verification and connects all the blocks together to build the UVM environment. The verification can then be performed with signals generated in the UVM to feed designs in SystemC. The results are gathered back in UVM using the monitor. The generated assertion component

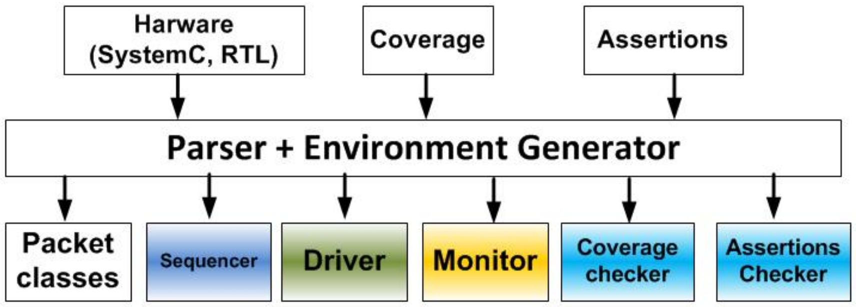


Figure IV.1: UVM testbench generation for a SystemC design. Using a SystemC specification, coverage and assertions, we generate all the necessary components in UVM. If not specified, coverage metrics are automatically generated.

then checks for bugs, while the coverage module measures the quality of verification. Figure IV.2 illustrates this approach. The sequencer is used to generate the inputs for the system and the driver mimics protocol of real-life communication components such as UART, USB. The monitor gathers the output signals from the system under test which are then evaluated for correctness by the assertion checker. The coverage checker is used to measure the verification coverage, a measure of quality used in the industry. Because the design under test is described in SystemC and UVM is based on SystemVerilog, we use the UVM Connect (UVMC) component to bridge the two descriptions. The UVM environment is encapsulated with SystemC/TLM module such that it is possible to drive the module with either sequences from the UVM environment or sequences from SystemC. This allows to verify the module either independently or as part of system-on-chip specified in SystemC. The proposed environment reduces steps in the testbench, creation while providing high quality of the verification by using a combination of coverage and assertions. UVM will automatically generate a testbench based on random stimulus to drive the DUT. The UVM environment can be configured to stop after a certain number of sequences or to stop after a percentage of coverage has been reached. This is useful depending on the verification test plan. The DUT encapsulated with the UVM environment can also be configured to be driven by a SystemC testbench. In that case, the UVM stimuli are ignored and UVM is only used for coverage and assertion-based verification. Our framework generates the complete UVM architecture, including the components explained in details the following sections.

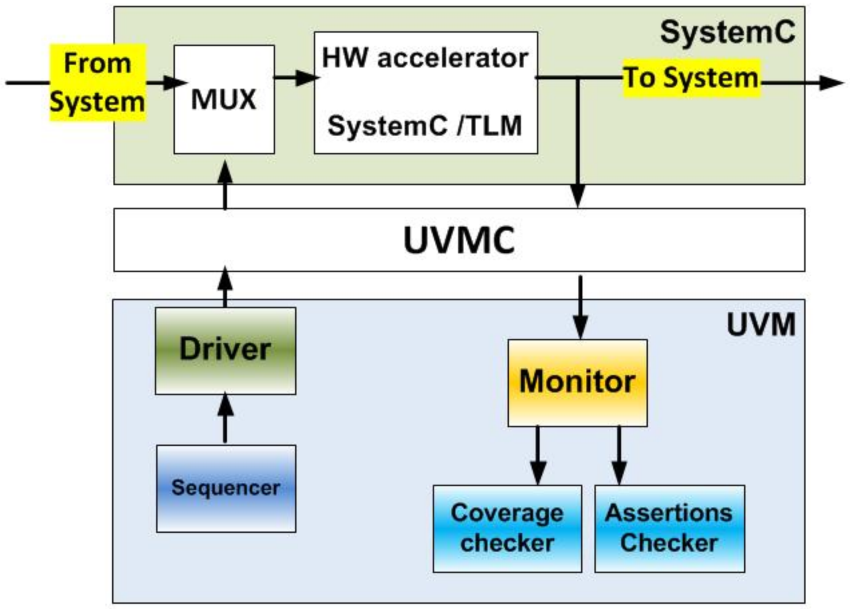
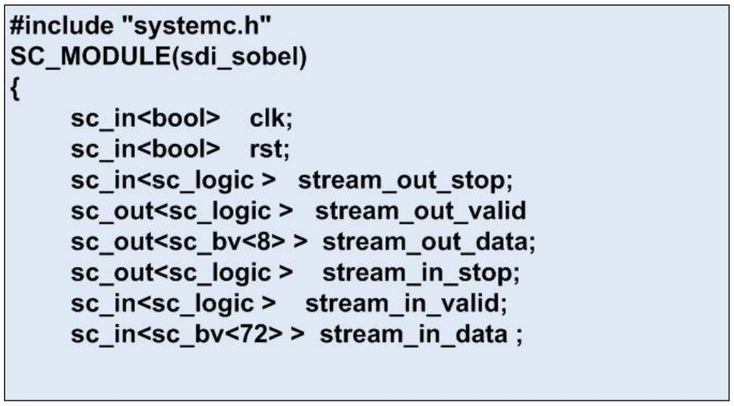


Figure IV.2: Verification environment. It is automatically encapsulated on top of the SystemC DUT.

IV.4 Generated components

IV.4.1 Packet classes

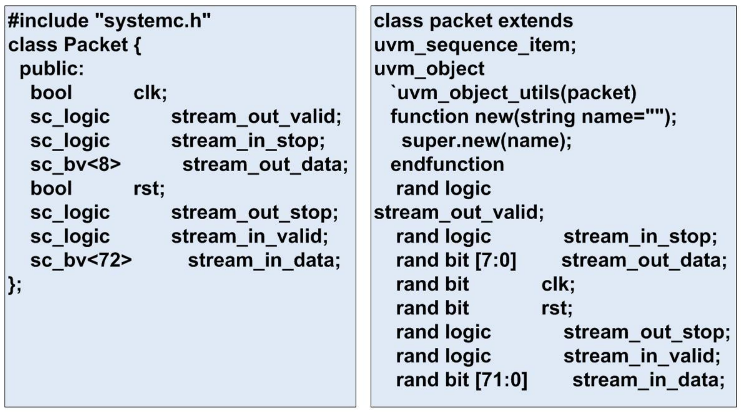
The UVM environment has to drive input ports of a design and observe output ports. A SystemC parser is used to retrieve the netlist (inputs and outputs) of the DUT. SystemC provides structures, such as *sc_in* and *sc_out*, to identify inputs and outputs. The netlist is used to generate one class *Packet* in SystemC and one class *packet* in UVM. A packet is a data container combining the inputs and the outputs of the DUT. The class packet in UVM is used to generate sequences in the UVM environment. Sequence item represents data for the stimulus of the DUT. The stimulus can represent a command, a bus transaction, or a protocol implemented inside the DUT. A sequence item may be randomized to generate different stimuli. UVM provides constructs to generate random and constrained packets. The class *Packet* in SystemC is used to drive input ports of the DUT and observe output ports of the DUT. The communication between a packet in SystemC and a packet in UVM is done using UVM Connect (UVMC) presented in section IV.4.4. Figure IV.3 shows an example of DUT in SystemC, It is an implementation of the Sobel convolution using a streaming data interface to receive his inputs and send out his outputs. The streaming data interface is a handshaking protocol that we implemented for streaming data between hardware accelerators. In that protocol, the input data (stream_in_data) is received when stream_in_stop is false and stream_in_valid is true; when stream_out_stop is false and stream_out_valid is true, the output data (stream_out_data) is sent. Stream_out_valid and stream_out_stop cannot be true at the same time. For the Sobel implementation, the input data must be less than 256 and the rst must be set at least for 3 clock cycles. All these properties are used later in the paper to derive the necessary coverage and assertion metrics. From this example, we extracted the netlist and produced the different classes as presented in Figure IV.4.



```
#include "systemc.h"
SC_MODULE(sdi_sobel)
{
      sc_in<bool>    clk;
      sc_in<bool>    rst;
      sc_in<sc_logic >   stream_out_stop;
      sc_out<sc_logic >   stream_out_valid
      sc_out<sc_bv<8> >  stream_out_data;
      sc_out<sc_logic >    stream_in_stop;
      sc_in<sc_logic >    stream_in_valid;
      sc_in<sc_bv<72> >  stream_in_data ;
```

Figure IV.3: Example of SystemC DUT. Sobel implementation with a streaming data interface.

