Des Autom Embed Syst (2006) 10:127–155
DOI 10.1007/s10617-006-0069-7

Design, synthesis and verification of a smart imaging core using SystemC

Wido Kruijtzer · Victor Reyes · Winfried Gehrke

Received: 3 February 2006 / Revised: 3 August 2006 / Accepted: 4 August 2006

Abstract In this paper the development of a smart imaging core following a SystemC-based design flow is presented. The smart imaging core integrates an ARM processor and two specific hardware blocks for image processing: a smart imaging coprocessor and a motion estimation coprocessor. A SystemC-based design flow is applied, comprising the design, synthesis and verification and synthesis of the two coprocessors, as well as the development and integration of the embedded software on the smart imaging core. The two coprocessors are successfully modeled and refined from C/C++-based algorithmic descriptions down to architecture reference models using SystemC and TLM concepts. For the RTL implementation of the coprocessor hardware high-level synthesis tools are used. The applied SystemC-based design flow enabled the iterative refinement of the architecture towards an optimal RTL implementation. Furthermore, the use of SystemC TLM supports the integration of fast functional models of the coprocessors on a virtual prototype platform of the target architecture. This virtual prototype is beneficially used during the embedded software development phase.

Keywords System level design · SystemC · High-level synthesis · System simulation · Image processing

W. Kruijtzer (✉)
Philips Research, High Tech Campus 31, 5656 AE Eindhoven, The Netherlands
Present address: Philips Semiconductors, High Tech Campus 46-2.84, 5656 AE Eindhoven, The Netherlands
e-mail: wido.kruijtzer@philips.com

V. Reyes
University of Las Palmas GC, Campus de Tafira, E35017 Las Palmas, Spain
Present address: Philips Research, High Tech Campus 31, 5656 AE Eindhoven, The Netherlands

W. Gehrke
Philips Semiconductors GmbH, Georg-Heyken-Straße 1, 21147 Hamburg, Germany

1 Introduction

The increasing integration capabilities of technology allow to enhance video compression cores with smart imaging functionality and to embed these cores even into low-cost camera devices. This is the starting point for new smart imaging applications that are able to analyze the content of images and video sequences enabling new consumer applications that are targeting various domains such as mobile and automotive. Cameras embedded in mobile phones are now becoming a commodity supporting applications like capturing and transmission of still images as well as video clips (Multimedia Messaging Services). With the increase of network bandwidth (e.g. 3G UMTS) real time mobile video links will become feasible, enabling new applications like mobile video telephony and video chat. The ease of use of these applications is of high importance as this is expected to be a crucial requirement for market acceptance of such new services. Thereby not only quality issues like frame and image stabilization are to be focused but also the user comfort. The automatic detection and tracking of the user's head is such an example, which helps to keep one's face in view of the camera during a mobile video telephone conference.

In the automotive domain, cars are equipped with more and more electronic systems that support the driver to avoid accidents. These systems are used to analyze complex driving situations and provide important and reliable information to the driver. Some of these driving aids use radars like the automatic cruise control, but driving assistance systems using cameras also appear since they have less interference with its surroundings. Furthermore, techniques like radar lack the possibility to classify detected objects. Here two examples are low speed obstacle detection [2], which deals with the detection of vehicles in a certain speed range, and pedestrian detection [3], which concerns the detection of pedestrians and an impact prediction in order to reduce the injuries of the pedestrians hit by a car.

The design and verification of complex applications such as the ones described above requires new advances in conventional design methodologies. Often SoC design is as a sequential approach where hardware (HW) development at the Register Transfer level (RTL) level precedes software (SW) development at the C and/or assembly code level. Design and verification methods based on RTL level limits the exploration of different design alternatives due to the enormous amount of details the designer has to handle and its slow simulations (in the order of few hundred cycles per second) [20]. Moreover, due to the exponential growth of embedded SW in current SoCs such sequential methodologies, in which the software cannot be developed until the HW platform is available, are not appropriate [21]. SystemC [19] and Transaction Level Modeling (TLM) [18] are becoming the main forces to overcome the limitations of conventional SoC design methodologies [22]. On the one hand, SystemC provides an executable specification of the system behavior, which can be used as a replacement of the ambiguous textual specifications. Such executable specification serves as a system-level test bench for the next steps in the design flow, which simplifies drastically the time and effort required for the verification. On the other hand, TLM increase significantly simulation speed, while still offering enough accuracy for exploring and validating implementation alternatives at the higher levels of abstraction [23]. Besides this increase in speed, TLM reduces the amount of detail the designer must handle, resulting in less modeling effort. Hence, TLM SystemC models can be built well in advance before the time-consuming RTL code development starts. This allows for embedded SW being developed in parallel with the HW. As an example of the increasing adoption of SystemC and TLM, companies such as *Texas Instruments* and *STMicroelectronics* have used a SystemC-based design methodology for the design and development of their *OMAP* [25] and *Nomadik* platforms, respectively. Furthermore, SystemC-based design flows are also able to close the gap from specification

to implementation by means of high-level synthesis tools, such as [14, 17, 24] or [28]. These tools can generate high-quality RTL from behavioral or even TLM SystemC descriptions allowing the rapid exploration of several implementation options. As an example, *Toshiba* has recently announced the successfully completion of an advanced multimedia H.264 design using its R-Cube methodology [29]. Moreover, [26] and [27] show positive examples of using SystemC behavioral synthesis for complex system development such as a PCI bus interface and an MPEG2 encoder, respectively.

