Accelerating embedded image processing for real time: a case study

Article *in* Journal of Real-Time Image Processing · May 2013

DOI: 10.1007/s11554-013-0353-2

CITATIONS
12

READS
448

4 authors, including:

Sol Pedre
Comisión Nacional de Energía Atómica
17 PUBLICATIONS **156** CITATIONS
SEE PROFILE

Tomáš Krajník
Czech Technical University in Prague
93 PUBLICATIONS **1,563** CITATIONS
SEE PROFILE

Patricia Borensztejn
University of Buenos Aires
13 PUBLICATIONS **49** CITATIONS
SEE PROFILE

Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Sol Pedre · Tomáš Krajník · Elías Todorovich · Patricia Borensztejn

Accelerating embedded image processing for real time: a case study

Received: date / Revised: date

Abstract Many image processing applications need real-time performance, while having restrictions of size, weight and power consumption. Common solutions, including hardware/software co-designs, are based on Field Programmble Gate Arrays (FPGAs). Their main drawback is long development time. In this work, a co-design methodology for processor-centric embedded systems with hardware acceleration using FPGAs is proposed. The goal of this methodology is to achieve real-time embedded solutions, using hardware acceleration, but achieving development time similar to that of software projects. Well established methodologies, techniques and languages from the software domain- such as Object-Oriented Paradigm design, Unified Modelling Language, and multithreading programming- are applied; and semiautomatic C-to-HDL translation tools and methods are used and compared. The methodology is applied to achieve an embedded implementation of a global vision algorithm for the localization of multiple robots in an e-learning robotic laboratory. The algorithm is specifically developed to work reliably 24/7 and to detect the robot's positions and headings even in the presence of partial occlusions and varying lighting conditions expectable in a normal classroom. The co-designed implementation of this algorithm processes 1600×1200 pixel images at a rate of 32 fps with an estimated energy consumption of 17μJ per frame. It achieves a $16\times$ acceleration and 92% energy saving, which compares favorably with the most optimized embedded software solutions. This case study shows the usefulness of the proposed methodology for embedded real-time image processing applications.

Keywords real-time image processing, hardware/software co-design methodology, FPGA, robotics, hardware acceleration

S. Pedre and P. Borensztejn
Departamento de Computación, FCEN-UBA, Argentina
Tel.: +54-11-45763390
E-mail: {spedre, patricia}@dc.uba.ar

T. Krajník
Czech Technical University in Prague, Czech Republic
E-mail: tkrajnik@labe.felk.cvut.cz

E. Todorovich
Departamento de Computación y Sistemas, FCE-UNICEN, Argentina
E-mail: etodorov@exa.unicen.edu.ar

1 Introduction

Many image processing applications require real-time solutions. A usual approach to achieve this performance goal is to exploit their inherent parallelism, by building implementations on parallel architectures, such as GPUs (Graphical Processing Units) or FPGAs (Field Programmable Gate Arrays). The appearance of the CUDA (Compute Unified Device Architecture) framework [25] has made it possible to implement image processing methods on GPUs with relatively small coding effort, resulting in their significant speedup. Although these implementations are based on affordable computational hardware found in virtually all modern PC systems, they are unsuitable for applications that require not only real-time performance but also small and low power consumption solutions. Such is the case of a wide range of embedded systems, from remote sensing applications and field robotics to mobile phones and consumer electronics.

In these cases, alternative solutions include using FPGAs or designing ASICs (Application Specific Integrated Circuits). These approaches not only are suitable for parallel solutions, but also allow tailored hardware acceleration, e.g, with particular memory access patterns or bit tailored multipliers/adders. ASICs provide the best solution in terms of performance, unit cost and power consumption. FPGAs are designed to be configured by a designer after manufacturing- hence "field-programmable". The ability to update the functionality after shipping, the partial re-configuration of a portion of the design, and the low non-recurring engineering costs relative to an ASIC design (notwithstanding the generally higher unit cost), offer advantages for many applications. Ac-

cording to the 2012 Embedded Market Survey conducted by Embedded magazine [4], 35% of 1,700 surveyed embedded engineers are currently using FPGAs in their designs. The highly parallel architecture of FPGAs with low power consumption, small size and weight. as compared with PC or GPU-based systems, provide an excellent platform for achieving real-time performance on embedded image processing applications.

The inclusion of processor cores embedded in programmable logic has also made FPGAs an excellent platform for hardware/software co-designed solutions. These solutions try to combine the best of both software and hardware worlds, making use of the ease of programming a processor while designing tailored hardware accelerator modules for the most time-consuming sections of the application. This not only accelerates the resulting system, as compared with the embedded processor solution, but also allows savings in energy. During 2011, the two major FPGA vendors (Xilinx and Altera) announced new chip families that combine powerful ARM processor cores with low-power programmable logic [1, 47]. While FPGA vendors have previously produced devices with onboard processors, the new families are unique in that the ARM processor system, rather than the programmable logic, is the center of the chip [38]. This strengthens the growing trend towards co-designed processor-centric solutions in FPGA-based chips. According to the 2012 Embedded Market Survey, 37% of the engineers that do not use FPGAs in their current designs confirmed that this trend would make them reconsider the matter [4]. Furthermore, several authors reported successful co-designed solutions in FPGA-based hardware for image processing and vision-based system, such as object tracking [14], background subtraction [37] and mobile robotic navigation [29].

The main drawback of FPGA-based methods is time-consuming development [1]. The rising complexity of these applications makes it difficult for designers to model the functional intent of the system in languages that are used for implementation, such as C or HDLs (Hardware Description Languages). Moreover, engineers regard the difficulty in programming FPGAs in HDL as an important reason for not using FPGAs [4]. This creates a great need for methodologies, languages and tools that reduce development time and complexity by raising the abstraction level of design and implementation. Advances have been made in high-level modeling using specific Unified Modeling Language (UML) profiles to simplify design. Besides, much work is being done in high-level synthesis tools, which translate constructs in C/C++ to HDL to simplify hardware implementation. However, there is still a great need for research in co-design methodologies, languages and tools, so that the recent combination of powerful processors with programmable logic can reach its full potential.

The main contributions of this article are:

- The proposal of a co-design methodology for the growing field of processor-centric embedded systems with hardware acceleration in FPGA-based chips. The goal is to achieve real-time embedded solutions, using hardware acceleration, but achieving development time similar to that of software projects. To reduce the development time, well established methodologies, techniques and languages from the software domain are applied, such as Object-Oriented Paradigm design, Unified Modelling Language and multithreaded programming. Moreover, to reduce hardware coding effort, semiautomatic C-to-HDL translation tools and methods are used and compared.
- The proposal of a simple and robust algorithm for multiple robot localization in global vision systems. This algorithm integrates an e-learning robotic laboratory for distance education that allows students from all over the world to perform experiments with real robots in an enclosed arena. Hence, the algorithm was specifically developed to work reliably 24/7 and to detect the robot's positions and headings even in the presence of partial occlusions and varying lighting conditions expectable in a normal classroom.
- The co-designed implementation of this algorithm following the proposed methodology. This solution presents -to the best of the authors' knowledge- the first implementation of such an algorithm in FPGA-based hardware. It also shows the usefulness of the proposed methodology for embedded real-time image processing applications.

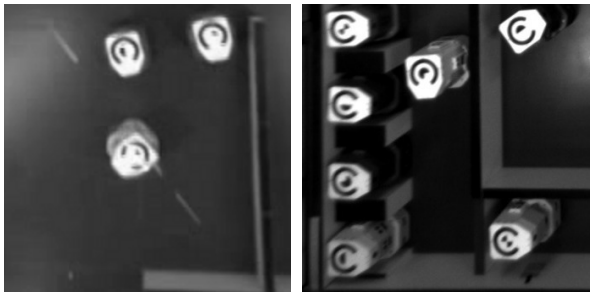
This article extends the works in [31, 30]. New aspects are: (1) the introduction of explicit parallel design from the initial stages and its corresponding multithreaded implementation both in the original C++ code and in the final embedded processor, using a specialized embedded operating system; (2) the comparison between two C-to-HDL semiautomatic synthesis tools and a guided C-to-HDL translation method; and (3) new experimental results, including time, acceleration, area and power consumption metrics.

This paper is organized as follows. Section 2 presents related work on the methodology proposal and global vision localization for multiple robots. Section 3 provides details of the proposed methodology. Section 4 describes the developed algorithm for multiple robot localization, while section 5 describes how the methodology was applied to this application to obtain an accelerated embedded solution. Section 6 presents and discusses acceleration, area and energy consumption results; finally Section 7 provides conclusions.

[1] According to the Embedded Market Survey, an average embedded system project takes 12 months and a group of 13.4 engineers (i.e, 13.4 man-years) [4]. An example of a project prototyped in FPGA is the memory controller of the ERC32 processor of the European Space Agency, which took 10 man-years and 25000 code lines [10].

2 Related Work

This section discusses related work on the image processing algorithm, FPGA-based image processing solutions and the proposed methodology. Multiple robot localization in global vision systems is discussed in Section 2.1; and FPGA co-designed implementations of image processing applications related to robot localization are dealt with in Section 2.2. Key parts of the proposed co-design methodology -high level modeling and high-level synthesis- are discussed in Section 2.3.



(a) Light stripes (b) Uneven illumination

Fig. 1: Lighting conditions on the SyRoTek Arena.

2.1 Vision-based multiple robot localization

The most popular localization approaches in mobile robotics are based on GPS. However, its precision is often insufficient and its signal is not available indoors. One way to tackle the problem of GPS unavailability is to calculate a mobile robot position from its sensory measurements and an environment map [43]. Another solution involves installing localization infrastructure based on radio waves [41], multicamera systems [21] or similar technology. These alternative global positioning systems are more useful in multirobotic systems, which require complete information of the environment for efficient coordination of robot actions. Moreover, they allow the use of small robots unable to carry heavy sensory and computational equipment and are often used to provide "ground truth" data in robotic experiments.

A typical application that requires a small-scale positioning system is the FIRA and RoboCup contests, where teams of small mobile robots compete in a soccer-like game. The robotic soccer rules require the robots to carry identification marks with a specific team color. Thus, the robot dress design usually comprises a team color and a combination of individual colors for distinguishing each robot [7, 17, 12]. Since a robotic soccer round takes five minutes, the playing field illumination is strong, the teams can recalibrate their systems every time a goal is scored, and the robots move very fast, the visual localization approaches of the robotic soccer domain focus on real-time response and precision rather than robustness to variable lighting conditions. For example, [6] proposes to diminish the processing time and increase accuracy by localizing a color patch surrounded by a white border directly in the raw Bayer format image. The localization methods used in robotic soccer have influenced design of localization systems used in path and motion planning for robots in industrial settings [33], in teaching activities [16], and also in experimental setups for multi-agent collaborative tasks, such as platoon formations [18].

Although the setup of the localization system presented in this work is similar to the ones used in robotic soccer, its main requirement is a reliable and continuous (24/ 7) operation in lighting conditions of a normal classroom. It has to deal not only with uneven and variable illumination of the operation area (see Fig. 1) but also with dynamic objects in its field of view. Therefore, the robot dresses might be partially obstructed or visually connected to some other object which would cause color segmentation approaches to fail. In addition, the system does not require such high framerates as in robot soccer, because the robots move ten times more slowly than a typical soccer robot of a MIROSOT league. Hence, the system presented in this work is based on convolution rather than segmentation, because although convolution-based approaches are more computationally demanding, they are also more robust to realistic lighting conditions and offer good position estimation precision. To achieve real-time operation, this convolution based algorithm needs to process the image at rates higher than the camera framerate. In this work, the real-time goal is to be able to process 1600×1200 pixel images at a rate of 30 fps.

2.2 Co-designed FPGA solutions for image processing algorithms related to robotic localization

FPGAs are evolving as complex hardware/software platforms providing powerful embedded microprocessor cores. This enables several acceleration approaches to run image and video algorithms in real-time. Modern devices for image acquisition and visualization can benefit from these acceleration techniques. For this reason, much work can be found in the literature about hardware/software implementations on FPGA for computer vision applications. In particular, some vision algorithms that can be applied to robot localization, such as object tracking [14] or background subtraction [37], have their co-designed FPGA solutions. Furthermore, FPGA is particularly suitable for other robotic applications too [29].

In [14] a video object tracking application is presented. The application is based on a Sequential Monte Carlo method that uses color segmentation and is targeted to CPU/FPGA hybrid systems. Based on a multi-

threaded programming model, the authors have developed a framework that allows design space exploration with respect to the hardware/software partitioning. Additionally, the application can adaptively switch between several partitioning states during run-time by means of partial reconfiguration in order to react to changing input data and performance requirements.

A hardware computing engine to perform background subtraction in video streams is presented in [37]. The embedded system detects people on low-cost FPGAs and is able to segment objects in sequences with resolution 768×576 pixels at 50 fps with an estimated power consumption of 5 W.