A similar procedure is applied in case of Transaction Level Modeling (TLM) IP. In TLM, there are 2 types of sockets: initiator and target. An initiator socket generates a transaction and sends it to a target socket. A transaction is characterized by the type of transaction (read or write), the address, a pointer to the data to be transferred and the phase of the transaction. Figure IV.5 shows the structure of a packet used to generate the transactions for driving (in case of a target socket) or monitoring (in case of an initiator socket) a TLM IP. The field cmd specify the TLM command encodings (TLM_READ_COMMAND, TLM_WRITE_COMAND). The field phase specifies the TLM phase delimiters such as BEGIN_REQ, END_REQ ,TLM_OK_RESPONSE.



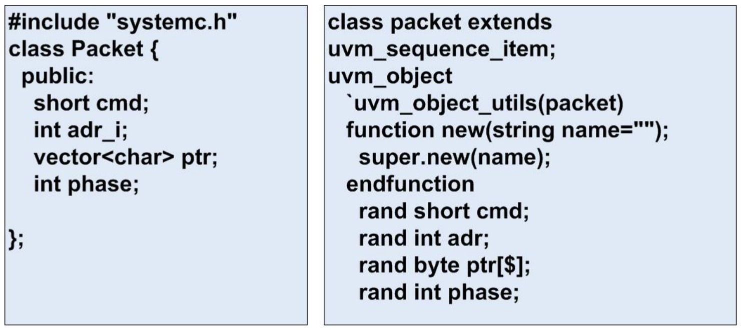
```
#include "systemc.h"
class Packet {
  public:
    bool        clk;
    sc_logic        stream_out_valid;
    sc_logic        stream_in_stop;
    sc_bv<8>         stream_out_data;
    bool        rst;
    sc_logic        stream_out_stop;
    sc_logic        stream_in_valid;
    sc_bv<72>        stream_in_data;
};
```

```
class packet extends
uvm_sequence_item;
uvm_object
   `uvm_object_utils(packet)
   function new(string name="");
     super.new(name);
   endfunction
    rand logic
stream_out_valid;
    rand logic          stream_in_stop;
    rand bit [7:0]      stream_out_data;
    rand bit           clk;
    rand bit           rst;
    rand logic          stream_out_stop;
    rand logic          stream_in_valid;
    rand bit [71:0]      stream_in_data;
```

Figure IV.4: Generated packet classes. On the left is the side structure of packet in SystemC for the DUT of Figure IV.3.On the right side is the structure of packet in UVM for the DUT of Figure IV.3.

IV.4.2 Sequencer

The sequencer is responsible for the generation of sequences and the coordination between sequences and the driver. A sequence implements the procedure to create sequence items. It is a set of packets with specific values for each field of a packet. Figure IV.6 shows sequences used in the proposed example. It presents the generation of random and constrained packets. The stream_in_data is constrained to be less than 256 as specified in section IV.4.1. The sequencer sends a packet or transaction to the driver. The driver implements the function *seq_item_port.get_next_item* to indicate to the sequencer when it needs a new packet or transaction. It also implements the function

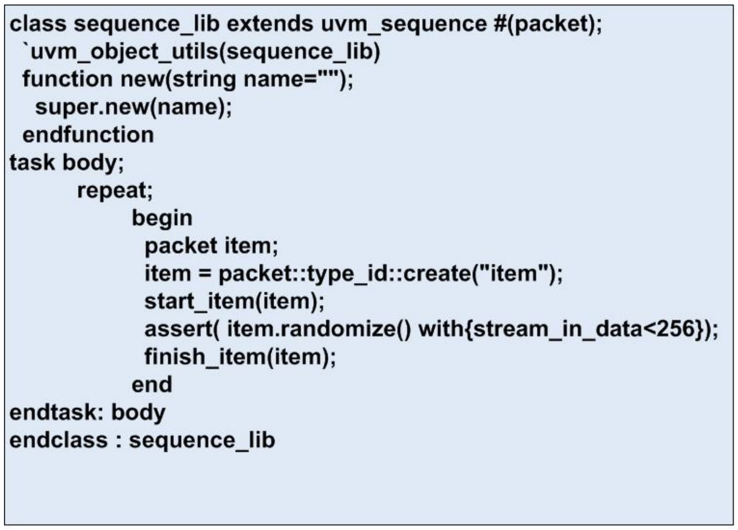


```
#include "systemc.h"
class Packet {
  public:
    short cmd;
    int adr_i;
    vector<char> ptr;
    int phase;

};
```

```
class packet extends
uvm_sequence_item;
uvm_object
   `uvm_object_utils(packet)
   function new(string name="");
     super.new(name);
   endfunction
     rand short cmd;
     rand int adr;
     rand byte ptr[$];
     rand int phase;
```

Figure IV.5: Generated packet classes for a TLM DUT. On the left is the side structure of packet in SystemC, on the right side is the structure of packet in UVM.



```
class sequence_lib extends uvm_sequence #(packet);
  `uvm_object_utils(sequence_lib)
  function new(string name="");
    super.new(name);
  endfunction
task body;
        repeat;
                begin
                  packet item;
                  item = packet::type_id::create("item");
                  start_item(item);
                  assert( item.randomize() with{stream_in_data<256});
                  finish_item(item);
                end
endtask: body
endclass : sequence_lib
```

Figure IV.6: Sequences Generation. Randomize() is used to generate random values. Stream_in_data is constrained to be less than 256 for the Sobel implementation.

seq_item_port.item_done to signal when it has finished to process a packet or a transaction. This mechanism, similar to a blocking transport interface, allows the synchronization between the sequencer and the driver.

IV.4.3 Driver

This module drives a packet or a transaction to the DUT ports. It receives sequence items and sends them to the DUT. The driver implements a function uvm_blocking_put_port to send a packet through UVM Connect (UVMC) to the SystemC side of the environment. We used a blocking port as a means for synchronization and this provides enough time to the DUT to process the packet.

IV.4.4 UVMC

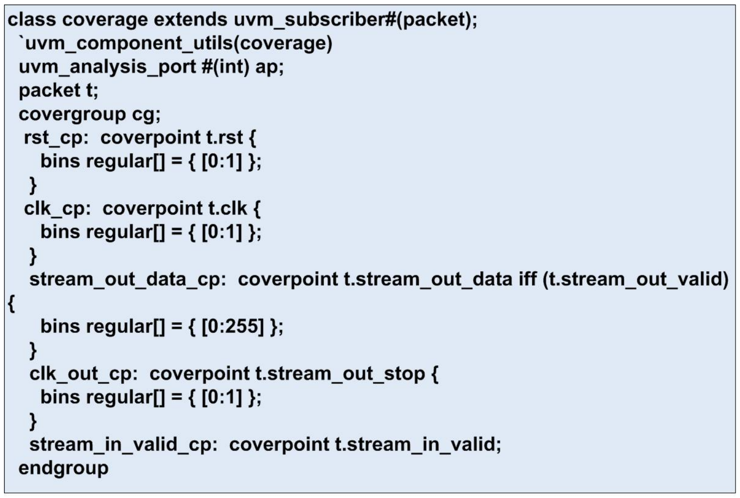
UVM Connect (UVMC)[14] from Mentor Graphics is an open-source UVM-based library that provides the communication between SystemC Components and SystemVerilog UVM Components using TLM connectivity between them. It also provides a means for accessing and controlling UVM simulation from SystemC. In the proposed environment, the communication from UVM to SystemC is done using *tlm_blocking_put_if* interface. This interface passes a packet from the driver to the SystemC/TLM hardware. *tlm_analysis_port* is used for the communication from SystemC to UVM. It sends a packet from SystemC/TLM hardware to the monitor. The use of TLM connectivity between UVM and SystemC provides a rapid simulation.