The aim of the work described in this paper is to develop a smart imaging core that can be embedded in a camera applying such a SystemC-based design methodology. This core should be low-cost, low-power and suitable of supporting the above mentioned automotive and mobile communication applications. The resulting architecture integrates two coprocessors that are designed using high-level synthesis tools taking the C-language as a starting point. Furthermore, the verification of the system architecture and the HW/SW integration is performed using abstract transaction level SystemC models.

The remainder of this article is organized as follows. In Section 2 the structure of the smart imaging algorithms and their mapping on an optimal smart imaging architecture is discussed. Section 3 describes the design flow used for the development of the smart imaging architecture. A detailed description of the coprocessors architecture and their design, validation and synthesis flows is given in Section 4. In Section 5 the virtual prototype used to port the applications onto the smart imaging core, as well as an FPGA based prototype are discussed. Section 6 finally presents some conclusions.

2 Smart imaging core

2.1 Algorithms description

The processing chain of a smart imaging application comprises several algorithms that can be clustered based on their properties with respect to data access and inherent parallelism. Throughout this paper algorithms are classified using the following three classes: low-level algorithms (LLAs), medium-level, algorithms (MLAs) and high-level algorithms (HLAs). A typical smart imaging application structure is depicted in Fig. 1.

The input as well as the output of LLAs is video data, i.e., pixels. The processing is typically performed on image segments that consist of a relatively small number of neighboring pixels,

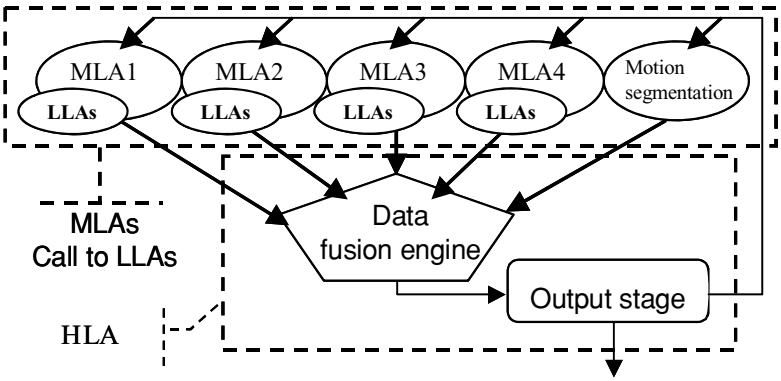

Fig. 1 Smart imaging application structure

which can be processed independently. As LLAs are processed on pixel-level, the amount of data to be processed and the associated computational performance required is relatively high. Examples of LLAs are linear kernel filtering, thresholding or morphological operations. The LLAs used for the smart imaging algorithms are provided through the CAMELLIA Image Processing Library (C-IPL) for which the source code can be found at *Sourceforge* [1].

The class of HLAs contains control tasks that deal with abstract semantic information extracted from a video scene. HLAs make the fusion between several MLAs, which individually are not sophisticated enough to yield a good result and also comprises the output stage that produces the result of the system. The amount of data to be processed and the computational performance required for a real-time implementation of these tasks is relatively low.

As a link between low-level pixel-based processing and the HLAs, the class of MLAs can be defined. Algorithms of this class are typically used for an abstraction of the scene contents. Their input data are mostly pixel data, whereas output data represent abstracted image data. MLAs typically rely on LLAs to perform their low-level operations. A typical example of MLA is labeling with feature extraction of the labeled objects.

As an example, the automotive application Low Speed Obstacle Detection [2] (LSOD) is composed of an HLA that combines the output of several vehicle detectors (MLAs) in order to obtain an exact detection and localization of vehicles. The MLAs used in LSOD are: shadow detection, edge detection, rear lights detection, symmetry detection and motion segmentation.

2.2 Algorithms analysis

Our goal is to bring smart imaging into the consumer market, in which high performance general-purpose processors are not accepted as a cost efficient solution. Typically the architectures in this domain contain a rather modest general-purpose processor (e.g ARM9) running at a few hundred MHz.

HLAs are typically sequential and a parallel execution of parts of these algorithms is in general not possible. They are associated with a complex irregular data-dependent control flow. Therefore, HLAs nicely fit on such a general-purpose processor as they are control dominated and their computational load is limited. LLAs however are associated with a high amount of inherent parallelism and relatively simple operations. Due to the high throughput rate required for the execution of LLAs, efficiency can be significantly improved by exploiting data-level parallelism. This typically is not efficient on a general-purpose processor. As an example Fig. 2 shows what processor frequency in MHz is needed to execute some LLAs on video data at 25 frames per second with the resolution of 352 × 288 pixels per frame.