In [29] authors propose an optical flow-based algorithm that estimates and compensates ego-motion to allow for object detection from a continuously moving robot. The system is implemented using a traditional HW/SW co-design approach on a Virtex-5 FPGA from Xilinx with an embedded PowerPC Processor. This implementation can process 31 fps at a resolution of 640×480 pixels.

Although the results in these three papers are competitive in flexibility, performance, and power consumption, as compared with state-of-the-art articles, in all of them the designer must design the solution and the hardware coprocessors in a traditional way. In our work, we propose a co-design methodology aimed at reducing development time, and show its applicability to the acceleration of embedded image processing algorithms. Moreover, to the best of our knowledge, this paper presents the first co-designed FPGA-based solution to the problem of multiple robot localization in global vision systems. This solution points to the development of "intelligent cameras": an embedded system including the image acquisition and processing on board so that there is no need to transfer the whole image to other computers.

2.3 High level modeling and high level synthesis

High level modeling and high level synthesis are two related areas that propose to raise the abstraction level of design and implementation in an effort to reduce the long development times associated with increasingly complex designs.

UML is a widely used language for system modeling in the software domain. It is a standard language of the Object Management Group (OMG), which has given rise to OMG standard UML profiles for SoCs, embedded systems and real time systems. The most closely related are UML for SoC, SysML and MARTE. The UML for SoC profile [26] was first introduced in 2009 and mainly defines structure diagrams through specific SoC stereotypes. SysML[27] is a general purpose modeling profile for systems engineering applications. It supports the specification, analysis, design, verification and validation of a broad range of systems. Although it is useful for embedded system design, it also includes support for things not specifically designed for this field such, as personnel or facilities. The UML MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [28] is the most adequate profile, adding capabilities to UML for model-driven development of Real Time and Embedded Systems (RTES). Since its standardization in November 2009, it has become the most recent industry standard in this field.

Based on these OMG standards, several profile extensions have been proposed to enable automatic code generation and/or extend modeling capabilities to more detailed aspects of embedded system design. For example, extensions of the MARTE profile for particular subdomains include proposals for partial run-time FPGA reconfigurability [32] and for dynamic power management [2]. Authors of [39] propose a UML-ESL profile for cache usage analysis. The key to these approaches is not automatic code generation, but the extension of the modeling language to include the particular sub-domains.

To enable automatic code generation, the most common approach involves extending the UML profiles to model SystemC or VHDL structures. In [35] a UML2.0 profile for SystemC was first proposed, and this trend continued in several works like [36], [23] and [34]. In [22], the SysML profile is extended for SystemC constructs, enabling automatic SystemC code generation for simulation, and automatic VHDL code generation for synthesis. In all these approaches, UML designs are just representations of SystemC models, so low-level details such as ports, modules, or SystemC data types need to be modeled in the UML diagrams. In fact, this is what enables the automatic SystemC code generation. In [44], authors propose to use a subset of the MARTE profile and generate rules to translate that subset to VHDL code. A key feature is that they use a subset of C++ as an Action Language to describe the behavior of states in state machines. Then, that C++ code is translated to VHDL using a high level synthesis tool.

In order to perform automatic HDL code generation, the aforementioned approaches restrict the modeling possibilities, for example with "one class-to-one module" synthesis, or the restriction that only complete objects -and not particular methods- can be mapped to hardware. Moreover, many SystemC or VHDL implementation details need to be modeled in the UML diagrams for the automatic translation to work. Our approach is to model the whole system using standard UML2.0 in an Object Oriented manner *prior* to hardware/software partitioning. This is particularly suitable for the processor-centric approach, in which most of the final application will be running in the processor. By modeling the whole application before hardware/software partition, the implementation details related to both hardware and software are abstracted away. This is crucial at this stage: it is what allows engineers to perform a

good, modularized OOP design. This type of design will later on enable the engineer to use profiling tools and find precisely which OOP methods need to be accelerated by hardware. In this way, useless translations to hardware can be prevented. Moreover, by using standard UML2.0, many tools from the software domain that have been developed for years can be used, such as Sparx Enterprise Architect [40]. These tools support not only explicit modeling of parallel algorithms, but also automatic C++ headers and code generation from the UML diagrams. Many tailored UML profiles, although better suited for modeling certain aspects of particular domains, are still behind in tool support.

Complementing the high-level modeling approaches, another important area is High Level Synthesis tools. These include semiautomatic tools to translate constructs using a subset of languages like C/C+ to HDL code. Examples of open-source semiautomatic tools include ROCCC [15], SPARK [13] and DWARV[49]. There are also many proprietary tools such as DIME-C [24], CatapultC [20] or AutoESL [48]. All of them require rewriting effort on the original methods to particular coding styles and C/C++ language subset, and hardware knowledge in order to generate optimized HDL. Since they have specific interface definitions, hand-coded interface modules are required to integrate the automatic-generated modules into the complete system. A comparison of open-source tools can be found in [45], and an excellent study of the proprietary AutoESL tool can be found in [42]. Another approach is to hand-code the modules, but using a clear coding method. In [9] the two-process method is proposed: a structured VHDL design applied on several designs made for the European Space Agency, including the LEON3 processor [8]. In [10], authors show that the method decreased man-years, code lines and bugs in many important projects, and that it is well suited for implementing OOP methods, thereby creating short and readable code. Although the two-process method is not a (semi)automatic tool, the coding guidelines create a schema that could be the starting point for such a tool.

In this work, we compare three different ways to translate the C++ class methods that need to be accelerated by hardware to HDL: open-source tool ROCCC, proprietary tool AutoESL, and the two-process VHDL coding method.

A key feature of the proposed methodology is the union of a good modularized OOP design captured in UML and implemented in C++ that makes it possible to find precisely which methods need to be accelerated by hardware; with semi-automatic tools or guidelines to translate these C++ methods to HDL. This union reduces hardware coding effort, traditionally the most time-consuming and error-prone stage of a hardware-accelerated application.

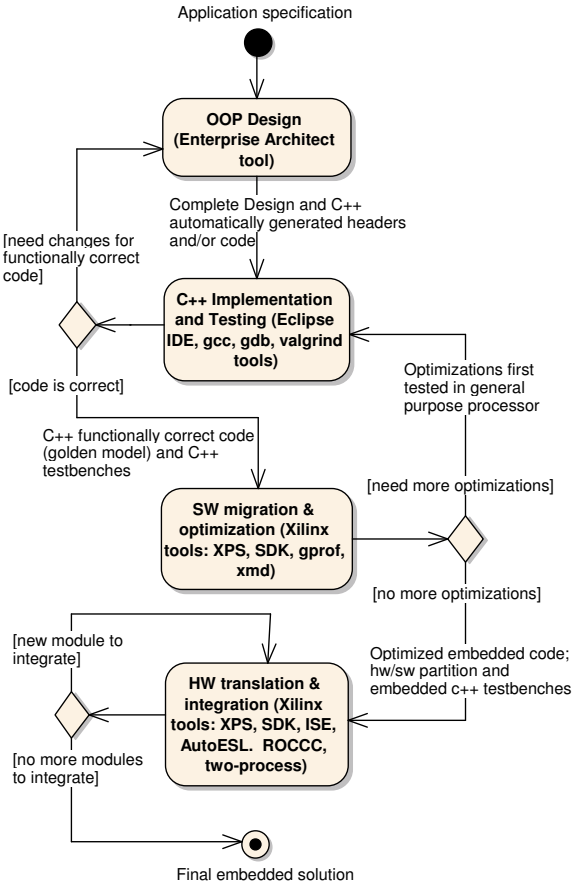


Fig. 2: Methodology overview

3 Methodology

The proposed methodology has four broad stages: A) OOP Design; B) C++ Implementation and Testing in a general purpose processor; C) Software migration, optimization and hardware/software partition; and D) Translation, testing and integration of each hardware module in the final embedded platform (see Fig. 2). In this Section, the steps of each stage are described, and useful tools are presented.

3.1 OOP Design

In this stage, an OOP approach and the use of UML language for modeling is proposed. As already stated, these are widely used in the software community, and many modified UML approaches for hardware specific applications exist. Hence, many software and hardware engineers are familiar with these techniques and at least some of the associated tools. The use of well established

techniques and tools is important to help reduce the design effort by raising the abstraction level, while not imposing the need for engineers to learn new languages, methods and tools.

The OOP design includes concepts such as encapsulation, inheritance, messaging, modularity, polymorphism, and data abstraction. In the proposed approach, abstraction, encapsulation and modularity are particularly important. A modular design -in which the responsibilities of each class are clearly identified, and concise methods are created- is a key part of the methodology. This kind of design assists the engineer in finding the exact methods that need to be accelerated by hardware using profiling tools. This helps to reduce hardware coding effort because no useless translations to hardware are done.

At this stage, the possible parallelizable sections of the algorithms can be identified and modeled. Standard UML offers several ways to model parallel thread-like behavior. One way is to model threads in the structural design as Active classes. Active objects (instances of an active class) model the concurrent behavior of real world objects, and can own an execution thread and initiate control activities. Threads can also be modeled in behavioral diagrams. Activity diagrams include specific fork/join bars to illustrate thread behavior. These diagrams also include swimlines to show which objects perform the activities modeled. Sequence diagrams also include `par` sections to indicate that the interactions in those sections may be executed in parallel or in any order. The most appropriate diagrams to capture the intended design vary in different applications. Several profiles have been proposed to model designs for cluster, parallel and heterogeneous computer architectures, which are beyond the scope of this work.

The complete design is done using Spark's Enterprise Architect [40], a comprehensive UML analysis and design tool. This tool allows for structural design - i.e, classes- and behavioral design -the description of the system behavior by interaction sequences, state machines, activity diagrams, etc. It automatically generates documentation and the class skeleton in several languages (C++, Java, Python, etc.). It can also generate automatic code in these languages, provided there is enough detail in the UML diagrams. The tool also allows reverse engineering, making it possible to reflect in the UML design changes made during implementation.

The output of this stage is the Structural and Behavioral design of the solution, and also the class skeleton with members, methods and all comments in C++ files.

3.2 C++ Implementation and Testing

The second stage is coding and testing the designed solution in a general purpose processor. Although any programming language may be used, the authors prefer C++ as the language of choice in this case study for several reasons. It is a widely used language with full OOP support, allowing to port the UML design perfectly. It also has ample support for running in different embedded processors. Thus, the software solution developed in this stage serves not only as a reference model but also as part of the final software that will run in the embedded processor. Finally, C++, along with C and SystemC, is one of the strong contenders for the best input language for High-level synthesis tools [3]. This means that much work is being done on tools to automatically translate C++ constructs to HDL.

The first step is to code all the methods of the class skeleton exported from the Enterprise Architect, using the behavioral design from the previous stage as a guideline. The code for some or all of the methods might be automatically generated. At this stage, the POSIX Thread API is used for multithreaded programming. This API is the most common interface for thread programming, with support in virtually all OS, including embedded OS such as Linux or Xilinx's lightweight embedded kernel xilkernel.

Next, the testing step includes both functional and correctness tests. Correctness tests assure that the available resources are correctly used -e.g, that the solution does not have any memory leaks. This testing stage uses the tools and resources available in a general purpose processor, making it much easier to obtain a functionally correct complete solution. Any changes that may be needed at this point to get to a functionally correct solution need to be reflected in the UML OOP Design in the previous stage. To do this, a very useful capability of the Enterprise Architect and many other UML capture tools is reverse engineering, so the UML design can be captured from C++ code. Tests developed at this stage can also be used for testing in the embedded platform later on. Functional verification against the specification can also be done.

The tool used is the Eclipse IDE, with the GCC compiler and GDB debugger. The GNU valgrind is used to assess correctness in the memory usage. All of them are widely used open source tools.

The output of this stage is a correct, executable solution of the problem running in a general purpose processor. This software solution may be used both as a reference model and as part of the final software that will run in the embedded processor, given that it uses the right language for codification, as well as widely ported libraries for OS services, such as threads or memory management. This reduces the software coding effort.