IV.4.5 Monitor

This module receives transactions and signals and makes them available to other components. The communication between the monitor and the coverage module is done using a simple *uvm_analysis_port* function. The analysis port is used to perform non-blocking broadcasts (from one entity to several entities) of transactions from a component to its subscribers. An analysis port can be connected to more than one component. A subscriber component provides an implementation of *uvm_analysis_imp port*. An *uvm_analysis_imp* receives all transactions broadcasted by a *uvm_analysis_port* and implements the analysis interface such as coverage collection. The monitor instantiates the assertions checker and maps it to the received transactions and signals.

IV.4.6 Coverage Checker

The coverage is integrated into the environment using SystemVerilog coverage properties. The SystemVerilog functional coverage constructs allow coverage of variables and expressions, as well as cross coverage between them. In SystemVerilog, the covergroup construct is used to specify a coverage model. Each covergroup can include a set of coverage points, cross coverage between coverage points and coverage options [15]. A coverage point can be a variable or an expression. Each coverage point includes a set of bins associated with its sampled values or its value-transitions. The bins can be explicitly defined by the user or automatically created during the generation of the UVM verification environment. The coverage model is implemented in a class *coverage*, subscriber component for the class *monitor*. We give to users the possibility to specify an external coverage input file to be used to generate the class *coverage*. SystemVerilog syntax must be used to define the coverage inside the file. The coverage input file is included inside the class *coverage* during the generation of the UVM environment. Figure IV.7 shows an example of class *coverage* for the DUT in section IV.4.1.A *sc_logic* variable should have only two possible values: 0 or 1. Therefore, the coverage of a sc_logic signals such as rst and clk is done with a bin [0:1].



```
class coverage extends uvm_subscriber#(packet);
  `uvm_component_utils(coverage)
  uvm_analysis_port #(int) ap;
  packet t;
  covergroup cg;
   rst_cp:  coverpoint t.rst {
     bins regular[] = { [0:1] };
    }
   clk_cp:  coverpoint t.clk {
     bins regular[] = { [0:1] };
    }
    stream_out_data_cp:  coverpoint t.stream_out_data iff (t.stream_out_valid)
{
     bins regular[] = { [0:255] };
    }
    clk_out_cp:  coverpoint t.stream_out_stop {
     bins regular[] = { [0:1] };
    }
    stream_in_valid_cp:  coverpoint t.stream_in_valid;
  endgroup
```

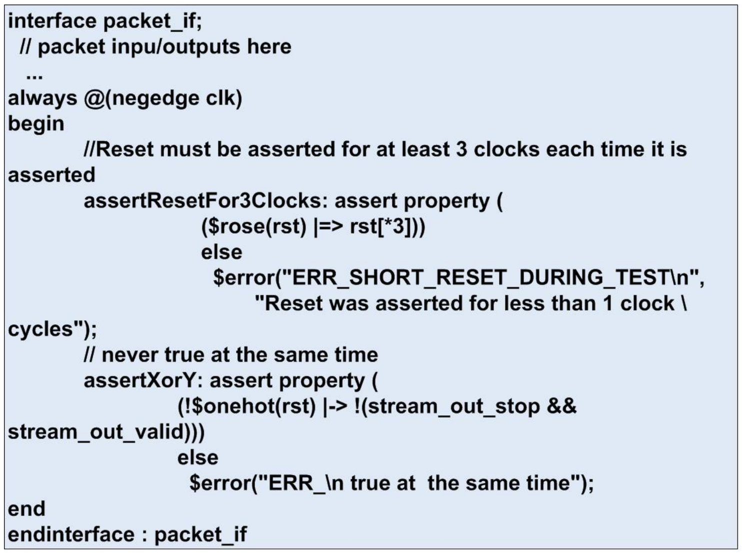
Figure IV.7: Coverage implementation in our environment. It shows the bins or set of values that we want to cover for each signal.

IV.4.7 Assertions Checker

Assertions are integrated into the environment using SystemVerilog assertions and are implemented to execute protocol or timing checking. They can be used to implement simple to-complex property/sequence checks for an interface or a protocol. The purpose of an assertion is to specify and check a set of properties that are expected to hold true in a given component. SystemVerilog provides two types of assertions: immediate and concurrent[16]:

- Immediate Assertions: are statements that include a conditional expression to be tested and a set of statements to be executed depending on the result of the expression evaluation.
- Concurrent Assertions: provide the means to specify sequential properties and to evaluate them at discrete points in time such as clock edges.

Figure IV.8 shows assertions implemented for the DUT in section IV.4.1. This figure presents two concurrent assertions evaluated at the negative clock edge. The first assertion specify the property that if the reset is set, it must be set for at least 3 clock cycles. The first assertion specify the property that Stream_out_stop and Stream_out_valid cannot be true at the same time in the streaming protocol as specified in section IV.4.1.



```
interface packet_if;
  // packet inpu/outputs here
   ...
always @(negedge clk)
begin
          //Reset must be asserted for at least 3 clocks each time it is
asserted
          assertResetFor3Clocks: assert property (
                         ($rose(rst) |=> rst[*3]))
                         else
                           $error("ERR_SHORT_RESET_DURING_TEST\n",
                                "Reset was asserted for less than 1 clock \
cycles");
          // never true at the same time
          assertXorY: assert property (
                      (!$onehot(rst) |-> !(stream_out_stop &&
stream_out_valid)))
                      else
                        $error("ERR_\n true at  the same time");
end
endinterface : packet_if
```

Figure IV.8: Assertion implementation in our environment. Rst must be set for at least 3 clock cycles. Stream_out_stop and Stream_out_valid cannot be true at the same time. These are 2 properties of the protocol implemented in the DUT.

IV.5 Verification Flow

The Verification is done in two phases:

- In a first phase, the UVM components are automatically generated from the DUT interface and a *uvm_env* class is used to instantiate and connect them to build the UVM environment.
- In the second phase, the UVM environment is compiled and simulated with an UVM simulator tool. During the simulation, Assertions are checked and coverage is collected for the specified DUT. Checking typically consists of verifying that the DUT Output meets the protocol specification.

The proposed environment can be used to verify new revisions or simple variations of SystemC models without needing to undo everything and redo them back again as long as the interface of the model is not changed. This environment can also be used to prove functional equivalence between an abstracted SystemC IP (the golden model) and the corresponding RTL implementation. Because the functional validation effort of high-level models (such as SystemC) compared to validation of RTL models is reduced, Systems-on-Chips design flow starts from a high-level specification. Once High-level models are verified, these models are refined into corresponding RTL implementations. Equivalence checking is used to guarantee the refinement correctness. We present a methodology based on simulation which relies on UVM to automate as much as possible the equivalence verification process between an abstracted SystemC IP (the golden model) and the corresponding RTL implementation by using:

- Output comparison. The same input stimulus should generate the same output for both implementations.
- Observability of assertions and coverage. The principle is that if a SystemC specification and the corresponding RTL implementation have the same interface, they should activate the same assertions and coverage points for the same input stimulus.

Given a SystemC specification S and its RTL implementation R, by applying a stimulus a_i on S and R, we denote S(a_i), $S_{cov}(a_i)$, $S_{abv}(a_i)$ respectively as the output, set of excised coverage points, set of excised assertions of the stimuli on the SystemC implementation; R(a_i), $R_{cov}(a_i)$, $R_{abv}(a_i)$ respectively as the output, set of excised coverage points, set of excised assertions of the stimuli on the RTL implementation. We define functional consistency as:

Definition 9 (functional consistency) *Given a SystemC specification S and its RTL implementation R, S and R are functionally consistent iff* $\forall$ *a set of stimulus ST =* $\{a_i, 1 <= i <= n\}$, $\forall a_i \in ST$, $S(a_i) = R(a_i)$, $S_{cov}(a_i) = R_{cov}(a_i)$ *and* $S_{abv}(a_i) = R_{abv}(a_i)$

This is true for designs with same interface (protocol) after refinement; the internals can differ. Figure IV.9 illustrates this approach; the same UVM environment drives a SystemC IP and the corresponding RTL implementation. Their outputs are collected inside the monitor and result comparison is performed by the scoreboard while the observability of assertions and coverage are done by the checkers.