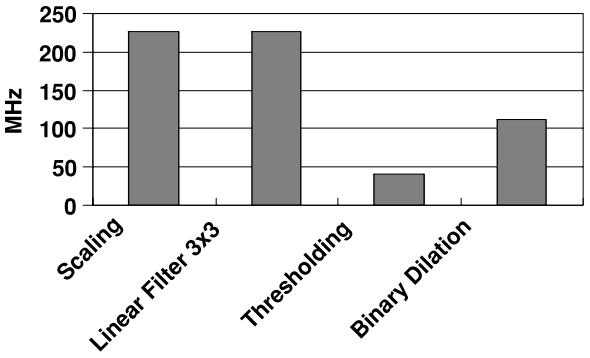

Fig. 2 ARM9 frequencies in MHz required for processing some LLAs

As can be seen both scaling and 3×3 linear kernel filtering need a 225 Mhz ARM9. The data is based on an ARM9 processor running un-optimized C-code, overhead for address calculations and memory latency are not considered. With optimized C-code we might achieve each single LLA to execute for the specified frame size and rate at a 100 Mhz processor.

Still the smart imaging application would not fit as each MLA uses several LLAs and typically a couple of MLAs are used in each application (e.g. LSOD). A further reduction of the computational load can be achieved by trying to process only the interesting part instead of the whole image in order to detect and track a certain object. Such interesting part is denoted as Region Of Interest (ROI). Only the ROIs are investigated by the LLAs. Even though the ROI based approach saves a lot of computations, it is impossible to make any assumptions about the ROI size and the amount of ROIs. Furthermore future applications may request a higher frame rate and higher resolution than the ones used in the example of Fig. 2. Clearly some form of hardware acceleration that exploits the inherent parallelism of the LLAs is needed.

2.3 Architecture and mapping

The classical coprocessor architecture is selected for the realization of the system. The co-processors execute a specific algorithm or a class of algorithms with similar computational requirements faster than a general-purpose programmable architecture resulting in a significantly higher ratio of computational performance and system cost compared to other architectural approaches.

The core architecture is depicted in Fig. 3. The underlying architecture template is based on an existing ARM9 architecture used in the mobile communications domain. The HLA and the more control-oriented part of the MLAs are combined together, which fits well to be mapped onto the embedded ARM9. All the LLAs are combined onto a single smart imaging coprocessor (SI). Likewise, the pixel processing part of the motion segmentation MLA is distinguished as an independent task, which is mapped onto a motion estimation coprocessor (ME). With this mapping approach flexibility can be achieved for the high-level

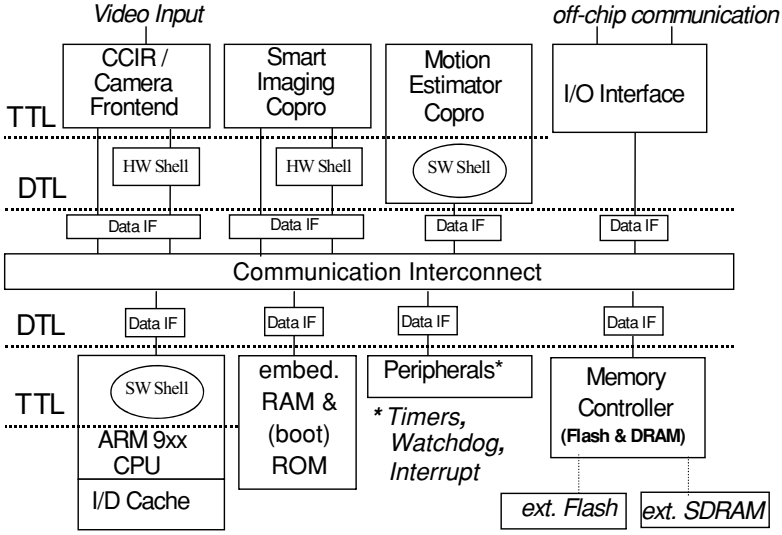

Fig. 3 Architecture of the smart imaging core

processing by SW changes of the CPU program code. In order to allow for a flexibility of the implementation of LLAs and parts of MLAs mapped onto coprocessors, architectural efficiency has to be taken into account. This is discussed in more detail in Section 4, which presents the coprocessor architectures. Compared to [4] and [5], this core is low-cost, low-power and targets a wide application area unlike [6] that is optimized for a single application. Furthermore our solution has built-in logic to work on image segments (i.e ROIs) instead of complete images, something not present in these other solutions.

Devices communicate with each other through the DTL protocol which is a Philips proprietary device level interface very similar to AXI [30], whereas tasks in the system communicate and synchronize with each other using the Task Transaction Level (TTL) interface and corresponding primitives [8]. On the one hand, application developers can use TTL to build executable specifications. On the other hand, TTL provides a platform interface for implementing applications as communicating HW/SW tasks on a platform infrastructure. TTL specifies services for inter-task communication, multitasking and task graph reconfiguration. These services can be accessed via the TTL interface, which hides the implementation details of the services from the tasks. Rather than offering a low-level interface and implementing e.g. synchronization as part of all the tasks, TTL factors out such generic services from the tasks to implement them as part of the platform infrastructure. In this way TTL effectively separates computation and communication aspects, which facilitates the construction of (streaming) IPs and makes them more reusable as they contain less implementation details. For each multiprocessor architecture the services can be implemented in a way that is optimal for that architecture.