3.3 Software Migration, optimization and HW/SW partition

In this stage, the whole software solution must be migrated to the final embedded processor in a testing environment, in order to fully characterize the resources

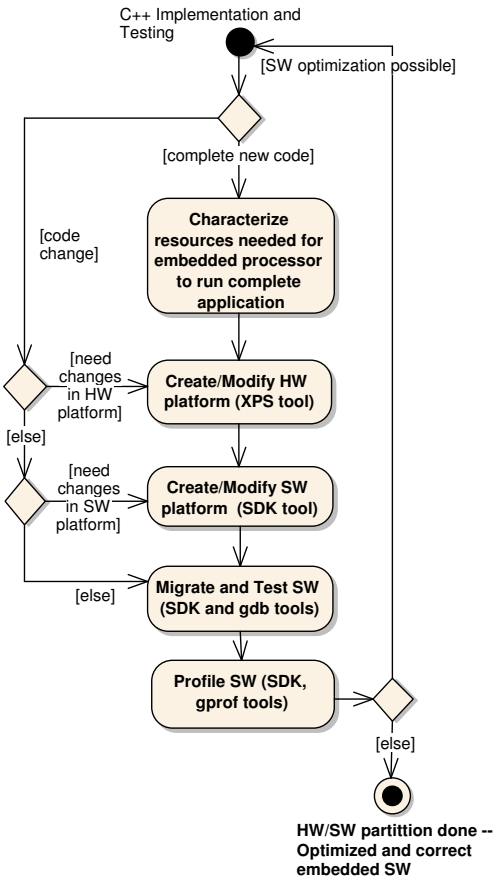


Fig. 3: Activity Diagram for the software migration, optimization and HW/SW partition stage

needed in the embedded platform, perform all possible software optimizations, and decide which parts of the system need to be accelerated by hardware. See Fig. 3 for an Activity Diagram of the steps in this stage.

The first step is to characterize the resources needed to execute the complete software solution in the embedded platform. This includes an analysis of the amount of memory needed, the different peripherals and their interfaces, and a general measure of the processor usage. It must also be determined whether an embedded OS will be necessary or not.

The second step is to configure and generate the hardware platform to execute the whole software solution in the embedded processor. The configuration of the embedded processor includes processor frequency and bus frequency, debugging and profiling configurations, coprocessors such as floating point units, internal memories and caches, interrupts, buses for peripherals, etc. The output of this step is all the necessary hardware so that the embedded processor may run, including inputs and outputs for the application, for debugging and for profiling.

The third step is to configure the software platform in the embedded processor. This includes the choice and configuration of operating system (if any), input/output libraries, debugging and profiling libraries and memory map. The output of this step is a software platform configured to run the whole software solution in the embedded platform.

The fourth step is to migrate the software solution to the embedded platform. The changes in the code needed for this migration depend on the embedded processor, needed peripherals and software platform. For a C++ solution with standard libraries such as malloc or pthread, most embedded operating systems and processors need only slight changes in the library interfaces to communicate with peripherals and memory. The output of this step is a complete executable software solution in the embedded platform.

The final step for this stage is to profile the software solution running in the target embedded processor. The profiler is useful to point out the methods that are time-consuming, although it is important to note that it may be impossible to profile some libraries that have not been compiled with the appropriate flags and also that profiling does not work with multi-threaded environment. For more reliable time measures in these settings, the internal processor's clock or some specially included external timer may be needed. Using this profiling information, the most time consuming methods are pointed out.

If possible, software optimizations for the particular architecture of the embedded processor can be done (e.g. the use of fixed point arithmetics in a non-FPU processor). Method optimizations that change the precision of the algorithms but save operations can also be studied. Taking into account that the tools for debugging and functional tests in a general purpose processor are much better than in the embedded processor, the changes are first performed in the C++ code in the general purpose processor. This affects the previous stage of the methodology (C++ Implementation and Testing) and probably even the OOP Design stage, as shown in the alternative branch in Fig. 3. As can be seen in that figure, the changes in code may or may not call for modifications in the previously generated hardware and software platforms. For example it may require to add Block RAM memory (i.e., hardware platform change) or a new software library (i.e., software platform change). This is an iterative stage: the procedure might be repeated as long as the real-time performance is not reached, and more software optimizations may be done.

Once no more software optimizations are possible, the final profiling points to the methods that need to be accelerated by hardware to achieve the required performance. Hence, the output of this step is the final hardware/software partition.

The tools for this stage depend on the embedded processor and FPGA, development boards and associated tools to be used. In this case study, the Avnet Virtex4-FX12 evaluation board was used. This board includes a Xilinx Virtex4 FPGA with an embedded PowerPC405, so Xilinx's Embedded Development Kit [46] is used. This kit includes the Xilinx Platform Studio (XPS) tool for hardware platform configuration and generation, and the Software Design Kit (SDK) used for software platform configuration and embedded software migration. SDK is an Eclipse-based tool that comes with a special GDB debugger and a special integrated GNU gprof profiler.

The output of this stage is the hardware/software partition and an optimized, functionally correct software solution running in the embedded processor.

3.4 Hardware translation, testing and integration

The fourth stage is to translate the C++ methods that need to be accelerated by hardware to HDL, totest and integrate them into the system, by implementing the needed interfaces in hardware and software. In Fig. 4 the Activity diagram of this stage can be seen.

In this work, three ways to translate the selected C++ methods to HDL are evaluated: the ROCCC tool, the AutoESL tool and the two-process coding method. These are shortly described in the following subsections. The entity and its interfaces are tested using simulation and testbenches, which change depending on the tool used. The output of this step is the hardware IP core that passes all the simulated tests.

The following step involves performing unit tests for the IP core implemented in the embedded platform. Hardware and software platforms needed to run these tests in the embedded processor must be generated, following the steps mentioned in Section 3.3. Then, unit test cases developed in Section 3.2 must be migrated. New test cases that take into account the particular details of the hardware implementation and its interaction with the embedded processor must also be done. The output of this step is the tested hardware IP core.

Finally, integration tests must be done to see if the whole system with the added hardware module is still functionally correct. For this purpose, the system tests developed in Section 3.3 can be migrated. The output of this step is the tested system, including the added hardware module.

Since only part of the system is accelerated by hardware, it is important to keep in mind the theoretical maximum expected improvement to the overall system. That theoretical maximum S'_{max} can be calculated by Amdahl's law. This law is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation, where the improvement has a speedup of S. For example, if 30% of the computation may be the subject of a speedup, P will be 0.3; if the improvement makes the portion affected twice as fast, S will be 2. Amdahl's law states that the overall speedup S' of applying the improvement will be:

$$S' = \frac{1}{(1-P) + P/S} \tag{1}$$

The theoretical maximum S'_{max} can be calculated by assuming that the portion P will have a infinity speedup ($S = \infty$). This maximum helps to indicate when translating a part to hardware or increasing parallelism is not worth the effort, since the impact on the complete application will not be significant enough. This equation is also useful to check if the measured complete accelerations are consistent with the measured partial accelerations.

The tools for this stage also depend on the embedded processor and FPGA. In this case study, Xilinx's Embedded Development Kit is used for hardware and software configuration and development. Xilinx's ISE is used for hardware modules implementation, and Xilinx's ISim simulator is used for unit simulation and testing. The AutoESL and ROCCC semi automatic tools are used to generate HDL code for the C++ methods.

The output of this stage is the tested system, including the complete software and all the integrated hardware modules.

3.4.1 The ROCCC tool

The ROCCC compiler tool by Jacquard Computing is designed to create hardware accelerators from a subset of C. The hardware generated is not intended to replace entire software applications, but instead provide an application speedup by replacing critical regions in software with a dedicated hardware component. Users code hardware module in C, and then use these modules in larger programs. ROCCC generates platform independent VHDL code from C descriptions.

In order to run the VHDL code on a particular platform, users must create glue code that attaches the automatically generated code to the system. There are specific guidelines as to how to present the data to the automatically generated hardware module, so the glue code must follow those guidelines. The tool implements several optimizations -loop unrolling, systolic array generation, pipeline, etc- which must be configured by the user in order to achieve an optimized code.

The ROCCC GUI is a plugin designed for the Eclipse IDE that works on both Linux and Mac systems. After VHDL code has been generated, the modules need to be imported to Xilinx ISE or similar in order to synthesize the design for the target FPGA.

The ROCCC is an open-source development, so download, documentation and examples are available for free from Jacquard Computing.

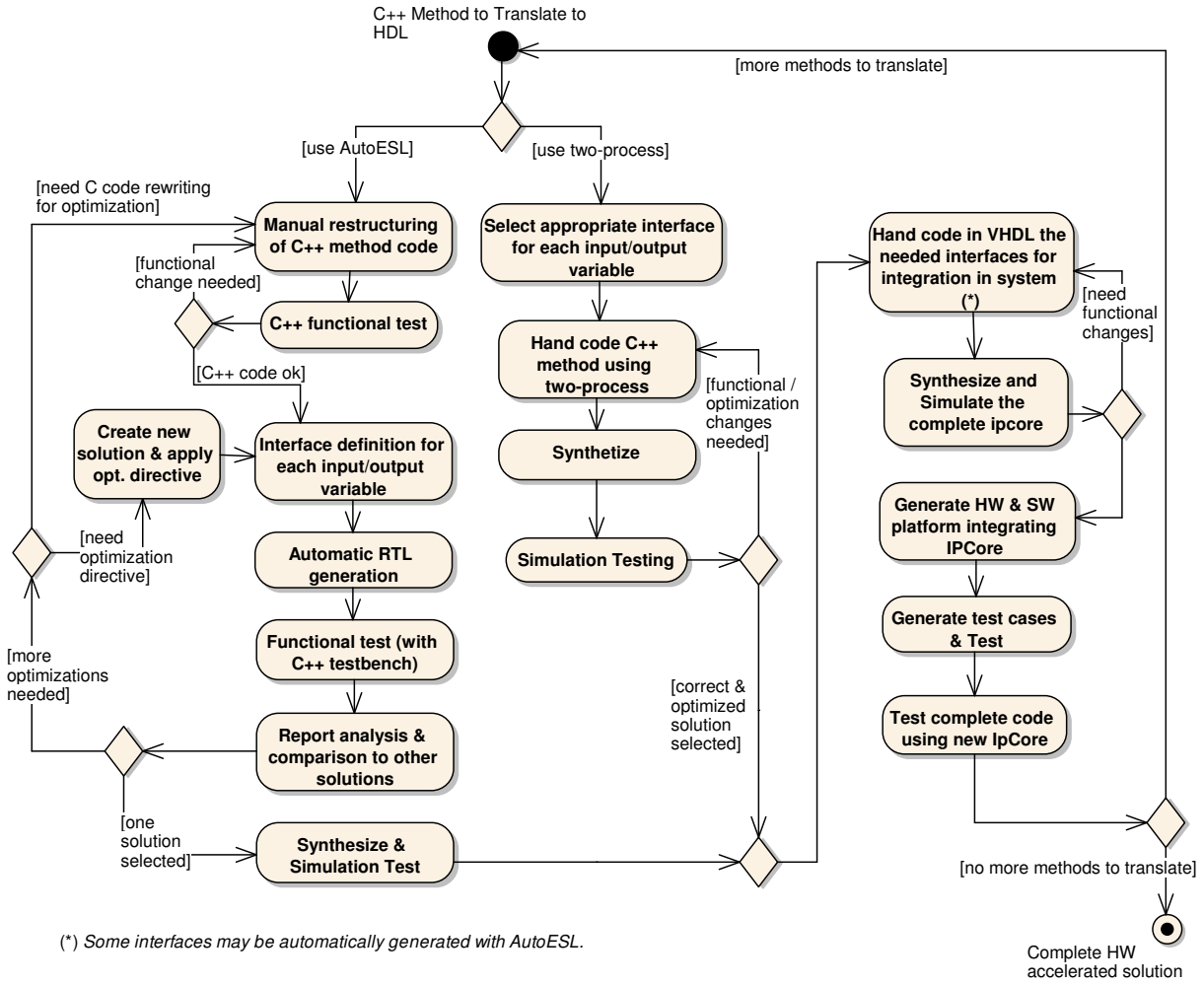


Fig. 4: Activity Diagram for the hardware translation, testing and integration stage, including the choices to use the AutoESL tool or the hand-coded two-process

3.4.2 The AutoESL tool

The AutoESL is a High Level Synthesis tool developed since 2006 by the AutoESL company, which was acquired by Xilinx in 2011. Xilinx has released this tool and also used it as a base for the new Vivado Design Suite. AutoESL takes as its input a C, C++ or SystemC description of functionality at a high level of abstraction. It then generates a device-specific Verilog or VHDL description of a hardware implementation.

AutoESL has several optimization directives that can be applied to a given design. From a given C/C++ code, different HDL solutions can be achieved using different directives. AutoESL provides reports that compare each solution in terms of timing, area and power consumption of the generated design. Although these measures are estimates, in a few minutes they enable the designer to see the achieved solution after applying an optimization directive, and also to compare it with other possible solutions.

The AutoESL GUI is an eclipse-based complete tool. It can be used to code and test the C/C++ original code. Moreover, the C/C++ testbench can also be used to test the generated design. In order to create some optimizations (like the use of bit-accurate variables) special C/C++ types exist and the C/C++ code may need changes. The Xilinx ISE tool is integrated into AutoESL, so the design can be synthesized from within the tool. Moreover, for some interface types, AutoESL can create automatic master or slave ports for Xilinx EDK's standard buses. This greatly simplifies the integration of the generated modules. However, for other types of inter-

faces -such as particular memory access patterns or other buses- the interface modules need to be hand coded.

AutoESL is a proprietary tool, with a substantial price per license and many years of development from both the previous company (called AutoESL) and Xilinx.