IV.6 Experimental Evaluation

In order to assess the proposed verification framework, we used SystemC cores provided by Opencores [17]. All experiments have been conducted on an Intel Celeron 2.4 GHz machine with 2 GB RAM running Linux. The generated environment was compiled

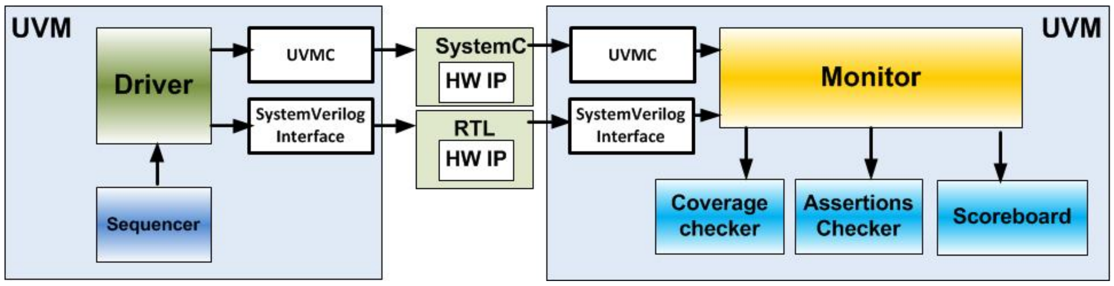


Figure IV.9: Functional consistency checking using the Proposed UVM environment.

and simulated using ModelSim from Mentor Graphics. Table IV.1 shows the number of lines in the DUT, the time (in seconds) to create the environment for the DUT, the CPU simulation time (in seconds) for the DUT, the coverage reached and the number of failed assertions for both UVM and SystemC testbenches. The UVM testbench has been configured to end after 3000 sequences. The SystemVerilog covergroups for the coverage were automatically derived from the DUT inputs and outputs and implemented using automatic bins in SystemVerilog. We derived the assertions to be checked by looking at the specification of the protocol implemented in the DUT and how it was implemented. RISC CPU is the SystemC implementation of a RISC CPU by Martin Wang. It includes several modules connected together such instruction cache, instruction fetch, instruction decoder and arithmetic logic unit. The UVM environment can be connected to any block. It can also act as a monitor (record inputs and outputs) when the DUT is driven by other blocks in the architecture. For the test, the environment was connected to the instruction fetch module.

SC DUT	Size	Tgen	Tsim	UVM testbench	
				Coverage(%)	Failed assertions
MD5	649	1	34	79	1
DES	4444	1	67	93	3
RNG	309	1	31	96	none
RISC_CPU	3019	1	54	80	one
FIR	652	1	38	79	none
USB	8873	1	98	80	5

Table IV.1: Experimental evaluation of the proposed environment. Size is the number of lines in the DUT, Tgen is the time (in seconds) to create the environment for the DUT, Tsim is the CPU simulation time (in seconds) for the DUT.

We can observe that the generation of the UVM environment is fast and practically does not scale with the size of the DUT. The results obtained from the statement coverage and assertions during simulation of the models confirm that the proposed UVM environment fully exercises the functionality of the models.

IV.7 Conclusion

An automatic Universal Verification Methodology (UVM) environment that enables assertions-based, coverage-driven, functional verification of SystemC models has been presented. The proposed environment is based on UVM-connect for the communication between SystemC designs and UVM. This environment speeds up the verification of SystemC IPs, both at module level and system level by automatically producing a complete and working UVM testbench, with all sub-environments constructed and blocks connected in the right way. Our approach can help verification engineers to leverage the time and effort for UVM testbench generation (manually design and connect UVM components for a specific model). Our experimentation showed that the proposed UVM environment fully exercises the functionality of SystemC models.

Bibliography

[1] IEEE Computer Society, “1666-2005 ieee standard systemc language reference manual,” URL: http://www.systemc.org., 2005.

[2] *Assertion-Based Verification*. Springer US. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-38152-7_13

[3] Accelera Systems Initiative, “Uvm,” URL: http://www.accellera.org/downloads/standards/uvm.

[4] C.Kuznik, W.Mueller, W.Ecker, V. Esen, and M. F. S. Oliveira, “A systemc library for advanced tlm verification,” *Design and Verification Conference (DVCon)*, 2012.

[5] M. F. Oliveira, C. Kuznik, H. M. Le, D. Grosse, F. Haedicke, W. Mueller, R. Drechsler, W. Ecker, and V. Esen, “The system verification methodology for advanced tlm verification,” in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS ’12. New York, NY, USA: ACM, 2012, pp. 313–322. [Online]. Available: http://doi.acm.org/10.1145/2380445.2380497

[6] V. Esen, T. Steininger, M. Velten, W. Ecker, and J. Smit, “Breaking the language barriers: Using coverage driven verification to improve the quality of ip,” *Design & Reuse.*

[7] Accelera Systems Initiative, “Systemc verification library,” URL: http://www.systemc.org/downloads/standards/.

[8] A. Habibi and S. Tahar, “On the extension of systemc by systemverilog assertions,” in *Electrical and Computer Engineering, 2004. Canadian Conference on*, vol. 4, 2004, pp. 1869–1872 Vol.4.

[9] Inc NextOp Software, “Nextop assertion-based verification,” URL: http://www.nextopsoftware.com/.

[10] P. Singh and G. Kumar, “Implementation of a systemc assertion library,” *Design & Reuse.*

[11] K. Schwartz, “technique for adding functional coverage to systemc,” *Design and Verification Conference (DVCon)*, 2007.

[12] G. Defo, C. Kuznik, and W. Muller, “Verification of a can bus model in systemc with functional coverage,” in *Industrial Embedded Systems (SIES), 2010 International Symposium on*, July 2010, pp. 28–35.

[13] S. Park and S.-I. Chae, “A c/c++-based functional verification framework using the systemc verification library,” in *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, 2005, pp. 237–239.

[14] Mentor Graphics, “Uvm connect,” URL: http://forums.accellera.org/files/file/92-uvm-connect-a-systemc-tlm-interface-for-uvmovm-v22/.

[15] A. B. Mehta, *SystemVerilog Assertions and Functional Coverage: Guide to Language, Methodology and Applications.* Springer Publishing Company, Incorporated, 2013.

[16] D. Bustan, D. Korchemny, E. Seligman, and J. Yang, “Systemverilog assertions: Past, present, and future sva standardization experience,” *Design Test of Computers, IEEE*, vol. 29, no. 2, pp. 23–31, 2012.

[17] “Opencores,” URL: www.opencores.org.

APPENDIX

I, certify that Michael Mefenza Nentedem is the first author of the article *"Automatic UVM Environment Generation for Assertion-based and Functional Verification of SystemC Designs"* and have completed at least 51% of the work in the article.

Name:____________________ Signature: ____________________ Date: ____________________

V Interface Based Memory Synthesis Of Image Processing Applications In FPGA

Michael Mefenza and Christophe Bobda

Abstract— Image processing applications are computationally intensive and data intensive and rely on memory elements (buffer, window, line buffer, shift register, and frame buffer) to store data flow dependencies between computing components in FPGA. Due to the limited availability of these resources, optimization of memory allocation and the implementation of efficient memory architectures are important issues. We present an interface, the Component Interconnect and Data Access (CIDA), and its implementation, based on interface automata formalism. We used that interface for modeling image processing applications and generating common memory elements. Based on the proposed model and information about the FPGA architecture, we also present an optimization model to achieve allocation memory requirements to embedded memories (Block RAM and Distributed RAM). Allocation results from realistic video systems on Xilinx Zynq FPGAs verify the correctness of the model and show that the proposed approach achieves appreciable reduction in block RAM usage.