TTL tasks can execute concurrently and connect with each other through unidirectional *channels*. A task is connected to such a channel via a *port* and communicates with other tasks by calling TTL interface functions on their ports. TTL offers seven interface types from which designers can use the most appropriate one for their application and platform. These types differ in the level of detail of the underlying platform that is exposed towards the programmer and in their potential implementation efficiency on different platforms. All interface types however are based on the same logical model, which enables interoperability across interface types. In [8] a full coverage of all seven interface types can be found. In the SI architecture the interface type RB (Remote, Blocking) is used that offers separate functions for synchronization and data transfer. Below the RB functions are shown in Fig. 4.

The availability of room or data in a channel can be checked explicitly by means of a blocking *acquire* function and can be signaled by means of a *release* function. The *acquire* and *release* functions synchronize for vectors of `count` tokens at a time. Data accesses can be performed on acquired room with the `store` function, which copies a vector of `size` values to the acquired empty tokens. The *store* function can perform out-of-order accesses on the acquired empty tokens using a relative reference `offset`.

The TTL interface is provided both as a hardware interface and as a software API, thereby enabling the integration of both hardware and software IP. Figure 3 shows how the TTL interface manifests itself in the architecture of the smart imaging core. In the bottom part of

Producer

```
reAcquireRoom (port, count)
store (port, offset, vector, size)
releaseData (port, count)
```

Consumer

```
reAcquireData (port, count)
load (port, offset, vector, size)
releaseRoom (port, count)
```

Fig. 4 TTL interface type RB

Fig. 3 the TTL interface is implemented as an API of a software shell executing on the ARM CPU. Software tasks executing on the CPU can access the platform services via the API. In the upper part of Fig. 3 the TTL interface for integrating the SI coprocessor is available as a hardware interface. A hardware shell implements the platform services on top of the lower DTL interconnect.

3 Design flow

To validate the correctness (and quality) of the applications executed in the targeted system architecture an FPGA based demonstrator or prototype is built. This validation comprises the verification of both the implemented coprocessors (functionality and performance), as well as the software optimizations required for its execution in an embedded system. The global design methodology applied for building the demonstrator is depicted in Fig. 5. The starting point is the smart imaging applications, which are developed on a standard PC using C++. By means of profiling of the applications and detailed analysis of the LLAs the HW/SW partitioning is derived as described in Section 2. The main part of the design flow is the SystemC based development of both coprocessors and the embedded SW.

The design approach applied for the coprocessors is a C++/SystemC based successive refinement of the architecture [19]. The result of this refinement is both a cycle accurate (CA) and programmers view (PV) model, according to the terminology defined by the OSCI