3.4.3 The two-process design method

Two-process is a structured VHDL design method that is particularly well suited for implementing OOP methods. The main goals for this design method are to provide a uniform algorithm encoding, increase abstraction level and improve readability and bug finding. These goals are reached by simple means: using record types in all ports and signal declarations, using only two processes per entity, and high-level sequential statements to code the algorithm.

The biggest difference between a program in VHDL and standard programming language, such as C, is that VHDL allows concurrent statements and processes that are scheduled for execution by events rather than in the order they are written. This reflects indeed the data-flow behavior of real hardware, but becomes difficult to understand and analyze when the number of concurrent statements passes some threshold. On the contrary, analyzing the behavior of programs written in sequential programming languages does not become a problem even if the program tends to grow. These programs are easier to understand because there is only one thread of control, and execution is done sequentially from top to bottom.

The two-process method only uses two processes per VHDL entity: one process that contains all combinational (asynchronous) logic, and one process that contains all sequential logic (registers). Using this structure, the complete algorithm can be coded in sequential (non-concurrent) statements in the combinational process while the sequential process only contains registers, i.e., the state.

In this way, methods of classes that need to be migrated to hardware can be translated as one VHDL entity, coding the whole algorithm using very similar sequential statements to the ones used in C++. For this stage, a correct, modular OOP design is vital, so that every method to be translated is short and concise. The key is not having to rethink the algorithm to accommodate the concurrent nature of hardware, but translating the algorithm from C/C++ syntax to VHDL syntax in a sequential manner.

The two-process method has shown to greatly decrease man-years, code lines and bugs. In [10] some comparisons for ESA projects can be found, including a comparison between the ERC32 memory controller MEC (designed with ad-hoc methods) and the whole LEON3 processor designed with the two-process method. Even though the LEON3 processor is a 100k gates design and the MEC is only 30k gates; the LEON3 took 2 man-years, 15000 code lines and had no bugs in the first silicon, while the MEC took 10 man-years, 25000 code lines and had to go through 3 silicon iterations.

All these reasons make the two-process method a good choice for translating the C++ object methods to VHDL.

Fig. 5: SyRoTek arena and robot with dress arc.

4 Multiple Robot Localization

The System for Robotic Teleeducation (SyRoTek) [19] is an e-learning platform for distance education of artificial intelligence, control engineering, motion planning and other fields related to mobile robotics. It has been successfully used in education and research by institutions across Europe and the Americas. The platform consists of fourteen autonomous mobile robots operating on a 24/7 basis in an enclosed area with dynamically reconfigurable obstacles (see Fig. 5). Users anywhere around the world can upload their algorithms to the robots, gather their sensorymotor data and analyze their behavior.

An important component of the platform is a visual localization system, which determines position and heading of each robot in the arena. It consists of a dedicated PC, an overhead camera and unique identification patterns placed on the individual robots. Due to the system 24/7 operation, it is desirable to implement the localization system on an embedded device with low power consumption. The real-time constraint for the system is that it should be able to process 1600×1200 pixel images at a rate of 30 fps.

4.1 Method overview

The 1600×1200 gray scale image that is provided by the localization system camera is processed in four consecutive steps. In the first step, the image is transformed to make the arena appear as a rectangle aligned with the image edges. The rectified image is then convolved with a 40×40 annulus pattern, and local maxima of the convolution are found. After that, endpoints of the robot dress

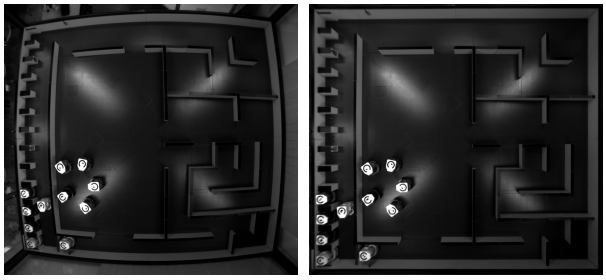


Fig. 6: Original and rectified arena image.

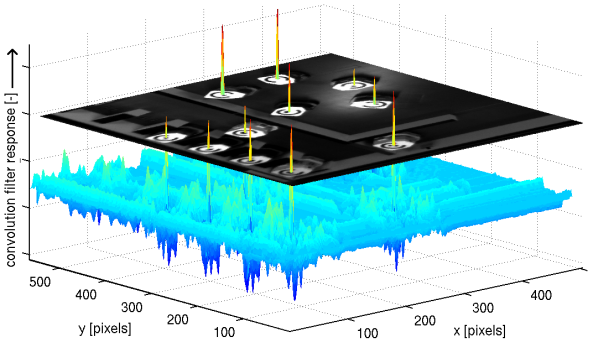


Fig. 7: Rectified image part and convolution filter response

arcs are found to determine the robot heading. Finally, binary identification tags at the robot dress centers are decoded.

4.2 Image rectification

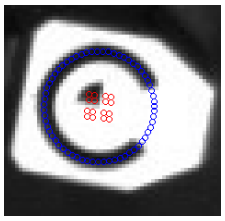
The purpose of this step is to remove radial and perspective distortion of the arena image, so that it appears as a rectangle aligned with the image edges (see Fig. 6). The radial distortion, which is caused by camera lens imperfection, was modeled by the method described in [50], and its parameters were established by using the MATLAB calibration toolbox [5]. The perspective transformation, which results from the camera misalignment, was modeled by a 3×3 projective transformation matrix. This matrix was calculated from the positions of the arena corners in the undistorted and rectified image by means of solving a set of linear equations.

Using the established parameters of both transformations, a look-up table mapping pixel coordinates of the rectified and captured image was generated. The look-up table allows to perform both transformations in a single step, thus reducing the number of floating point operations per pixel of the generated image. Since the mapping of the pixels is not one-to-one, the brightness of each rectified image pixel was calculated from four pixels of the captured image by bilinear interpolation.

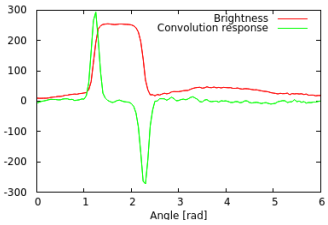
4.3 Position estimation

The circular shape of the robot dress outer arc allows to decompose the robot localization to 2D position estimation followed by orientation calculation. To determine the robot position, the rectified image is convolved with an 40×40 pixel annulus pattern with outer and inner diameter equal to the sizes of the dress arc (see Fig. 5). The response of the convolution filter is then searched for local maxima, which indicate robot positions in the arena (see Fig. 7).

Given the limited robot speed, camera resolution and fps, the convolution of the entire image is not necessary.



(a) Sampled pixels



(b) Convolution response

Fig. 8: Orientation and identification process

During standard system operation, the convolution is performed only in a neighborhood of each robot's position in the previous frame. This also means that the image rectification can be performed only in the areas where convolution is about to be calculated. Processing of the entire image is performed only when the system starts, resumes from idle state, or if the robots are removed or added to the arena. The system administrator can also force to search robots in the entire image in the case the tracking algorithm fails.

4.4 Orientation calculation

As soon as positions of the robots are known, the robot orientation is established from the dress arc. First, positions of several sampling points along the dress ring are calculated. The brightness of each point is estimated from its neighbouring pixels by bilinear transformation, constructing a vector that contains the brightness of the pixels on the dress ring. The vector is normalized and convolved with a kernel corresponding to the expected brightness gradient at the endpoints of the dress arc (see Fig. 8).

Minimum and maximum of the convolution correspond to the positions of these endpoints, and therefore, orientation of the robot is computed as the average of

the argmin and argmax of the resulting vector. To verify the angle estimation, the distance of the found minimum and maximum is compared to the dress arc angle.

4.5 Robot identification

The last step establishes the robot number by decoding a binary tag in the robot dress center. Once the position and orientation of the robot is known, brightness of sixteen pixels around its center is measured (see Fig. 8). Average brightness of each pixel group is then calculated and a threshold separating white and black values is established (each identification tag has at least one white and one black segment). The calculated brightness are thresholded and the sequence of the four results encodes the robot number. Since the robots are tracked, the identification is performed only once. After that, the identification is run only to verify correctness of the localization algorithm.

5 Hardware/Software co-designed solution

In this section, the steps of the proposed methodology were applied to the multiple robot localization problem in order to achieve the accelerated embedded solution.

5.1 OOP Design

In this stage, an OOP Design was made and expressed in UML diagrams. Also, parallelizable sections were identified. The overall structural design of the solution is shown in Fig. 9.

The `Robot` class contains the information of each robot, i.e., position, heading and id. The class `PositionCalculator` calculates the new position of a robot. For this task, the `exec` method takes as parameters a Robot and a neighborhood of 50×50 pixels of the image around its position in the previous frame. By convolving the 40×40 `conv_mask` in that image section, the new position of the robot is found. The sizes for the convolution masks and neighborhood are configurable. In Fig. 10, a sequence diagram for the calculation of the new position of one robot can be seen. The class `AngleCalculator` calculates the new heading of a robot, by sampling the arc points on the 40×40 unbarreled section of the image where the robot actually is. The identification of the Robot is done with the method `getRobotID` of the class `PositionCalculator`.

Matrix operations are performed by a `Matrix` class. Since most matrices are sub-matrices of bigger ones (e.g., an image section is a sub-matrix of image), memory is only dealt with in very specific moments. The `LoadableMatrix` class inherits from `Matrix` and performs actual memory movements. Finally, the `Image` class depends on `LoadableMatrix`, as it has an instance of that class to contain the image data. The `Image` class knows about image undistortion operation, implementing both bilinear and nearest neighbor interpolation for comparison purposes.

It is important to note that there are a variety of possible structural designs for the solution. In the software community, much work has been done in design pattern, starting in 1994 from the foundational book "Design Patterns: Elements of Reusable Object-Oriented Software" [11]. Evaluating which patterns are applicable and useful in embedded software, and which are not, is a promising and vast research area.

At this stage the possible parallel sections need to be identified. Analyzing the algorithms allows one to see that the different stages in finding a robot's new position and orientation are not parallelizable. For the position calculation, the image section the robot was in needs to be already undistorted, and then for angle calculation the robot's new position is needed. However, the complete process for each robot is independent of any other robot, so the complete process can be done in parallel for each of the fourteen robots, for example in fourteen different threads. These are shown in the activity diagram of Fig. 11, where the bars show the fork and joins, and the horizontal swimlanes show which class is responsible for each activity.

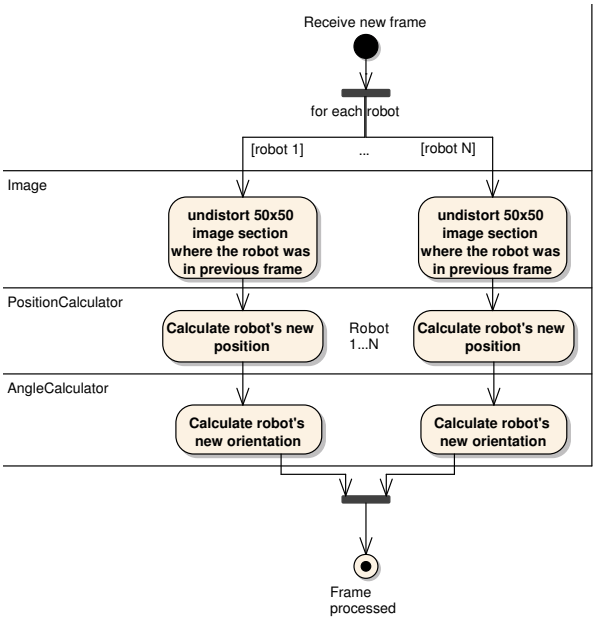


Fig. 11: Activity Diagram showing the parallel nature of the new position and orientation calculation for each robot (two robots in this diagram).