Keywords— Interface, Vision, Memory synthesis, FPGA.

V.1 Introduction

Most image processing applications perform three kinds of operations: point operations such as GST (Gray Scale Transformation), window operations such as edge detectors, and some complicated perpendicular transforms such as DFT (Discrete Fourier Transform) [3]. In all these operations, the data flow dependencies require data to be stored in memory elements. In FPGA, these memory elements are logical memories implemented by allocating them to embedded memories to eliminate the overheads associated with data fetches to external memories. Typically the design bottleneck will be the memory resources and memory data transfers from/to off-chip memories. Ineffective use of the memory hierarchy requires extra transfers of data and program and can significantly increase both execution time and resource consumption. The standard method of allocating row buffers to memory involves declaring arrays which are implemented by directly instantiating one block RAM or several per array. This approach also means one port is used for writing and the other for read. This leads to inefficient use of memories. Memory hierarchy must be taken into consideration to minimize the overall system cost. Hence, there is a need for efficient implementation of memory elements and efficient allocation to available FPGA memory types (Block RAM, distributed RAM) in designs. One major advantage of efficient resource usage is reduced power consumption. Existing works on memory optimization for real-time video processing systems are only focused on window-based memory architecture [3] [4] [7] [6]. Several studies were carried on memory mapping which consists of selecting memory components from a library and/or selecting where the memory components are placed and the way in which they are connected to the hardware logic. However, only Block RAMs are considered as available memory [1] [7] [6] [14]. Work in[8] presents a mapping algorithm using Block RAM and Distributed RAM, however only window-based memory is considered; the memory access dependencies on different sources is not considered along with port mapping. Moreover, none of these works propose a model to computing the memory requirement of a processing chain. *To address these issues, we present an interface model for modeling image processing chains and generating memory elements. Based on that model, we evaluate the memory requirement and present an approach that allocates efficiently it to a given FPGA architecture.* We propose an architecture for memory elements including windows, buffers and shift registers. By taking information about the FPGA architecture our optimization model allocates memory elements to the available FPGA on-chip memories. The on-chip memories organized as Block RAMs and distributed RAMs. Distributed RAMs are built from the logic resources and are ideal for small memories. The Block RAMs are more suited to larger on-chip memory storage requirements. *The main contributions of this paper are to present an interface model to evaluate memory requirements for image processing applications and to extend previous works for efficient usage of FPGA memory resources. The allocation considers both Block RAMs and Distributed RAM.* With real-life video processing applications, we demonstrate the importance of our approach.

The rest of this paper is organized as follows: Section 2 present related work on memory optimization for real-time video processing systems. In Section 3 the proposed interface and optimization model are presented. Section 4 discusses experimental results of the approach and Section 5 concludes the paper.

V.2 Related work

This section discusses related work on memory hierarchy and memory optimization for image processing applications.

Regarding memory architecture for image processing applications, Work [3] presents a parameterized memory structure for window operations with emphasis on data reuse but do not address the memory allocation optimization. Lawal et al. also present a memory structure for window operations and optimization model targeting only block RAM [6] and both block RAM and distributed RAM [7]. The proposed memory architecture targets only window applications and the optimization model focus only on reducing the number of block RAM used. In contrary, we present an architecture for most image processing applications based on window, buffer and shift registers.

Work [11] considers on-chip memories for memory mapping. In [5], the same technique is improved for dual-ported on-chip memories. In both cases, the framework considers only one type of physical bank and does not handle both single and dual ported at the same time. Work [8] presents a model to optimize the mapping a memory requirement to a set of embedded memories (Block RAM and Distributed RAM). Our model derives its roots from this work. The main difference is that we consider the memory access dependencies on different sources along with different types of sources and port mapping.

We address the limitations of the previous works with an interface model to evaluate memory requirements in image processing applications and to extend previous works for efficient usage of FPGA memory resources during the allocation a given memory requirement to FPGA embedded memories.

V.3 Architectural model

V.3.1 Image processing operations

An image is characterized by its resolution or size and by the width of a pixel. For VGA images, the resolution is 640 x 480. For RGB color images, the width of a pixel is generally 24 bits, 8 bits for each color. In this paper, we denote w x h, the image resolution and p the width of a pixel. Image processing operations can be classified as:

Low-Level Operations Low-level operations transform image data to image data. This means that such operators deal directly with image matrix data at the pixel level. These operations can be classified into point, neighborhood and global operations. Point operations are the simplest of the low-level operations since a given input pixel is transformed into an output pixel. Such operations include arithmetic operations, logical operations, table lookups, and threshold operations. Local neighborhood transformation from an input pixel to an output pixel depends on a neighborhood of the input pixel. Such operations include two-dimensional spatial convolution and filtering, smoothing, sharpening, image enhancement. Finally, global operations build upon neighborhood operations in which a single output pixel depends on every pixel in the input image. A prominent example of such an operation is the discrete Fourier transform which depends on the entire image. These operations are quite data intensive as well.

Intermediate-Level Operations Intermediate-level operations transform image data to a slightly more abstract form of information by extracting certain attributes or features of interest from an image. The transformations involved lead to a reduction in the amount of data from input to output. Intermediate operations primarily include segmenting an image into regions/objects of interest, extracting edges, lines, contours, or other image attributes of interest such as statistical features.

High-Level Operations High-level operations interpret the abstract data from the intermediate-level, performing high level knowledge-based scene analysis on a reduced amount of data. Such operations include classification/recognition of objects or a control decision based on some extracted features. They are less data intensive and more inherently sequential rather than parallel.

In all these operations, the data flow dependencies require data to be stored in memory elements.

V.3.2 Image processing memory elements

These memory structures allow the user to have the temporal and spatial data contexts required by an application to work on the current pixel. They include:

Shift Register Shift registers are one of the simplest and most commonly used memory structures in DSP processing. The idea behind a shift register is to provide a one-dimensional temporary data buffer to store the incoming samples from a streaming interface. The duration for which a sample is stored in the shift register is determined by the algorithm being implemented.

Buffers They are used for synchronization between to processing elements working at a different speed or with a different data width (Packing and Unpacking data). They can be implemented using FIFO (different speed) or using a FSM (different width and same speed). A line buffer is a multi-dimensional shift register capable of storing several lines of pixel data. Typically, line buffers are implemented as block RAMs to avoid the communication latency to off-chip DRAM memories. Also, a line buffer requires simultaneous read and write access, which takes full advantage of the dual-port nature of block RAMs. Although a memory window is a subset of a line buffer, a line buffer cannot be used directly in most video and image processing algorithms. A frame buffer stores a complete frame.

Memory Windows In video and image processing, a memory window is defined as a neighborhood of N pixels centered on pixel P. The memory window can also be viewed as a collection of shift registers, which forms a 2-dimensional data storage element. This kind of memory is usually implemented as flip-flops for these reasons: It has fewer data elements. Only the pixels required to compute some characteristic of P are stored. An example of this is the 3 x 3 memory windows used in edge detection. All pixels in the neighborhood must be simultaneously available when computing the value of P.

V.3.3 Image processing applications

We model image processing applications as set of components and sources connected together. A component is a processing element composed of a core that represents the task's functionality and an interface that establishes data transmission on the input/output ports. A source is a memory element that store and forward a pixel, a group of pixel or a frame using the same interface model as the components. Figure V.1 show an example of processing application with four components A, B, C and D and five sources S1 to S5.

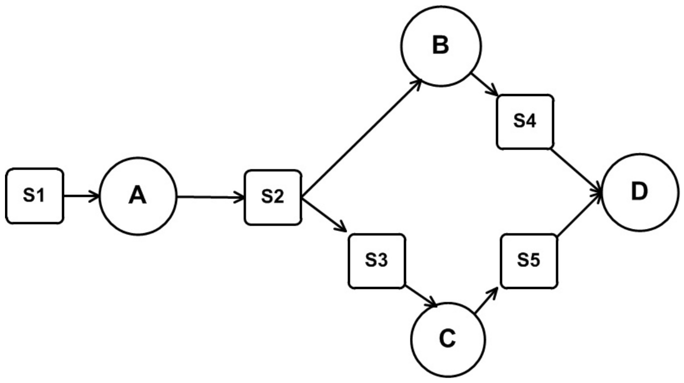


Figure V.1: Model of an image processing application. A, B, C and D are components; S1 to S5 are memory sources.