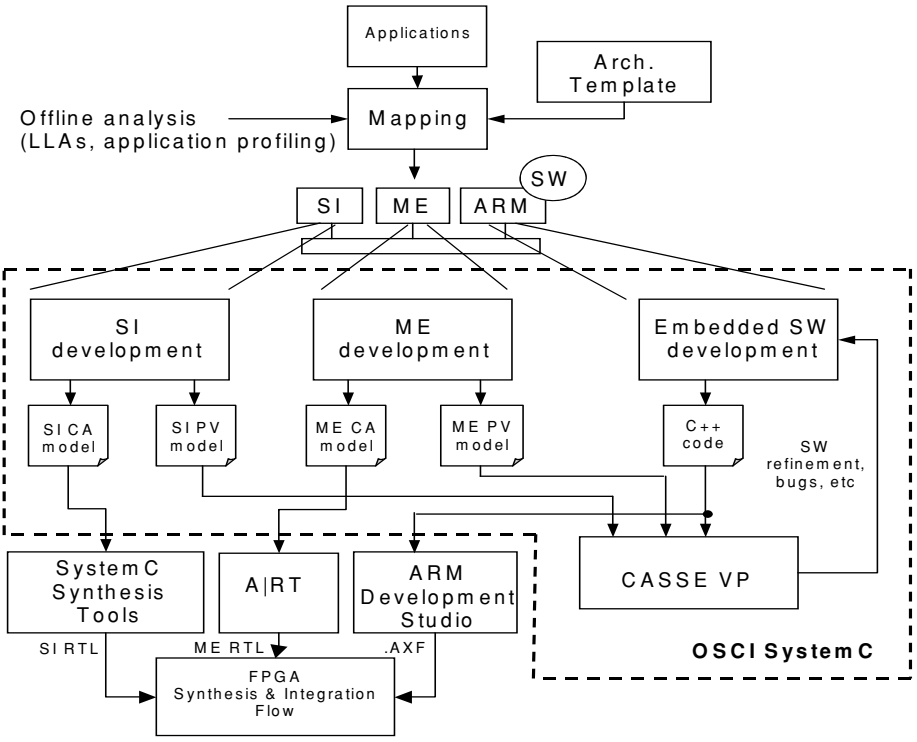

Fig. 5 Design flow

TLM standard [18]. The CA models can directly be used in the C/SystemC based synthesis tools CoCentric [17] and A|RT [14] resulting in an RTL description of the coprocessors. The PV models are integrated in a virtual prototype (VP) to enable early SW development. This approach allows verifying the HW/SW integration in an early stage before the FPGA based demonstrator is available. Furthermore the ME and SI can be intensively verified together with the software before pursuing their actual FPGA implementation by using the VP as a system test bench. The VP is built using CASSE [7], which models architectural elements at the higher abstraction level using transaction-level modeling techniques. More details regarding the coprocessor development are described in Section 4. The embedded SW development using the VP is described in Section 5.

The applied design flow results in validation of the correct functioning of the applications at three levels. During the execution of the applications on the PC intermediate results for each MLA composing the application, as well as high-level information regarding the expected output, are gathered and dumped into files. Both textual files (e.g. objects position, detection probability, etc.) and pixel images are generated at certain execution points of the application, which are used as golden reference for a comparative check later on.

The next level of validation is based on the usage of the VP. Thereby, application functionality is validated by means of comparing the results (checkpoints) produced by the original application with the results produced by the VP.

Finally, the third level of validation is based in the FPGA based demonstrator. Thanks to the VP, software can be directly integrated in the demonstrator. However, embedded compilation of the SW has to be tested in order to check possible inconsistencies in the embedded execution. We ensured that both the virtual and the FPGA based prototype are composed of the same elements, which respond in the same address range. Final verification is carried out by comparing the application checkpoints when running in the FPGA with the results obtained with the VP and with the reference applications.

4 Coprocessors development

This section will give an overview of the development process of both coprocessors. First the coprocessors internal architecture is explained followed by a detailed description of the applied SystemC based design and synthesis flow. Finally the use of SystemC in the verification of the SI coprocessor is explained.

4.1 Coprocessor architectures

4.1.1 Smart imaging coprocessor

As smart imaging applications have a clear need for acceleration of basic image processing tasks, the architecture of the smart imaging coprocessor (SI) is adapted for the efficient execution of this algorithmic class. The requirements of smart imaging applications are extracted by the analysis of sample applications from different fields of smart imaging applications. The SI accelerates most of the functions of the C-IPL including arithmetic and morphological operations, linear kernel filtering, horizontal and vertical summing, scaling, lookup-table based pixel mapping, histogram, moments and min-max computation.

As the coprocessor is used to accelerate the execution of these LLAs, one option is to implement each LLA as separate dedicated HW component. In this case the combination of all LLA functions would result in the final coprocessor. One of the major disadvantages of

this approach is the poor HW utilization: Only one LLA (i.e. one dedicated HW component) would be active at a time, while the other components would be idle. Another disadvantage is that the coprocessor's functionality would be limited to the functionality of the implemented dedicated components.

In order to avoid these disadvantages the coprocessor is implemented as a so-called macro-programmable device. This approach increases the architectural flexibility while supporting sharing of HW resources, like arithmetic components and embedded storage elements, among different LLAs. At the same time the overhead associated with general-purpose micro-programmable architectures is limited as discussed later in this section.

Most of the envisaged algorithms can be implemented by exploiting data parallelism based on concurrent processing of neighboring pixels. Thus, a classical SIMD architecture has been chosen for the data path of the SI. This approach has been successfully integrated as ISA extensions of general-purpose CPUs for more than a decade. An example of this approach is the MMX instruction set extension [11]. In order to improve the performance of the SI further, the data path is composed of several arithmetic units. Per clock cycle up to 6 arithmetic operations can be executed on each pixel. The data path is organized as a vertical arithmetic pipeline, i.e., the arithmetic operations are executed in a fixed order. The major advantage of this approach is the avoidance of register files with multiple ports, which can become very costly with respect to silicon area. In order to achieve reasonable clock rates, the arithmetic unit contains several pipeline stages. Typically, this approach would lead to a degradation of performance for algorithms that contain data-dependent decisions due to pipeline hazards. The implementation of the SI data path tries to minimize the impact of hazards by moving data dependent decisions into the data path itself. As a simple example of this approach thresholding is considered: In case of a general purpose CPU this algorithm requires a compare operation combined with a subsequent branch instruction that depends on the result of the compare operation. If the compare operation is associated with a significant latency caused by the pipeline depth of the arithmetic data path, the overall performance is degraded significantly. The SI coprocessor supports the comparison and subsequent selection of the result as an integral part of the arithmetic pipeline. Thus, the control flow is kept data independent and no hazards occur during the execution of this algorithm. The arithmetic units receive up to three input operands and create one output result per clock cycle. All operands and results are stored in local memories. Data processing and communication with external memory can be executed in parallel. This approach allows pre-loading of a certain image segment while the previous one is processed by the coprocessor.