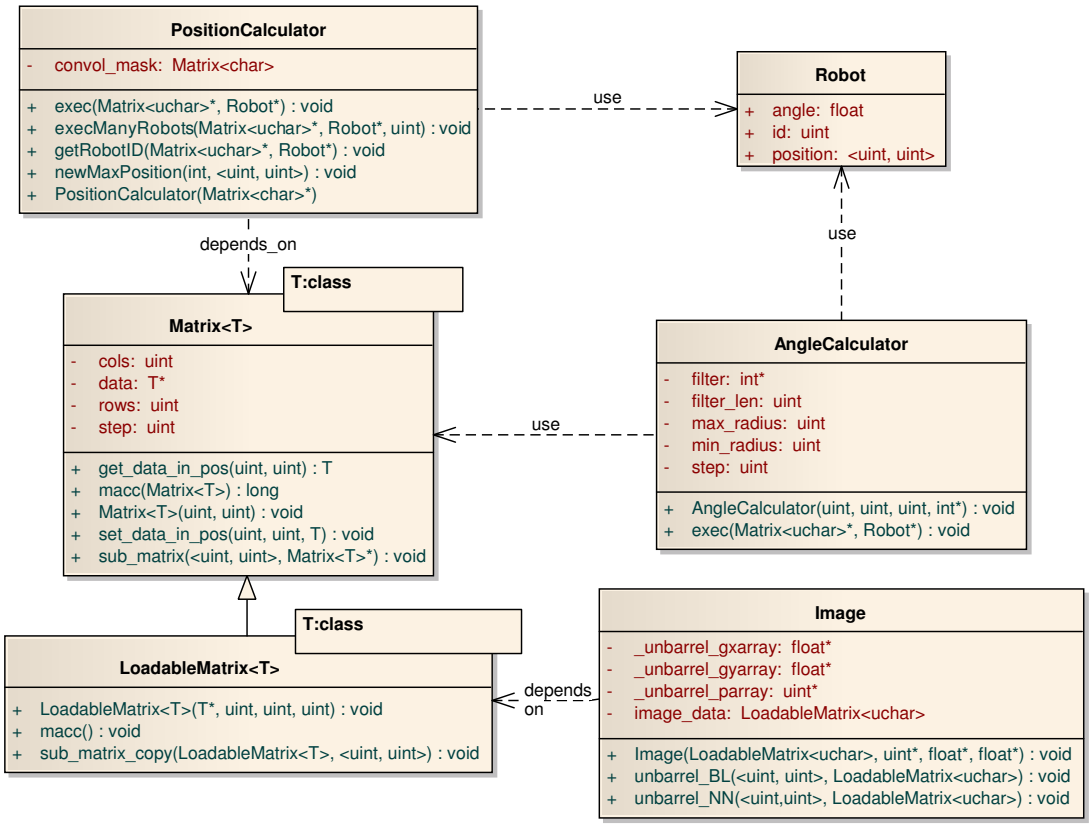


Fig. 9: Structural Design

5.2 C++ Implementation and Testing

The class skeletons were automatically generated with the Enterprise Architect, along with partial code for some methods. The whole system was coded using C++ and the Eclipse IDE. For multi-threaded programming, the Posix threads C library was used, implementing the calculation of each robot's position and orientation in a separate thread.

Unit tests were developed for the Matrix classes and also system tests with a test suite of images from the arena and the known positions and angles of the robots. The OpenCV library was used for image handling.

5.3 Software Migration, optimization and HW/SW partition

An Avnet development kit including a V4-FX12 FPGA with a PowerPC405 embedded processor was used. The development tools used were Xilinx's Design Suite 11.2 for hardware and embedded software development, and GNU valgrind and gprof for preliminary resource characterization.

First, the peripherals, memory and resources needed to run the software solution in the embedded processor were characterized. The application needs extensive memory, since the image is a 1600×1200 grayscale image and there are three precalculated undistortion arrays, each storing one floating point per pixel. That is close to 24 Mbytes of required memory. Hence, the on-chip fast BlockRAMs included in the FPGA were not enough, and off-chip memory, such as Flash and SDRAM, was needed. For initial tests, the images from the camera could be loaded into this external memory, so the peripherals related to image capture could be -for the time being- left apart. To have an idea of the necessary processor resources, profiling was done in a general-purpose processor. The execution time to process an image with 14 robots in a Core i5 M480 (2 cores@2.67GHz) is 30.74 ms, including undistortion, localization and angle estimation for each robot.

Using this resource analysis, the hardware platform needed to run the software solution in the embedded processor was generated. The memories included were Flash, SDRAM and Block RAMs, and they were connected through an IBM PLB (Processor Local Bus) bus to the

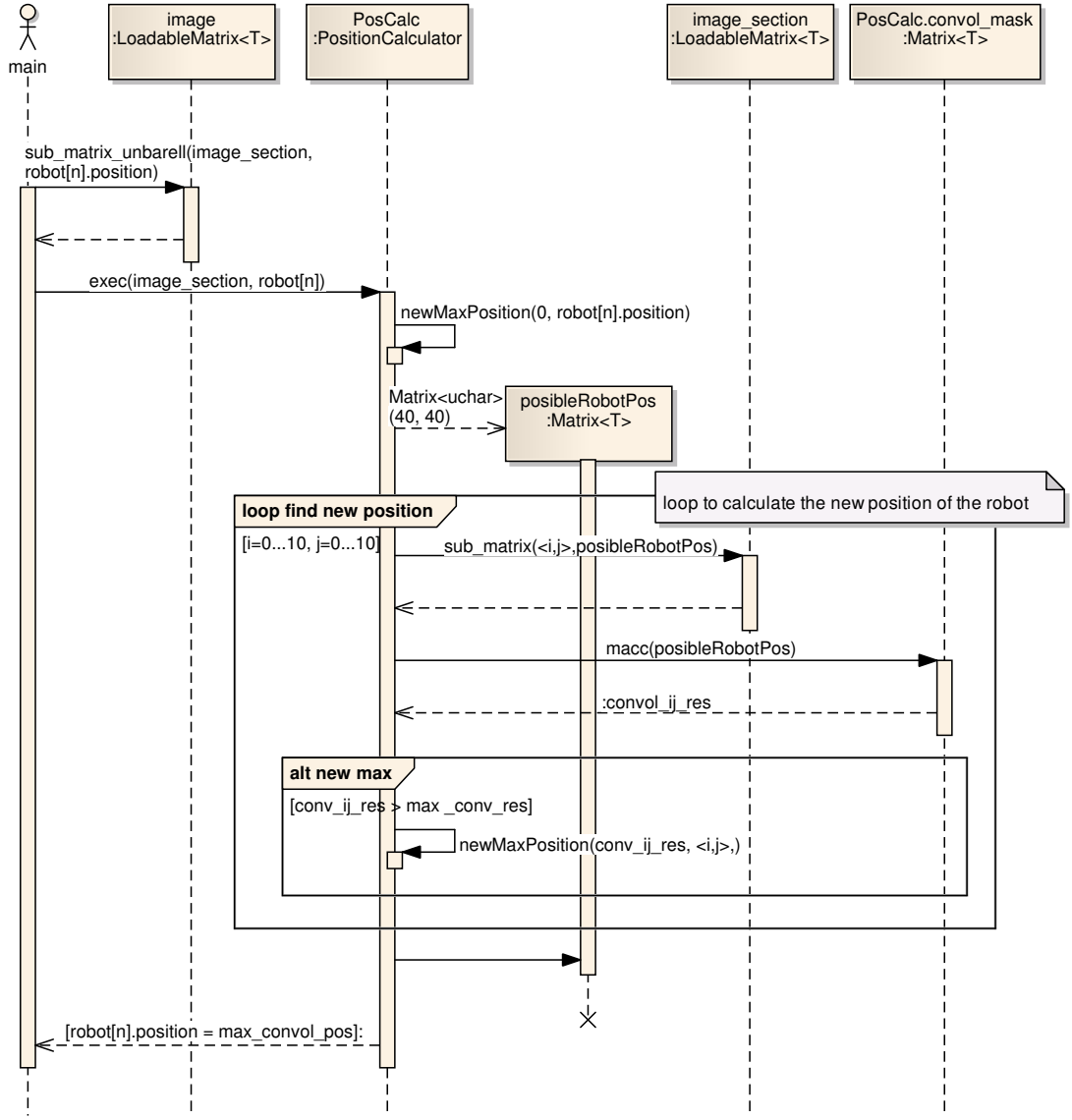


Fig. 10: Sequence Diagram for the calculation of the new position of one robot

processor. The Block RAMs were included so that small image sections that are used many times (such as the 50×50 pixel neighborhood of the robot) could be stored in these on-chip memories. The PowerPC405 data and instruction internal caches were also configured. For the initial testing phases, special programmable logic modules for debugging and profiling were also added. The PowerPC405 and the PLB bus were set at their maximum frequency (300 MHz for the PPC, 100MHz for the PLB).

Next, the software platform needed to be generated. Since the solution uses threads, an embedded operating system with thread support is needed, preferably following the POSIX API. Xilinx offers support for two possibilities: xilkernel and Linux. Xilkernel is an open source kernel shipped with EDK, which supports the core features required in a lightweight embedded kernel, with a POSIX API for thread and mutexes. The Linux distribution does not come with EDK but it is available for compilation for the PPC405 core. Both are suitable solutions for the application, but xilkernel offers a simpler solution that is enough for the requirements of this application.

However, since the target platform has only one processor with one core, the execution of all threads is sequential, and hence there is no need for multi-threaded programming for the initial software optimizations. In this platform, the use of threads can only come in handy for parallel processing if more than one hardware accelerator is included, as explained in the following subsection. Using a standalone (not OS) platform in this stage simplifies measuring execution times (and debugging), and allows the use of Xilinx's profiling tool to guide the software optimizations (since this tool only works in the standalone platform). Hence, for this stage, a standalone software platform was generated for the processor.

A version of the C++ code without threads was built and tested in the general purpose processor, since that is the base for software migration in this stage. The migration of the complete software solution to the embedded processor required only two minor changes. In the embedded solution, images were loaded from the Flash memory instead of using OpenCV, and dynamic memory for image sections was replaced by BlockRAMs. These interface changes were encapsulated in a single configuration file, so the rest of the code was unchanged.

Finally, the complete software solution was profiled in the embedded processor. Since Xilinx's profiler does not measure the time completely (e.g., the time for interrupts or some of Xilinx's libraries), it was only used to estimate the percentages of time spent in each method. The real overall time that the application takes was measured using the internal timer of the PPC, saving the timestamp when execution starts and then when it ends. These time measurements were corroborated with oscilloscope measures.

Since the PowerPC405 has no FPU, all floating point operations were emulated by Xilinx's library. Hence, software optimizations were developed for the PowerPC's particular architecture. Profiling results for each code version are shown in Table 1, and the corresponding bar graphic can be seen in Fig. 12.

The first column corresponds to the original code. The complete software solution takes 1.6 seconds. Most of the time is spent in angle calculation, so it was the first thing to tackle. The first optimization consisted in using pre-calculated cosine and sine masks to find the arc-points that needed to be sampled for angle calculation (see second column). Also, all floating point operations in the angle calculation were changed to fixed point arithmetics (see third column). These changes took the total time down to 0.9 seconds, and the percentage of time spent in angle calculation down to 1.71%.

At this point image undistortion and matrix convolution took almost half of the time each. Image undistortion could be simplified by taking the nearest neighbor to calculate the pixel in the undistorted image, instead of the bilinear interpolation with the four closest pixels. Of course, the nearest neigbour interpolation would decrease precision of robot position and angle estimation.

The impact of nearest neighbor estimation on the algorithm precision can be estimated as follows. The nearest neighbor estimation can be modeled as a quantization noise, which adds errors up to half a pixel. It can be assumed that this maximal noise- i.e half a pixel- does not appear more than twice per one robot dress since the distortion is not so high. Therefore, in comparison to the bilinear transformation, the robot position estimation precision might be decreased by one pixel, which corresponds to an extra 3 mm (or 0.1% relative to the arena dimensions) localization error. If both the arc endpoints and the robot position estimation are affected by the noise, the angle estimation algorithm errs by two pixels. Since the arc circumference is approximately 120 pixels, the maximal introduced error is 0.1 radians (or 1.6%). To verify the aforementioned estimations, one thousand images of the arena were taken with robots at known positions, and the localization errors with the bilinear and nearest neighbor undistortion methods were calculated. The largest error introduced by the nearest neighbor interpolation was 1.59% for angle estimation and 3 mm for position estimation, which is in good accordance with the aforementioned calculation.

Switching to nearest neighbor interpolation caused the image undistortion times to fall dramatically (see fourth column). The same change was introduced for the brightness calculation in the angle estimation (see fifth column). The aforementioned precision loss was far outweighed by the increase in the algorithm's efficiency.

All these changes were first implemented and tested in the general purpose Corei5 processor, using its debugging and testing resources, and keeping the golden reference model up to date. Migration to the PowerPC did not require code changes. The test suite was images with fourteen robots in the arena loaded in the Flash memory. Results for profiling in the Corei5 processor are also shown in this table. The fastest code was used for this test (including all optimizations and floating point arithmetics).

The final column for the PowerPC in the profiling table shows that 95.46% of the time is spent in the `Matrix::macc` method. Although so far all methods could be optimized by software taking into account the PowerPC architecture, the `Matrix::macc` that does the convolution could not be accelerated by software. Moreover, although slightly over 3× acceleration was achieved with software optimizations, the total time was still very high, and only enabled a 2 fps throughput. The only solution for decreasing this time and get closer to the 30 fps goal was to accelerate the `Matrix::macc` in hardware.

This method is called 100 times in `PositionCalculator::exec`, which searches for the new position of a robot, representing almost all the time spent in position calculation. It is important to note that the modularity of the OOP design and the encapsulation of the matrix operations in a separate class allowed the profiling to ac-

Table 1: Profiling results for software optimizations. Times for complete solution in milliseconds.