Interface and component model An interface is a shared boundary across which two separate components of system exchange information. We present the Component Interconnect and Data Access (CIDA) interface, that fulfills the requirement of the interface model presented in [2] and provides a set of common features that IP cores can use. The interface formalism models the interaction that a system component carries out with its environment. This interaction is performed by means of input and output actions. Input actions describe the behavior that the component expects (or assumes) from the environment. Output actions represent the behavior it communicates (or guarantees) to the environment. Formally, an Interface Automaton is a tuple $S =< Q, q^0, Q^f, A^I, A^O, A^H, \rightarrow>$ where Q is a finite set of states with $q^0 \in Q$ being the initial state and $Q^f \subseteq Q$ being the set of final states; $A^I, A^O and A^H$ are pair wise disjoint finite sets of input, output, and hidden actions, respectively, A is the set of all actions i.e. $A = A^I \cup A^O \cup A^H$; $\rightarrow \subseteq Q \times A \times Q$ is the transition relation that is required to be input deterministic (i.e. $(q, a, q_1), (q, a, q_2) \in \rightarrow$ implies $q_1 = q_2$ for all $a \in A^I$ and $q, q_1, q_2 \in Q$). CIDA is used to model interaction between system components and their environment. This interaction is performed by means of input and output actions. Input actions describe the behavior that the component expects (or assumes) from the environment. Output actions represent the behavior it communicates (or guarantees) to the environment. CIDA features include a streaming interface for exchanging data among hardware and DMA for exchanging data with global memories. CIDA can be parameterized to efficiently accommodate different peripherals. To manage local, direct and remote data access, we differentiate between the two main properties of CIDA: CIDA Streaming for streaming-based design and CIDA-DMA for streaming design that incorporate global memory access. CIDA-Streaming is based on a handshaking protocol with the different configurations and the inputs, outputs and internal signals used to wrap a user logic function as shown in Figure V.2. The data signals are made generic to handle different data width during the streaming. CIDA DMA is made of a streaming

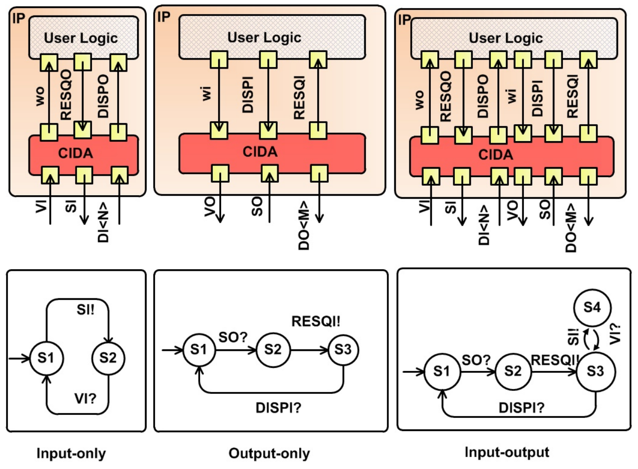


Figure V.2: CIDA streaming. On top, interface description; on bottom, interface automaton.

interface and a memory-mapped interface. CIDA DMA is used to write incoming streams into a memory frame buffer without processor intervention. Reciprocally, images can be read from the frame buffer and streamed out. A control logic containing memory-mapped registers, is used to reset or start/stop execution of DMA operations from the software. It also contains a set of registers for configuring and gathering the status of IPs. This relieves the designer from implementing memory-mapped registers and bus interface in each IP which needs a configuration from the software. Using the previous interface model, we define a component C as a tuple (U, I) where

- U is the core function of C.
- *I* is an interface through which C interacts with other components, for instance, a messaging interface or a procedural interface.

Components are executable units that read data and write data to ports. They are composed of a core that represents the task's functionality and an interface that establishes data transmission on the input/output ports. The component model provides a framework for representing components and composing them. A component is characterized by his input data width I_c and his output data width O_c. I_c and O_c are multiple of p i.e. a component processes a pixel or a set of pixels and output a pixel or a set of pixels.

Memory Source model this paper will focus on the following memory elements: shift registers, buffers and windows.
Shift Registers *:* are characterized by a width w_i (size of a word) and a length l_i (number of words). A shift register can have a fixed or a variable length; we will consider fixed length to able to evaluate the memory requirements. They can be captured with the CIDA interface and implemented with simple dual-port RAM (1 read port, 1 write port) using either Distributed RAM or a Block RAM in FPGA.
Buffers*:* are characterized by a width w_i (size of a word at the input), a width w_o (size of a word at the output) and a depth d (number of words). Buffers can be can be captured with the CIDA interface and can be customized to utilize block RAM, distributed RAM or built-in FIFO resources available in some FPGA families to create high-performance, area-optimized FPGA designs.
Windows*:* A N x M memory window uses N-1 line buffers. Line buffers can be captured with the CIDA interface and implemented using Simple Dual-port RAM and True dual-port RAM. The Simple Dual-port RAM provides two ports, A and B, write access to the memory is allowed via port A, and read access is allowed via port B.

V.3.4 Embedded Memories

In FPGA, memory elements can be implemented using Block RAM and Distributed RAM. RAM can be implemented as shift registers, FIFO, single-port RAM, single dual port RAM and true dual port RAM. Their availability depends on the target FPGA. For this work, we will use Zynq XC7z02- 0CLG484-1 as FPGA platform. The Zynq XC7z020CLG484-1 contains 140 Dual-port 36 Kb block RAM with port widths of up to 72 bits wide. Each 36 Kb Block RAM can be configured as two 18 kb Block RAMs. Each block RAM has two completely independent ports that share nothing but the stored data. Each port can be configured as 32K x 1, 16K x 2, 8K x 4, 4K x9 (or 8), 2K x 18 (or 16), 1K x 36 (or 32), or 512 x 72 (or 64). The two ports can have different aspect ratios without any constraints. Each block RAM can be divided into two completely independent 18 Kb block RAMs that can each be configured to any aspect ratio from 16K x 1 to 512 x 36. Block RAM can be used as Programmable FIFO logic, single-port RAM, single dual-port RAM and true-dual port RAM. Distributed RAMs are built from the logic resources (CLBs) and are ideal for register files closely integrated with logic. CLBs are organized into four slices where each slice consists of two Look-Up Tables (LUTs), for a total of Eight LUTs per CLB for random logic implementation or distributed memory. Memory LUTs are configurable as 64x1 or 32x2 bit RAM or shift register (SRL) [13]. The Zynq XC7z020CLG484-1 contains 13300 slices and only Between 25-50% of all slices can also use their LUTs as distributed 64-bit RAM or as 32-bit shift registers (SRL32) or as two SRL16s [13]. By cascading the memory of each LUT larger and wider memories can be achieved. Distributed RAM can be used as FIFO, single-port RAM, single dual-port RAM and true-dual port RAM. Table V.1 gives the total available embedded memories for Zynq XC7z020CLG484-1.

Memory	RAM size	Number	Total size(Kb)
Block RAM	36 Kb	140	5040
Distributed RAM	64 b	6650	415

Table V.1: Total available emebedded memories for Zynq XC7z020CLG484-1.