For very specific functions the architecture allows the introduction of function specific units. An example of such a unit is the RLE (Run-Length-Encoding) unit. This unit transforms a binary image into a list of so-called runs, indicating the number of subsequent zeros or ones in a line of the input picture.

Another important architectural aspect is the implementation of coprocessor control, i.e., control of the execution of arithmetic functionality. As flexibility is regarded as a very important topic in order to cope with moderate changes of the application even if the design of a coprocessor has been finished, a programmable implementation should be preferred. On the other hand, competitiveness for a specific application requires an area efficient system solution. Therefore a macro-programmable coprocessor control approach is adopted: The data path is designed to process microinstructions, which are issued by the micro control unit. An additional macro control unit is used to control special sequences of microinstructions. The control is therefore split into two hierarchy levels, the micro control unit and the macro control unit. At the lower level of this hierarchy the arithmetic units and the memory accesses are controlled by a VLIW approach that supports a high degree of flexibility. In order

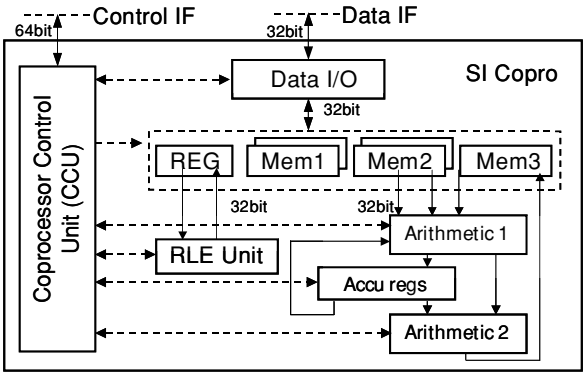

Fig. 6 Smart imaging coprocessor

to avoid the drawbacks of the classical memory- and bandwidth-hungry VLIW architectures, the macroinstruction level is introduced in the SI. This level is used to translate mighty so-called macroinstructions into a sequence of VLIW microinstructions. Several classes of macroinstructions are being used: I/O instructions control the data traffic with system memory and allow initiating a transfer of arbitrarily sized 2-dimensional blocks of data with a single instruction. Execution instructions typically execute a basic image-processing algorithm on an image segment previously loaded into local memory. Configuration instructions are used to set pseudo-static data, like image base addresses, segment information and data like filter coefficients. The described hierarchical control approach can be viewed as another important extension to the principle of the vector based SIMD programming model of current general-purpose CPUs. The adaptation of the data-path's arithmetic and the chosen hierarchical control strategy allows choosing a well-suited trade-off between flexibility and area efficiency for the envisaged application domain. The resulting coprocessor architecture is depicted in Fig 6. A more detailed overview on the SI coprocessor is described in [9, 10].

4.1.2 Motion estimation coprocessor

Motion estimation is one of the time-critical tasks in the application algorithms. Its computational and addressing complexity is huge due to the typical sum-of-absolute-difference operations performed at pixel level, the required sub-pixel (quarter-pixel) accuracy, the number of motion vector candidates, the number of passes (scans) per frame, frame rate, etc. Therefore, one of the decisions at system level is to map the motion estimation task onto a coprocessor. The block-based Motion Estimation coprocessor (ME) accelerates the motion segmentation MLA. The goal of motion segmentation is to identify moving objects from their motion. The motion segmentation is integrated tightly with motion estimation through a loop in which candidates for motion estimation are generated based on the result of segmentation. First a motion model for each block is calculated after which blocks are grouped into segments that have a similar motion model and low sum-of-absolute-differences using a *Breadth First Search* algorithm.

Currently, two contrasting implementations are often considered for such high performance video processing: ASICs and DSPs. ASICs optimally meet performance and power requirements, but lack flexibility. DSPs are highly flexible, but have significant overhead in achieving the performance requirements for a low power budget. The ME is therefore designed as an Application Specific Instruction Processor (ASIP). ASIPs offer performance, power and area that are comparable to ASICs but are superior in terms of performance, power

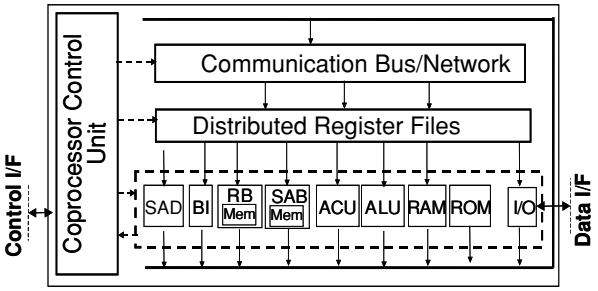

Fig. 7 Motion estimation coprocessor

and area compared to DSPs for applications in their domain. ASIPs, tuned to an application domain, can be based on any processor architecture template such as a VLIW architecture, or a vector processing architecture. It is interesting to note that the choice of the ASIP template architecture greatly depends on the characteristics of the application domain and the available tool flow. Among the available tool flows for ASIP design, namely A|RT [14], LISA [13] and CHESS [12], the A|RT-based tool flow is used. This tool flow uses a VLIW architecture template.