	PPC405@300 MHz					i5@2.67GHz
	orig. code	cos_mask	fixed pt.	unbarrel_NN	angle_NI	all opt.
Matrix::macc	28.56%	44.13%	51.00%	93.95%	95.46%	93.33%
angleCalc::exec	44.96%	14.96%	1.71%	3.16%	1.60%	2.73%
Image::unbarrel	26.48%	40.91%	47.29%	2.89%	2.94%	2.34%
complete solution	*1630*	*1078*	*926*	*501*	*495*	*30.74*

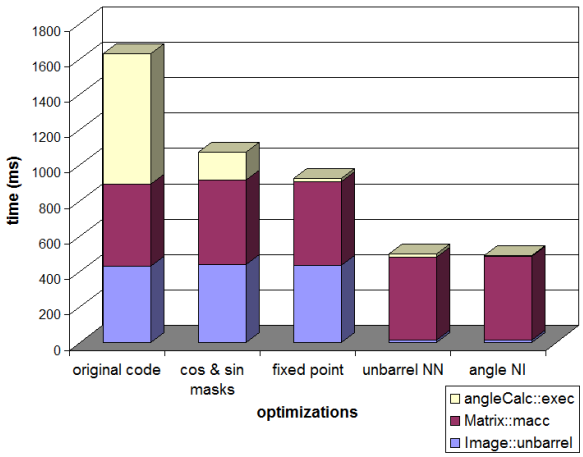


Fig. 12: Profiling results for software optimizations in the PPC.

curately point where the most time-consuming operation was, thus preventing useless translations to hardware.

An output of this stage was the complete, correct and optimized software version running in the embedded PowerPC405 processor. The definite hardware-software partition led to the translation of the `Matrix::macc` method to hardware.

5.4 Hardware translation, testing and integration

Next, the hardware module for the `Matrix::macc` was implemented, including its interface with the memory and embedded processor. Hardware and software changes were introduced to integrate this hardware module in the solution. In this section, we show the translation using two-process. In section 5.4.2, we show a comparison with AutoESL and ROCCC solutions.

The first step was to decide which interfaces were best suited for the module. The `macc` method of a `Matrix` class object takes as a parameter a matrix of the same size to convolve with itself. Hence, a good solution was to connect the hardware module to two Block RAMs, one per matrix. The PowerPC is connected to the other port of each Block RAM so it can load the matrices data. The convolution is performed between two 40x40 matrices, but it is known that one of those matrices is part of a bigger one (the 50x50 image section). Hence, the PowerPC needs to tell the `macc` module in which address of each Block RAM the matrices to be multiplied start. The size and step of the matrixes are configurable parameters. When the convolution is done, the hardware module can send the result back to the processor. This means that for each convolution, there are three data exchanges: two addresses and one result. The default bus used by EDK 11 to connect peripherals is the PLB bus. This is a complex bus that is prepared for many different types of slaves. A much simpler bus is the Device-Control Register(DCR), a bus that can connect many slave modules in a daisy chain manner. This bus takes up less logic and is the simplest solution that achieves the desired communication pattern.

When Block RAM memories are added in EDK, they are wrapped in an interface and connected automatically through a 64-bit PLB bus. This poses the restriction that the other Block RAM port (that is, the port that is connected to the `macc` hardware module) needs to be also 64 bits wide. Hence, an interface between the Block RAM wrapper and the `macc` module needs to be coded to extract the desired bytes from the 64-bit wide memory data.

Each module (the macc module, the DCR interface module, and the memory interface modules) was tested separately and then integrated into a single ipcore. This ipcore was included in the hardware platform in XPS. Software was changed to use this hardware accelerator instead of calculating the macc in software. For this purpose, the only change was to replace the `Matrix::macc` method code by sending the two addresses through the DCR bus, sleeping the processor while waiting for the hardware module to work, and asking through the DCR bus for the result. The amount of time needed for the `macc` hardware module to complete the convolution was $17\mu s$.

The best possible complete-system performance was achieved since each part (hardware and software) ran at its maximum frequency. For this, a Digital Clock Manager (DCM) available in the FPGA was used and the connection between the embedded processor and hardware was done in an asynchronous way, i.e, using memories and the DCR bus. Table 2 shows the execution times using one hardware macc accelerator. The measures were all taken using the PowerPC internal timer.

Table 2: Profiling results for hardware accelerated solution.

	ms	%
posCalc::exec	27.35	54.82
angleCalc::exec	7.96	15.96
Image::unbarrel	14.58	29.22
complete solution	*49.89*	100

As can be seen from Table 2, with one core the reached acceleration was 9.92×: the last software accelerated version took 495 ms and the hardware accelerated version took 49.89 ms. This factor multiplied the already achieved 3× improvement through software accelerations, achieving so far almost 30× acceleration. The achieved throughput so far was 20 fps, a very good throughput but still short from the 30 fps goal. The throughput could be further improved by making use of the inherent parallel sections in the algorithm, multithreaded programming and the ability to replicate the hardware `macc` module.

5.4.1 Multithreaded programming

An analysis conducted during OOP Design indicated that the complete process of finding the new position and orientation of a robot is independent from other robots -and hence parallelizable. When having only one processor and one hardware accelerator, this property of the algorithm cannot be exploited. However, if more hardware accelerators are added, there is a chance for parallelization.

As already discussed, a xilkernel platform with thread support was set up in the PowerPC; and the complete process (undistortion, position calculation and angle calculation) for one robot was placed in separate threads. Since each position calculation calls the `Matrix::macc` method 100 times, and that method sleeps waiting for the hardware to end, the processor is idle to execute another thread during that time. The most usual way for multithreading scheduling is preemptive multitasking, in which the OS decides when to switch the thread context, using a scheduling policy. However, the time slot assigned to each thread in xilkernel scheduler is 10 ms, too large compared with the 17μs each thread is sleeping while waiting for the hardware to end. Hence, cooperative multitasking needs to be used. In this approach, each thread relinquishes control when it reaches a stopping point, using the `yield()` function that makes the next thread to continue execution. While one thread is waiting for one hardware convolution to end, the other thread can send a new pair of addresses to the other hardware convolution accelerator. In this way, the processor and two hardware modules are working in parallel. This is achieved with very small software code changes: just creating the threads, using the `yield` function in the macc method, and joining the threads. The xilkernel has to be set up and the hardware accelerator replicated.

Table 3 and Figure 13 show the results for profiling with one, two, four and six hardware accelerators.

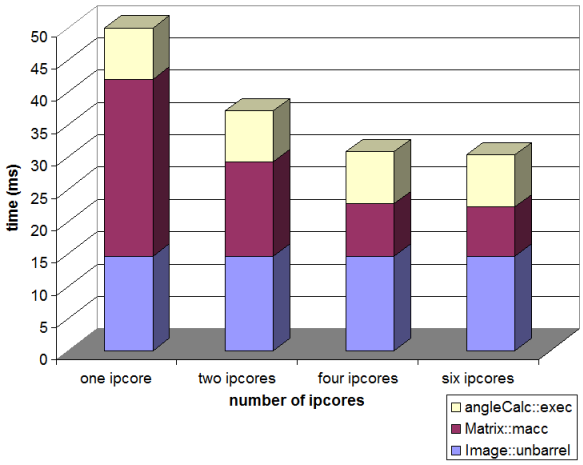


Fig. 13: Profiling results for hardware acceleration

From these results, it can be seen that with four cores the solution processes 32.5 fps, achieving the goal of 30 fps. To do so, software optimizations, hardware acceleration and parallelization using multithreaded programming and many ipcores were needed.

5.4.2 AutoESL, ROCCC and two-process comparison

For comparison purposes, the AutoESL and ROCCC high level synthesis tools were also used to synthesize the methods that needed hardware acceleration. With the ROCCC tool only the most time consuming part of the `Matrix::macc` method was implemented: the multiplication and accumulation of two vectors, without taking into account that they were matrices. This was done in this manner since the ROCCC version used did not have support for multidimensional arrays -this has been included in the latest version. Table 4 presents a comparison of area, frequency and VHDL code lines between the ROCCC generated code and the two-process code.

This table shows that the area requirements for the ROCCC generated code are around 11 times higher than the two-process implementation. The maximum frequency obtained with the ROCCC tool is 57% of the one obtained with the two-process. The C code had to be rewritten in order for the ROCCC generator to work. Moreover, there are specific guidelines as to how to present the data for the hardware module to use, so all the modules to feed data need to be hand-coded in VHDL/Verilog, as the modules to connect with the PPC.

Table 3: Profiling results with one, two, four and six hardware accelerators

	one ipcore		two ipcore		four ipcore		six ipcore	
	ms	%	ms	%	ms	%	ms	%
posCalc::exec	27.35	54.82	14.66	39.55	8.13	26.44	7.62	25.18
angleCalc::exec	7.96	15.96	7.83	21.12	8.04	26.15	8.06	26.64
Image::unbarrel	14.58	29.22	14.58	39.33	14.58	47.41	14.58	48.18
complete solution	*49.89*		*37.07*		*30.75*		*30.26*	

With AutoESL, the complete `Matrix::macc` was implemented. In order to get a synthesizable C code, many changes had to be made. `Matrix` is a template class, since both unsigned and signed char matrices are used in the application (the image is unsigned char, but the convolution mask is signed). The template had to be taken away, making the translation for a particular type. Moreover, the method had to be translated from a class method to a standalone function. Hence, the attributes of the class object (rows, cols, step, data) had to be translated to either function parameters or variables. Also, for the tool to synthesize a BRAM memory port for the matrices, the matrices could not be passed as memory addresses, but had to be passed as fixed sized arrays (40×40 for the convolution mask and 50×50 for the image section). Hence, an extra parameter had to be included to tell the function at which offset of the big 50×50 matrix, the 40×40 matrix to convolve starts. After these changes were made, a first synthesizable solution was achieved, and HDL was generated.

From this solution several optimization directives can be applied, such as bit accurate data types, correct interfaces, loop unrolling or pipelining. Data types could not be more optimized, since the 8 bit char representation is the smallest possible for this problem. From the interfaces, the code had already been changed to take both matrices as single-port BRAMs. For the extra offset parameter, an `ap_none` interface was selected, so that AutoESL would not generate any particular protocol for the variable, and later on this new parameter could be included in the DCR bus. The complete module has AutoESL's `ap_hs` handshake protocol, which would later on need a hand-coded module to integrate to the DCR bus interface. Taking as a guide the already hand-coded design, a pipeline optimization to the inner loop was applied. With these optimizations, the AutoESL design achieved the two-process design throughput.

Table 5 provides a comparison between the first AutoESL synthetizable solution, the solution after optimization and two-process solution. Although comparative reports previous to actual synthesis are provided, the results in this table are the ones provided after Xilinx's ISE implementation from the automatically generated VHDL source.

It is very interesting to see that the optimized version of the automatic code achieves a smaller solution with the same throughput- and similar maximum operating frequency- as the two-process hand-coded version. It is true that the two-process is focused on code clarity and size more than area optimization, and that the optimization directives used in AutoESL were inspired in the hand-coded design. However, these results are very good for an automated tool like AutoESL. They are also consistent with the reported results in a BDTI Benchmark that implemented a wireless communications DQPSK receiver with the AutoESL tool and compared it with hand-coded design [42].

To include this solution into the complete system, all the interface modules need to be hand-coded. The DCR bus interface is not one of the supported buses in this AutoESL version, so a hand-coded interface module between the DCR bus and the `ap_hs` handshake interface is needed. Since in XPS the BRAMs are automatically included in a 64-bit wrapper and the module takes in 8 bit data, a memory interface also needs to be handcoded.

	ROCCC	Two-Process
Slices	652	59
Slice FF	779	107
LUTs	1099	112
BRAMs	2	0
GCLKs	4	1
DSP48s	1	1
Latency	3200	1600
Freq (Mhz)	125.98	216.29
lines of code	1467	170

Table 4: ROCCC and Two-Process comparison for vector MACC

	AutoESL		Two-process
	First code	Opt. code	
Slices	71	95	144
Slice FF	73	89	128
LUTs	104	125	214
BRAMs	0	0	0
DSP48s	1	1	1
Latency	4882	1606	1606
Freq (Mhz)	167	144	166

Table 5: AutoESL and Two-Process comparison for Matrix::macc

6 Acceleration, area and power consumption results and analysis

In this section, an analysis of the acceleration, power and area results is presented. All the results in this section use the hardware cores generated with the two-process method.

6.1 Acceleration

6.1.1 Overall Acceleration

This section analyzes the overall acceleration based on software optimization, hardware acceleration and parallelization with many ipcores and multithreading programming. Figure 14 shows the execution times of each different solution, and Fig. 15 shows the overall acceleration of each solution over the original code and the most optimized software version.