V.3.5 Memory synthesis problem

Memory synthesis is the process of mapping various logical data structures used in the design to some appropriate physical instances. As described in section V.3, source memories are realized by combining one or more embedded memories together. The objective of Memory synthesis is to find the set of combinations that yields the minimum cost subject to achieving the desired image processing. If the memory object does not completely occupy the Block RAM there will be unused memory area. The memory mapping algorithm used in this paper is based on work [8]. The main difference is that we consider the memory access dependencies on different sources along with different types of sources and port mapping. To find the optimal use of the Block RAM, The following notations and variables are used in the formulation:

- M is the set of all the available Block RAM M_k and K is the number of Block RAMs. $M = \{M_k | k = 1, 2, .., K\}$

- S_{M_k} is the size of the Block RAM M_k and is specified by the FPGA. For Xilinx Zynq, S_{M_k} is 36 kbits. The memory objects allocated to the Block RAM determine the depth D_{M_k} and width W_{M_k} of M_k.

- W is the set of all possible widths W_n for Block RAMs on the FPGA. 1, 2, 4, 8, 9, 16, 18, 32, 36, 64 and 72 are allowed on Xilinx Zynq FPGA.
  $W = \{W_n | n = 1, 2, .., N\}$

- R is the set of all memory objects R_i to be allocated and I is the number of memory objects. $R = \{R_i | i = 1, 2, .., I\}$. The size S_{R_i} of memory object R_i is defined as the product of the depth D_{R_i} and the data width W_{R_i} of the memory object R_i.
  $S_{R_i} = D_{R_i} * W_{R_i}$. P_i defines the number of ports of R_i; P_i=1 for single-port and simple dual-port; P_i=2 for true dual-port.

- $C_{i,j}$ defines the relation between the memory accesses of R_i and R_j. A value of 0 or 1 on $C_{i,j}$ means no simultaneous memory access or simultaneous memory access respectively.

- If W_{R_i} is not a member of W, R_i is partitioned into r_j partitions such that the width, w_r, of each partition is a member of W where $j = 1, 2, , J$ and J is the number of partitions in object R_i.

- Memory object R_i may be allocated to as many Block RAMs as required.
  $\sum_{k=1}^{K} D_{i,k} W_{R_i} \leq S_{R_i}$ where $D_{i,k}$ is the part of depth D_{R_i} allocated a M_k.

- For all R_i in R and a M_k in M, the sum of the allocations may not be more than the size of the Block RAM. $\sum_{i=1}^{I} D_{i,k} W_{R_i} \leq S_{M_k}$

- Our goal is to use a minimum number of Block RAMs by allocating memory requirements below a certain threshold to CLBs. The threshold is chosen such that power consumption by the CLBs implementing the memory does not exceed the power consumed by Block RAM. The power was used as a measure to compare Distributed RAM and Block RAM because it can be estimated from vendors tools whereas the area, in term of gate counts, cannot. We allocated R_i to Block RAM when the power consumption by CLB allocation is not larger than that by Block RAM. To evaluate the threshold, we estimated the power consumption for different widths and depths of a single-port RAM and a dual-port RAM. We used Xilinx Coregen to generate the memories and the power was estimated using Xilinx Power Estimator (XPE). Tables V.3 and V.4 shows the results of the evaluation. We observed that when the memory size is less than or equal to 8192 bits, power consumption by CLB allocation is not larger than that by block RAM. CLBs are also used as logic to implement IP cores and therefore their usage as distributed RAM should be limited to allow enough resources for other cores. By computing the slices estimation for RAM with different widths and depths, we can see that implementing 8192 bits of distributed RAM required up to 82 slices which is an acceptable number.

Table V.2: Power estimation for RAM with different widths and depths

Width x depth	Block RAM (mW)	Dist. RAM (mW)
8 x 32	10,09	0,64
8 x 64	10,16	0,76
8 x 128	10,18	1,37
8 x 256	10,25	2,26
8 x 512	10,27	4,12
8 x 1024	9,92	7,43
8 x 2048	9,5	16,13
8 x 4096	18,85	31,44
8 x 8192	36,77	60,67
16 x 64	10,54	1,44
32 x 64	14	5,77
16 x 128	10,6	2,57
32 x 128	14,19	8,31
16 x 256	10,65	4,16
32 x 256	14,31	12,34
16 x 512	10,7	7,98
32 x 512	13,42	19,55
16 x 1024	10,38	15,06
32 x 1024	22,79	30,38

Table V.3: Single-port RAM

Width x depth	Block RAM (mW)	Dist. RAM (mW)
8 x 32	11,35	0,92
8 x 64	11,41	1,09
8 x 128	11,45	1,81
8 x 256	11,54	3,35
8 x 512	11,56	5,43
8 x 1024	11,61	10,48
8 x 2048	10,87	21,07
8 x 4096	20,92	40,56
8 x 8192	39,98	79,7
16 x 64	14,88	2,17
32 x 64	27,89	6,7
16 x 128	15,1	3,56
32 x 128	27,9	9,72
16 x 256	15,15	6,43
32 x 256	28,12	14,78
16 x 512	15,33	11,16
32 x 512	28,3	26,43
16 x 1024	15,56	22,19
32 x 1024	28,48	46,59

Table V.4: Dual-port RAM

- The unused memory space in M_k is defined as U_{Mk}. $U_{Mk} = S_{Mk} - \sum_{i=1}^{I} D_{i,k} W_{Ri}$

- The objective function of the algorithm is to minimize the sum of all U_{Mk}. Minimize $\sum U_{Mk}$ Subject to the previous constraints. This will also minimize the number of Block RAMs used.

The above Integer Linear Programming (ILP) formulation does not capture the complete cost of the resultant design as it does not account for the cost of the address decoding logic which it is not the focus of this paper. Since, Block RAMs are of a fixed size and fixed number of ports, we present an efficient linear time algorithm to perform the mapping. The proposed allocation is presented in algorithm 2. In step 1, the algorithm ensures that memory objects are conform to the allowable port width in the considered FPGA. In step 2, the algorithm creates global memory objects by grouping memory objects with same width, same number of ports and no simultaneous memory access. Steps 3-16 ensure that the algorithm iterates through all the memory objects starting with the first. In step 5, the GMO is allocated to Distributed RAM when its size is such that the power for the allocation in Distributed RAM is less than the one in Block RAM. In step 8, the GMO are allocated to a complete Block RAM when the number of ports of the GMO is 2 which is the number of the Block RAM. In step 10, the algorithm allocates a GMO to one port of a Block RAM while optimal use of unallocated memory space in the Block RAM through the second port is implemented in step 11.

V.4 Experimental results

Table V.5 shows the details of the various benchmarks used to evaluate the performance of the proposed solution.

The hand follower project identifies skin like colors (like hands) in the image and uses mean shift to follow it while moving around. The steps within the processing chain for this task are: grabbing images from the camera, convert Bayer to RGB, convert RGB to HSV and filter the skin like colors, perform mean shift algorithm to follow the skin colors (initial window in the middle of the screen), add the current window as a red rectangle to the video stream, the results are put into memory and sent to a display.

The segmentation, shown in figure V.3, does the foreground/ background segmentation. It stores an initial image as reference background and defines a pixel as foreground if it has a significant different color value compared to its corresponding background pixel. Two CIDA DMA are used. One CIDA DMA reads the calculated foreground and stores it to the memory where it is read for the display. The second CIDA DMA handles reading the old background frame from the memory and storing the new background in the memory. The central IPCore is the segmentation core which reads two image streams (old background frame, current image) and gives back also two image streams (new background frame, foreground). The input image is converted from Bayer pattern to RGB after streamed from the camera and before entering the segmentation core.

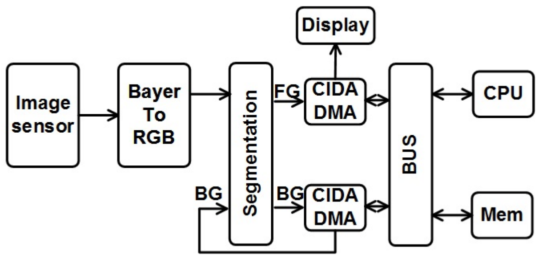


Figure V.3: Segmentation implementation.

Harris corner test case, shown in figure V.4, calculates the Harris corner points, a well known interest point descriptor. The incoming image is converted into a RGB image. Then the discrete approximation of the horizontal and vertical derivative is calculated, using 3x3 Sobel operators. Harris points are then found as points where both the horizontal and vertical derivative are significantly large. The results are put into memory and sent to a display.