The VLIW architecture template of A|RT is composed of: standard function units, Application Specific Units (ASUs), a control unit and an interconnect structure of registers and multiplexers. The standard functional units include Arithmetic-Logic Units (ALUs), Address Calculation Units (ACUs), RAM and ROM. The ASUs are user defined units and typically are used to accelerate critical kernels of an algorithm. In our case the ASUs are tailored for accelerating the inner kernels of motion estimation. After analysis of the motion estimation algorithm, the following ASUs are defined (see Fig. 7): Search Area Buffer (SAB), Reference Block Buffer (RBB), Bi-linear Interpolation unit (BI) and a sum-of-absolute-difference unit (SAD). The motion estimation algorithm is block-based using 8-bit pixel blocks of 16×16 pixels. Therefore each ASU is designed for processing 16 pixels in parallel in order to ensure execution of the operations in the innermost loop of the motion estimation algorithm in one cycle for each pixel line of a 16×16 block. The ASUs are based on an earlier developed general ME template [15].

In order to have a predictable system design, the complete search area (from previous frame) is stored in the *Search Area Buffer* (SAB). By restricting the motion vector candidates to the search area, this approach results in improved performance and reduced power dissipation. The SAB memory is organized as 6 banks, each containing 32 pixel-lines of 32 bits (four 8-bit pixels). During a read operation, a number of banks are selected and the resulting bank outputs are concatenated and aligned to produce a single 16 pixel-line. One 16 pixel-line can be delivered every clock cycle. The *Reference Block Buffer* (RBB) is used to store the reference block from the current frame. The RBB is organized as four banks, each containing 16 pixel-lines of 32 bits. During a read operation, the four bank outputs are concatenated to deliver 16 pixels in parallel. The *Bi-linear Interpolation* unit (BI) is used for generating corresponding pixels for the SAD calculation in case sub-pixel accuracy of motion models is required. The BI is pixel line organized and it generates 16 interpolated pixels in one clock cycle. The *sum-of-absolute-differences* unit (SAD) is used to calculate the SAD of every candidate motion model. It compares a block within the current frame and the corresponding block within the previous frame shifted by the motion model candidates.

In contrast to the motion estimator described in [15] the ME calculates SAD values per 16×16 macro-blocks as a weighted sum of SADs from both luminance and chrominance pixel blocks. Each video component (Y, U, V) is calculated sequentially by the ME using

three loops and the set of ASUs described above. The resulting ME coprocessor is flexible within an application domain and can be programmed for different video applications while benefiting from the instruction-set that accelerates motion estimation functionality.

4.2 Coprocessors design and synthesis

4.2.1 Smart imaging coprocessor

The design approach chosen for the smart imaging coprocessor (SI) is based on successive refinement of the architecture by applying a C/SystemC based approach. An overview on the different abstraction levels used during this refinement is depicted in Fig. 8.

Algorithmic level (*AL*). The starting point for the design of the coprocessor has been the C-IPL library, which is written in plain C. This code describes only the algorithmic behavior of the LLAs and does not take any architectural aspects into account (See Fig. 9(a)).

Programmers view (*PV*). Based on the analysis of the C-IPL library the partitioning into common low-level arithmetic operations executed by the coprocessor and the control SW executed by the system CPU has been selected. In order to verify this initial partitioning, a model at the PV level (Fig. 9(b)) has been created. Since the functionality of the coprocessor is done on a pure functional description without taking cycle-based timing into account, this model can achieve a relatively high simulation speed that is close to the speed of the AL description. The resulting SI coprocessor at the PV level is used to develop the final application SW as explained in Section 5.