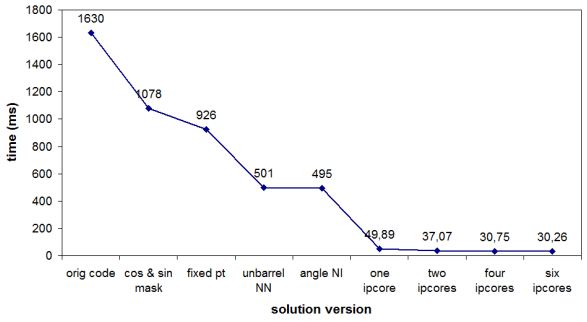


Fig. 14: Execution times of all solutions

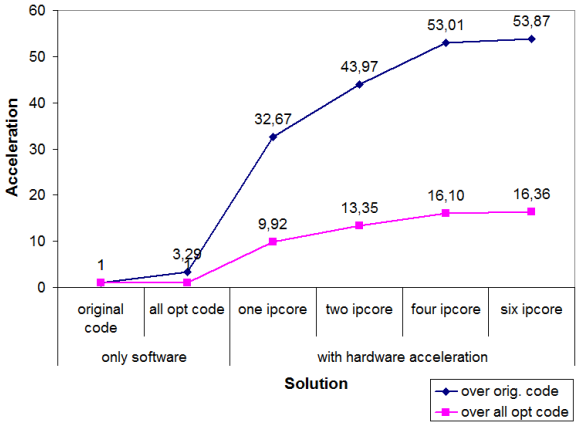


Fig. 15: Acceleration of each hardware accelerated solution compared with software solutions

From these figures, it can be seen that software optimizations account for approximately 3×, hardware acceleration for almost 10×, and parallelization using multithreading and hardware replication for another 1.6×. It is interesting to note that, while hardware acceleration yields the most speedup, it is traditionally the most time-consuming step. Software optimization, as well as multithreaded programming, are achieved with smaller software changes. This also shows the importance of research in high level synthesis area that seeks to reduce the time spent in a step that has the potential of providing vast acceleration.

6.1.2 Theoretical maximum

The maximum theoretical acceleration is an important analysis, bound by Amdahl's law, as described in Section 3.4. It demonstrates how close each solution is to the most optimized possible, and helps the designer decide when it is not worth doing any more work, because the maximum possible acceleration achievable from further optimizations is too low, as compared with the extra work required. Since Amdahl's law applies to parallelization, it makes sense to apply it only from the most optimized software version, and analyse the effect of hardware acceleration and parallelization with many ipcores and multithreading programming.

As can be seen from the profiling information of the most optimized code, the portion P that can be accelerated, i.e., the `Matrix::macc`, is 0.9546. Assuming an infinite speedup of that portion, Amdahl's law yields:

$$S'_{max} = \lim_{S \to +\infty} \frac{1}{(1-P) + P/S} = \frac{1}{(1-0.9546)+0} = 22$$

With one core, the acceleration reached 9.92× (Fig. 15). That is 45% of the theoretical maximum, so it seems reasonable to try to improve it. An interesting point is that in that solution, acceleration is not really obtained by parallelization in the sense of Amdahl's law, i.e., multicore parallelization, but by changing the implementation platform from software to hardware. When adding parallelization in the multicore sense, with four cores and multithreaded programming, the acceleration goes up to 16.1×, which is 73% of the theoretical maximum. Figure 16 shows the percentage of the theoretical maximum achieved by adding each hardware ipcore, and compares it with the most optimized software solution.

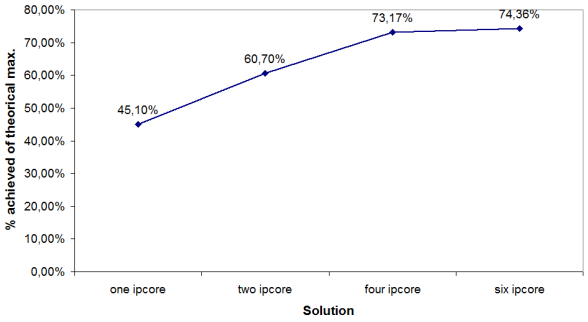


Fig. 16: Acceleration achieved with respect to Amdahl's theoretical maximum

6.1.3 Theoretical maximum for acceleration using only parallelization with many ipcores and multithreading programming

Figure 17 shows the acceleration obtained with each new hardware ipcore, and compares with the solution with only one hardware ipcore. This figure shows the acceleration taking into account only Position Calculation, which is the section of the algorithm where parallelization was really done, and also the acceleration of the complete solution.

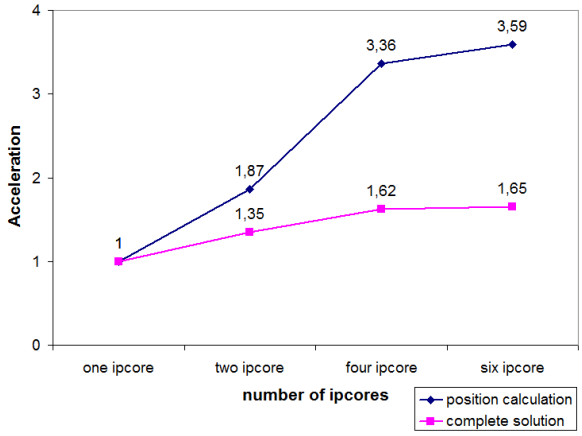


Fig. 17: Acceleration of each additional hardware ipcore, as compared with only one hardware ipcore.

As already discussed, the addition of one core amounts to 10× acceleration. The addition of four cores and multithreading programming amounts to an extra 1.6×. Although this extra 1.6× is what allows the solution to get to the 30 fps performance goal, it seems small as compared with the 10× original gain obtained by adding hardware acceleration. An interesting analysis involves using Amdahl's law to check which theoretical maximum is achievable by the parallelization step, that is, by comparing each ipcore added to the solution that already has software optimizations and hardware acceleration with one ipcore.

The portion P that can be accelerated by parallelization, i.e., the `Matrix::macc`, is 0.5482 (see Table 2). Of course, it is lower than in the most optimized software solution, since it already includes the acceleration provided by hardware implementation. Assuming an infinite speedup of that portion, Amdahl's law yields:

$$S'_{max} = \lim_{S \to +\infty} \frac{1}{(1-P) + P/S} = \frac{1}{(1-0.5482)+0} = 2.21$$

Therefore, even though 1.6× seems small, it amounts to $(1.6/2.21) * 100 = 72,4\%$ of the theoretical maximum acceleration that Amdahl's law yields.

The analysis with Amdahl's law can help decide when the possible achievable acceleration is not worth the effort of adding a new ipcore. However, in this problem there is a way of obtaining another interesting measure: what is the theoretical maximum amount of hardware maccs that can be added and still provide some acceleration? The answer to this question is related to the software overhead to manage an extra ipcore. For example, if the time spent in the `yield` function was exactly half the 17 μs that the hardware module takes, then it would not be worth adding more than one hardware accelerator. With two accelerators, the processor would not be able to feed data into both of them on time: one would always be idle. Hence, the answer lies in the relationship between the time taken by the `yield` function and the really parallel activity, in this case, the hardware `macc`. Although it is not possible to measure the exact time the `yield` takes by itself, an approximation was measured in a system that has only two threads and only yields among them, resulting in 1.9μs. This means that approximate 8.9 yields can be theoretically executed while a hardware macc is working, so that at most 7 or 8 hardware maccs can be included. This is a rough upper limit, but it still means, for example, that the position of the fourteen robots in the arena would not be able to be processed completely in parallel under this setting. The processor, which is the one executing the whole control of the application, will be the bottleneck.

After four cores, no acceleration is obtained by adding a new core (see Fig.17). That means that five threads are running: one for the main application and one for each ipcore. This is less than the calculated theoretical maximum of 7 cores, but still reasonable since this system has more threads and each thread does more than just yield, and, therefore, uses more processor time -which is now the bottleneck of the system.

Another important thing to note in Figure 17 is that when adding a second core, the obtained acceleration for

the Position Calculation is not 2 but 1.87. This is expected, since there is a software overhead for adding a core: the overhead involved in creating an extra thread and all the yielding logic between the threads, and in joining the threads after processing. In the case of four cores, the difference between the theoretical speedup of 4 and the obtained one of 3.36 is bigger. From four cores on, there is no acceleration gain. From these results it is likely that the exact point at which the processor becomes 100% busy is around four cores.

6.2 Area

The area occupied by each solution can be seen in Table 6. It should be noted that there are extensive area requirements just to get the PowerPC embedded processor and the necessary memories to run the software solution. Figure 18 shows the percentage of extra slices required for each solution.

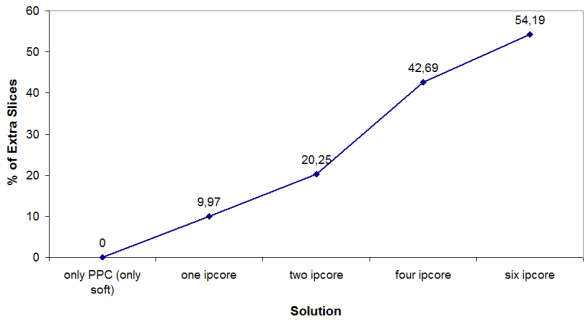


Fig. 18: Percentage of extra slices occupied as compared with the only software solution

An interesting fact to note is the impressive routing effort of Xilinx's tools, which achieve a working design that occupies 99% of the slices in the FPGA. It is also important to notice not only the occupied slices, but also each category, to get a clearer idea of the occupancy of the FPGA. The routing tool at the begining leaves more slices with less occupation, and then it not only occupies more slices, but also more LUTs, FF and other elements in each slice.

6.3 Power and Energy consumption

Table 7 presents an estimation of power and energy consumption obtained by using Xilinx's Xpower. This analysis only includes the consumption of the FPGA core: the external load and the I/O consumption are not considered. Since these are estimations, perhaps the most significant number is the saving of energy shown in the last column.

Table 7: Power and Energy consumption. Hardware implemented using two-process.

Design	I(mA)	P (mW)	T (ms)	E (μJ)	Save
opt. code	377	453	495	224	0%
1 ipcore	394	473	49,89	24	89%
2 ipcore	422	506	37,07	19	92%
4 ipcore	460	552	30,75	17	92%
6 ipcore	506	607	30,26	18	92%

Figure 19 shows the estimated energy consumption per frame, and Figure 20 presents the estimated energy saving. The energy saving is expressed in the same way as the acceleration is portrayed in the previous figures,i.e., in terms of how many times less energy each solution consumes. Of course, this shows the obvious correlation between acceleration and energy consumption: although adding new ipcores makes the power consumption higher, since the acceleration obtained is very important, the energy consumption is significantly reduced. However, when the time acceleration is smaller (e.g., from 2 to 4 ipcores) or almost zero (e.g., from 4 to 6 cores), the energy saving is not so big, or is even smaller than in previous solutions. This is because the acceleration gain is not enough to counteract the increase in power consumption. Hence, taking into account only energy consumption, adding more than 2 ipcores does not make much sense.

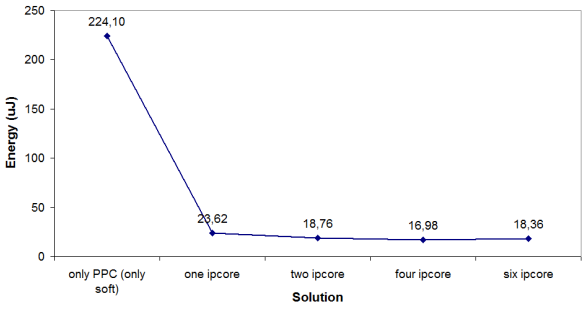


Fig. 19: Estimated energy consumption per frame for each solution.

Finally, 92% energy saving with four ipcores, as compared with the software solution, or in other words, the possibility to process one frame consuming 13× less energy, entails a very important result that advocates for hardware acceleration and parallelization in FPGA based chips.

Table 6: Area occupied by each solution. Hardware implemented using two-process.

	only PPC (only sw)		1 ipcore		2 ipcore		4 ipcore		6 ipcore	
	amount	%	amount	%	amount	%	amount	%	amount	%
Slices	**3,530**	**64**	**3,882**	**70**	**4,245**	**77**	**5,037**	**92**	**5,443**	**99**
Slice FF	4,136	37	4,379	40	5,054	46	6,408	58	7,763	70
LUTs	3,690	33	4,236	38	5,083	46	6,871	62	8,665	79
BRAMs	13	36	13	36	17	47	25	70	33	91
DSP48	0	0	1	3	2	6	4	12	6	18
PowerPC	1	100	1	100	1	100	1	100	1	100

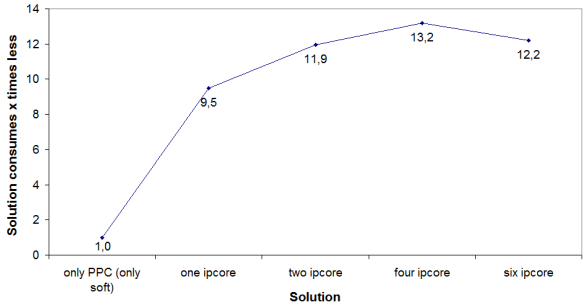


Fig. 20: Estimated energy saving.