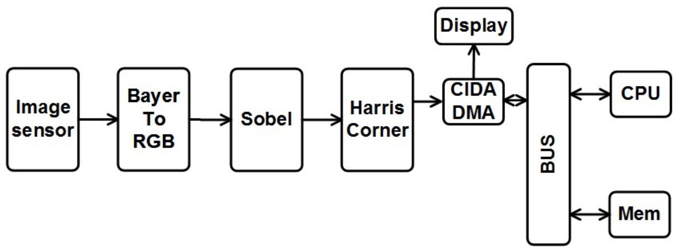


Figure V.4: Harris corner implementation.

The JPEG encoder is an implementation of the JPEG compression available on [9].

Table V.5 compares the result of the default mapping in Xilinx tools with the result of the proposed mapping algorithm. The results show a reduction in the number of used block-RAMs most of the time and even more reduction when distributed RAM is considered. This is because small memory requirements were implemented using CLBs rather than block-RAMs. There was no improvement for the Harris corner test case because most of the memory objects were bigger than the size of a Block RAM.

Design	default mapping	our mapping w/o dist. RAM		our mapping w dist. RAM		
	BRAM	BRAM	CMP	BRAM	Dist. RAM	CMP
Harris Corner	35	35	0%	33	4	5.71%
Hand Follower	41	39	4.87%	37	5	9.75%
Segmentation	37	35	5.40%	32	5	13.51%
JPEG Encoder	11	7	36.36%	6	1	45.45%

Table V.5: Experimental evaluation.

V.5 Conclusion

In this paper, we presented a streaming interface, called Component Interconnect and Data Access (CIDA), based on interface automata formalism, for modeling image processing chains and memory elements. An optimization model for the allocation of the memory elements to embedded memories in FPGA is also presented. The proposed approach considers both block RAM and distributed RAMs as possible candidates for implementing logical memory on an FPGA during memory allocation. CIDA has successfully been used to design several video applications such as Harris corner, segmentation, hand follower and more. Our experiments show the efficacy of the proposed allocation algorithm.

Algorithm 2 Memory allocation algorithm

```
Input: R[$R_1, ..., R_I$ ]: set of I memory objects.
  M[$M_1, ..., M_K$]: set of K Block RAMs.
  St: memory size up to which the power for the allocation in Distributed RAM is less
  than the power for the allocation in Block RAM.
Output: MA[$MA_1, ..., MA_K$]: set of K Allocated Block RAMs
  Configure memory objects (R) such that $W_{R_i} \in W$
  Create global memory objects (GMO) by grouping memory objects with same Width,
  same number of ports and $C_{i,j} = 0$.
  Starting with the first GMO and the first Block RAM
  if size (GMO) < St then
    Allocate GMO to Distributed RAM
  else
    if port (GMO)==2 then
      Allocate GMO to Block RAM
    else
      Allocate GMO to Block RAM via port A
      If Block RAM is not fully used find maximum use of remaining memory via port B
      using another GMO satisfying the constraints port (GMO)==1 and size(GMO) <
      St
    end if
  end if
  Select the next GMO when the current has been fully allocated
  Select the next Block RAM when all the memory space has been optimally used
  Return the set of allocated Block RAMs after allocating all GMOs
```

Bibliography

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, 2011.

[2] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[3] Y. Dong and Y. Dou. A parameterized architecture model in high level synthesis for image processing applications. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 523–528. IEEE Computer Society, 2007.

[4] Y. Dong, Y. Dou, and J. Zhou. Optimized generation of memory structure in compiling window operations onto reconfigurable hardware. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 110–121. Springer, 2007.

[5] W. K. Ho and S. J. Wilton. Logical-to-physical memory mapping for fpgas with dual-port embedded arrays. In *Field Programmable Logic and Applications*, pages 111–123. Springer, 1999.

[6] N. Lawal, M. O'Nils, and B. Thornberg. C++ based system synthesis of real-time video processing systems targeting fpga implementation. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–7. IEEE, 2007.

[7] N. Lawal, B. Thornberg, and M. O'Nils. Architecture driven memory allocation for fpga based real-time video processing systems. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pages 143–148. IEEE, 2011.

[8] L. Najeem. *Memory Synthesis for FPGA Implementation of Real-Time Video Processing Systems*. PhD thesis, 2008.

[9] OPencores. Jpeg encoder :: Overview, 2009.

[10] J. Vasiljevic and P. Chow. Using buffer-to-bram mapping approaches to trade-off throughput vs. memory use. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[11] S. J. Wilton. *Architectures and algorithms for field-programmable gate arrays with embedded memory*. PhD thesis, Citeseer, 1997.

[12] Xilinx. Xilinx power estimator. *Xilinx Power Estimator available on: http://www.xilinx.com/products/design_tools/logic_design/xpe.htm.*, 2014.

[13] Xilinx. Zynq-7000 all programmable soc overview. *Zynq-7000 all programmable soc overview, advance product specification-ds190(v1. 7) available on: http://www. xilinx.com/support/documentation-/data sheets/-ds190-Zynq-7000-Overview. pdf*, 2014.

[14] H. Zhou, Z. Lin, and W. Cao. High-level technology mapping for memories. *Computing and informatics*, 22(5):427–438, 2012.

APPENDIX

I, certify that Michael Mefenza Nentedem is the first author of the article *"Interface Based Memory Synthesis Of Image Processing Applications In FPGA"* and have completed at least 51% of the work in the article.

Name:____________________ Signature: ____________________ Date: ____________________

VI Conclusions

VI.1 Summary

The complexity of video-based embedded systems is currently high and is expected to rise even further in the future as consumers and applications demand more functionality and performance. There are several challenges to embedded video system development in FPGAs. Some of these challenges include the design reuse, design flow, resources usage and design verification.

In this dissertation, we presented a novel interface, the Component Interconnect and Data Access (CIDA), and its implementation, based on interface automata formalism. CIDA can be used to capture system-on-chip architecture, with primarily focus on video-processing applications, which are mostly based on data streaming paradigm, with occasional direct memory accesses. We introduced the notion of component-interface clustering for resource reduction and provided a method to automatize this process. With real-life video processing applications implemented in FPGA, we showed that our approach can reduce the resource usage (#slices) by an average of 20% and reduce power consumption by 5% compared to implementation based on vendor interfaces. We used that interface for modeling image processing applications and generating common memory elements. Based on the proposed model and information about the FPGA architecture, we also presented an optimization model to achieve allocation memory requirements to embedded memories (Block RAM and Distributed RAM). Allocation results from realistic video systems on Xilinx Zynq FPGAs verified the correctness of the model and showed that the proposed approach achieves appreciable reduction in block RAM usage.

A framework for fast prototyping of embedded video applications using the proposed interface was also presented. Starting with a high-level executable specification written in OpenCV, we apply semi-automatic refinements of the specification at various levels (TLM and RTL), the lowest of which is a system-on-chip prototype in FPGA. The refinement leverages the structure of image processing applications to map high-level representations to lower level implementation with limited user intervention. Our framework integrates the computer vision library OpenCV for software, SystemC/TLM for high-level hardware representation, UVM and QEMU-OS for virtual prototyping and verification into a single and uniform design and verification flow. With applications in the field of driving assistance and object recognition, we proved the usability of our framework in producing performance and correct design. The verification is done at IP level using an automatic Universal Verification Methodology (UVM) environment that enables assertions-based, coverage driven and functional verification of SystemC models based on CIDA. Our experimentation showed that the proposed environment can rapidly be integrated to a SystemC design while improving its coverage and assertion-based verification.

VI.2 Future Work

Future works should focus on investigating the use of design space exploration in the proposed SystemC/OpenCV environment for optimal partitioning into hardware/software from a specified design. Another extension could be the integration of the proposed verification environment into a complete hardware/software system on chip modeled with SystemC. Such integration will allow an investigation of UVM capabilities in verification of hardware and software components.