6.4 Overall analysis

A relevant question about acceleration, area and energy consumption results would be: which is the solution that best balances all these measures? Data indicates that the solution with 2 ipcores seems more appropiate. This solution achieves almost 27 fps, a 13.15× acceleration from the most optimized code solution, 92% energy saving, and with only 20% area increase. The one-core solution only achieves 20 fps with 89% energy saving and 10% area increase. The four-core solution achieves 32 fps, but with almost no extra energy saving and doubling the area increase to 40% as compared with the two-core solution. The six-core solution, on the other hand, offers almost no acceleration with an extra energy and area penalty. Hence, it seems reasonable to point the two-core solution as a good balance between acceleration, energy consumption and area.

However, the design target was to process at 30 fps, and the two-core solution falls 3 fps -i.e. 10%- short of this performance goal. Hence, the four-core solution is the most suitable. Using four cores, the final hardware accelerated solution processes over 32 fps of 1600×1200 pixel images, thereby achieving a real-time embedded solution to the problem. The acceleration from the original solution to the final software optimized and hardware accelerated solution is 53×, while the acceleration from the optimized software solution is 16×. The XC4VFX12 FPGA -which is the smallest Virtex4 FPGA- is 92% occupied, as compared with the original 64% for only software solution. On the other hand, the use of four cores represents an estimated 92% energy saving from the software solution, that is, 13× less energy consumption to process each frame. This embedded solution takes 30.7 ms to process an image, while the most optimized software solution in a Corei5 (2 cores@2.67GHz) takes 30.4 ms. This means that the embedded solution achieved by following the proposed methodology runs with a comparable speed as to the method implementation on an up-to-date general purpose processor, but it is smaller, cheaper, and demands less power and energy.

7 Conclusions

In this work, we proposed a methodology to achieve real-time embedded solutions using hardware acceleration, but with development times similar to software projects. This methodology applies to the growing field of processor-centric embedded systems with hardware acceleration in FPGA-based chips. The methodology is applied to a novel algorithm for multiple robot localization in global vision systems, demonstrating its usefulness for embedded real-time image processing applications.

The methodology helps to reduce design effort by raising the abstraction level while not imposing the need for engineers to learn new languages and tools. Taking advantage of the processor centric approach, the whole system is designed using well established high level modeling techniques, languages and tools from the software domain. In other words, it is an OOP design approach expressed in UML and implemented in C++ using multithreaded programming. The methodology also helps to reduce software coding effort since the C++ implementation provides not only a golden reference model, but may also be used as part of the final embedded software. Hardware coding, traditionally the most time-consuming and error-prone stage of hardware-accelerated applications, is simplified. The key to reducing hardware coding effort is to join a good OOP design implemented in C++, which allows engineers to precisely find the methods that need to be accelerated by hardware, with automatic tools or guidelines to translate the selected C++ methods to HDL.

A simple and robust algorithm for multiple robot localization in global vision systems is also presented.

The algorithm was specifically developed to work reliably 24/7 and to detect the robot's positions and headings even in the presence of partial occlusions and varying lighting conditions. To achieve a real-time embedded solution able to process over 30 fps, we applied the methodology, and performed software optimizations, used hardware acceleration, and extracted parallelism by including multiple ipcores in a multithreaded programming environment. The final embedded solution processes 1600×1200 pixel images at 32 fps, uses four hardware acceleration cores, occupies 92% of the XC4VFX12 FPGA and consumes approximately $17\mu J$ of energy per frame. This represents a $16\times$ acceleration with respect to the most optimized software solution, with a 43% increase in area but a 92% energy saving.

Acknowledgements Xilinx Design Suite was donated by Xilinx University Program. This work has been partially supported by Czech project No. 7AMB12AR022, by EU within ICT - 216240 and Argentinian projects MINCyT RC/11/20, UBACyT 200158 and PICT-2009-0041.

References

1. Altera (2011) Altera Introduces SoC FPGAs: Integrating ARM Processor System and FPGA into 28-nm Single-Chip Solution. URL http://www.altera.com/corporate/news_room/releases/2011/products/nr-soc-fpga.html
2. Arpinen T, Salminen E, Hmlinen TD, Hnnikinen M (2012) MARTE profile extension for modeling dynamic power management of embedded systems. Journal of Systems Architecture 58(5):209 – 219
3. Bailey B, Martin G (2010) ESL Models and their Application. Electronic System Level Design and Verification in Practice. Springer
4. Blanza D, Holland C (2012) Embedded Market Survey. EETimes and Embedded URL http://seminar2.techonline.com/~additionalresources/esd_apr2012/ubme_embeddedmarket2012_full.pdf
5. Bouguet JY (2008) Camera calibration toolbox for Matlab. URL http://www.vision.caltech.edu/bouguetj/calib_doc/.
6. Brezak M, Petrović I, Ivanjko E (2008) Robust and accurate global vision system for real time tracking of multiple mobile robots. Robotics and Autonomous Systems 56(3):213–230
7. Bruce J, Veloso M (2003) Fast and accurate vision-based pattern detection and identification. In: IEEE International Conference on Robotics and Automation - ICRA, IEEE, pp 1277 – 1282
8. ESA (2011) European Space Agency VHDL. URL http://www.esa.int/TEC/Microelectronics/SEMS7EV681F_0.html
9. Gaisler J (2004) A structured VHDL design method. In: Fault-tolerant Microprocessors for Space Applications, Gaisler Research, pp 41–50, URL http://www.gaisler.com/doc/vhdl2proc.pdf
10. Gaisler J (2011) A structured VHDL Design Method. URL http://www.gaisler.com/doc/structdes.pdf
11. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
12. Gunay N, Dadios E (2011) A robust and accurate color-based global vision recognition of highly dynamic objects in real time. In: 8th Asian Control Conference (ASCC), IEEE, pp 90 –95
13. Gupta S, Dutt N, Gupta R, Nicolau A (2003) SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In: 16th Intl. Conf. on VLSI Design, IEEE, pp 461–467
14. Happe M, Lubbers E, Platzner M (2011) A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking. Journal of Real-Time Image Processing pp 1–16
15. Jacquard (2011) ROCCC 2.0. URL www.jacquardcomputing.com/roccc/
16. Keskin O, Uyar E (2009) A framework for multi robot guidance control. In: Holonic and Multi-Agent Systems for Manufacturing, Lecture Notes in Computer Science, vol 5696, Springer Berlin Heidelberg, pp 315–323
17. Klančar G, Kristan M, Kovačič S, Orqueda O (2004) Robust and efficient vision system for group of cooperating mobile robots with application to soccer robots. ISA Transactions 43(3):329 – 342
18. Klančar G, Matko D, Blažič S (2009) Wheeled mobile robots control in a linear platoon. Journal of Intelligent and Robotic Systems 54:709–731
19. Kulich M, Chudoba J, Kosnar K, Krajnik T, Faigl J, Preucil L (2013) SyRoTek-Distance Teaching of Mobile Robotics. IEEE Transactions on Education 56(1):18 –23
20. Mentor-Graphics (2011) CatapultC - www.mentor.com/esl/catapult. URL www.mentor.com/esl/catapult/overview
21. Michael N, Mellinger D, Lindsey Q, Kumar V (2010) The GRASP multiple micro UAV testbed . IEEE Robotics and Automation Magazine
22. Mischkalla F, He D, Mueller W (2010) Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2010, pp 1201 –1206
23. Mueller W, Rosti A, Bocchio S, Riccobene E, Scandurra P, Dehaene W, Vanderperren Y, Ku L (2006) UML for ESL Design - Basic Principles, Tools, and Applications. In: IEEE/ACM International Conference on Computer Aided Design, pp 73–80
24. Nallatech (2011) DIME-C. URL www.nallatech.com/Development-Tools/dime-c.html

25. NVIDIA (2012) CUDA: Parallel Programming. Www.nvidia.com
26. OMG (2006) UML Profile for System on a Chip (SoC) Version 1.0.1 - formal/06-08-01. URL www.omg.org/spec/SoCP/1.0.1/
27. OMG (2010) Systems Modeling Language (SysML) Version 1.2 - formal/2010-06-01. URL www.omg.org/spec/SysML/1.2/
28. OMG (2011) UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems Version 1.1 - formal/2011-06-02. URL www.omg.org/spec/MARTE/1.1/
29. Paul J, Laika A, Claus C, Stechele W, El Sayed Auf A, Maehle E (2012) Real-time motion detection based on SW/HW-codesign for walking rescue robots. Journal of Real-Time Image Processing pp 1–16
30. Pedre S, Krajní k T, Todorovich E, Borensztejn P (2012) Hardware/Software Co-design for Real Time Embedded Image Processing: A Case Study. In: Alvarez L, Mejail M, Gomez L, Jacobo J (eds) Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications, Lecture Notes in Computer Science, vol 7441, Springer Berlin Heidelberg, pp 599–606
31. Pedre S, Krajník T, Todorovich E, Borensztejn P (2012) A co-design methodology for processor-centric embedded systems with hardware acceleration using FPGA. In: IEEE 8th Southern Programmable Logic Conference, IEEE, Brazil, pp 7–14
32. Quadri IR, Gamati A, Boulet P, Meftali S, Dekeyser JL (2012) Expressing embedded systems configurations at high abstraction levels with UML MARTE profile: Advantages, limitations and alternatives. Journal of Systems Architecture 58(5):178 – 194
33. Rao R, Taylor C, Kumar V (2006) Experiments in Robot Control from Uncalibrated Overhead Imagery. In: Ang J MarceloH, Khatib O (eds) Experimental Robotics IX, Springer Tracts in Advanced Robotics, vol 21, Springer Berlin Heidelberg, pp 491–500
34. Riccobene E, Scandurra P (2009) Weaving executability into UML class models at PIM level. In: First European Workshop on Behaviour Modelling in Model Driven Architecture (BM-MDA), CTIT Workshop Proceedings Series, Enschede, The Netherlands, pp 10–28
35. Riccobene E, Scandurra P, Rosti A, Bocchio S (2005) A SoC design methodology involving a UML 2.0 profile for SystemC. In: Proceedings of the conference on Design, Automation and Test in Europe - Volume 2, IEEE Computer Society, Washington, DC, USA, DATE '05, pp 704–709
36. Riccobene E, Scandurra P, Rosti A, Bocchio S (2006) A model-driven design environment for embedded systems. In: Proceedings of the 43rd annual Design Automation Conference, ACM, New York, NY, USA, DAC '06, pp 915–918
37. Rodriguez-Gomez R, Fernandez-Sanchez E, Diaz J, Ros E (2012) Codebook hardware implementation on FPGA for background subtraction. Journal of Real-Time Image Processing pp 1–15
38. Santarini M (2011) Zynq-7000 EPP sets stage for new era of innovations. Xcell Journal 75:8–13
39. Silva-Filho et al (2011) An ESL Approach for Energy Consumption Analysis of Cache Memories in SoC Platforms. International Journal of Reconfigurable Computing 2011:1–12
40. Sparx Systems (2011) Visual Modelling Platform. URL http://www.sparxsystems.com/products/ea/
41. Steggles P, Gschwind S (2005) The UBISENSE smart space platform. In: Third International Conference on Pervasive Computing
42. Technology BD (2010) The AutoESL AutoPilot High-Level Synthesis Tool. Tech. rep.
43. Thrun S, Burgard W, Fox D (2005) Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). The MIT Press
44. Vidal J, de Lamotte F, Gogniat G, Soulard P, Diguet JP (2009) A co-design approach for embedded system modeling and code generation with UML and MARTE. In: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, DATE '09, pp 226–231
45. Virginia et al (2007) An empirical comparison of ANSI-C to VHDL compilers : SPARK, RORCC and DWARV. In: Anual Workshop on Circuits Systems and Signal Processing ProRISC, pp 388–394
46. Xilinx (2011) Platform Studio and the Embedded Development Kit (EDK). URL http://www.xilinx.com/tools/platform.htm
47. Xilinx (2011) Xilinx Introduces Zynq-7000 Family, Industry's First Extensible Processing Platform. URL http://press.xilinx.com/index.php?s=34135&item=18
48. Xilinx (2012) Vivado Desing Suite. URL http://www.xilinx.com/products/design-tools/vivado/index.htm
49. Yankova Y, Kuzmanov G, Bertels K, Gaydadjiev G, Lu Y, Vassiliadis S (2007) DWARV : Delftworkbench automated reconfigurable VHDL generator. In: Intl Conf on Field Programmable Logic and Applications, IEEE, pp 697–701
50. Zhang Z (1999) Flexible camera calibration by viewing a plane from unknown orientations. In: Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on, vol 1, pp 666–673