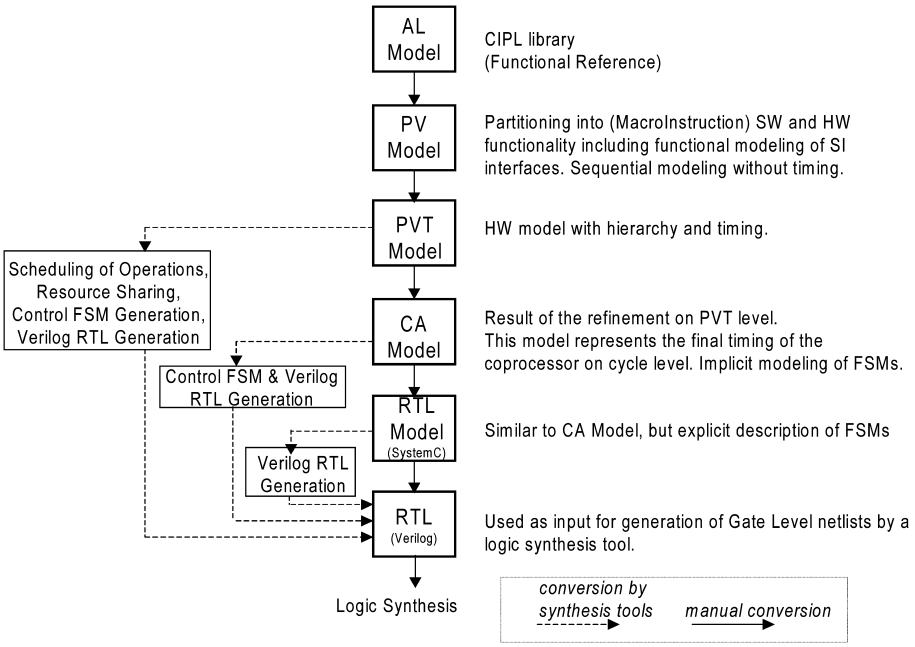

Fig. 8 Abstraction levels applied for the coprocessor modeling

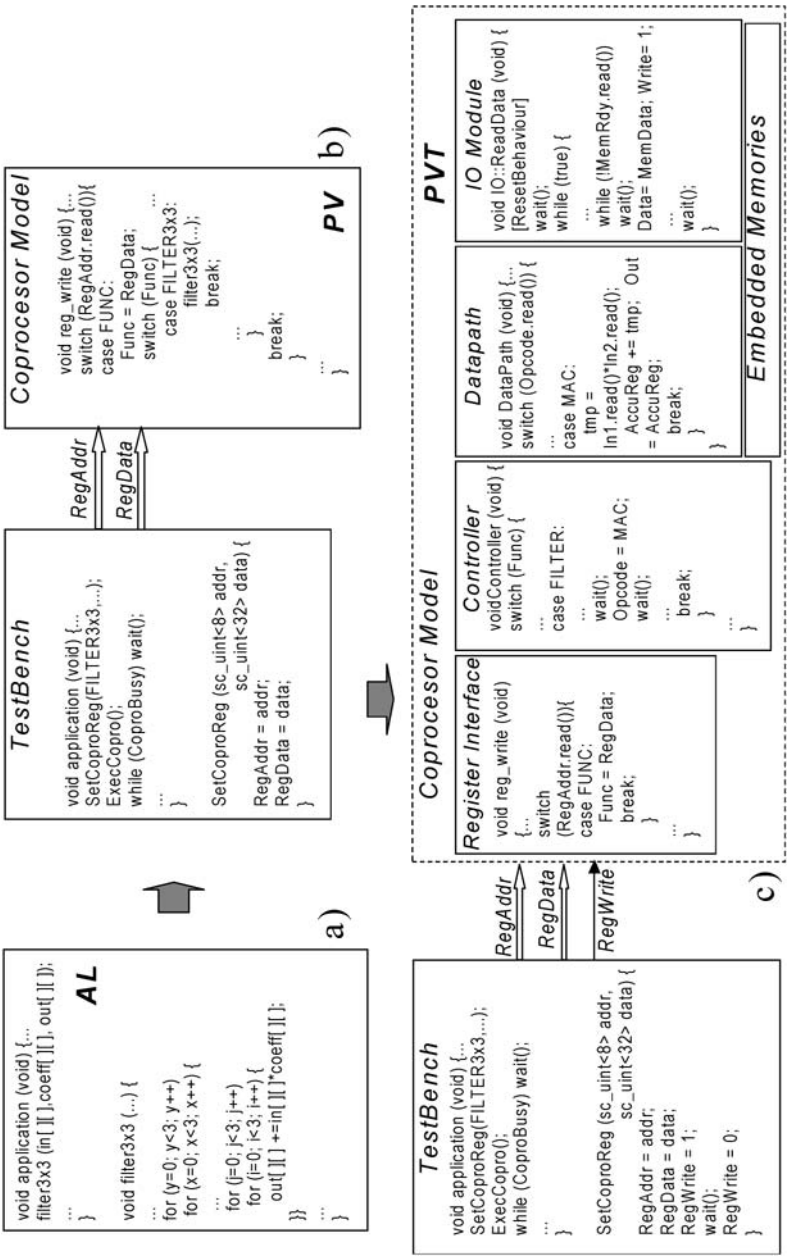

Fig. 9 Modeling of the SI coprocessor

Programmers view with timing (*PVT*). The PV model is refined reflecting the coprocessor's HW partitioning and the timing of internal and external SI interfaces. This model can be regarded as a PV model extended with timing information. The coprocessor PVT model (Fig. 9(c)) is composed of several sub-modules according the SI internal architecture as depicted in Fig. 6. This model reflects the internal partitioning of the coprocessor architecture and implements the bit-true and cycle-true behavior of the register interface as well as the IO interface, which is communicating with the shared system memory. Communication between the sub-modules of the coprocessor is explicitly implemented. The initial timing information used in this PVT model is based on experience of the designer.

Further optimizations of the coprocessor with respect to functionality of the sub-modules as well as the arithmetic performance of the coprocessor are carried out at the PVT level. It is obvious that such optimizations cannot be continued without taking the achievable clock frequency for the target semiconductor technology into account. Therefore, it is reasonable to perform initial logic synthesis runs as early as possible. The results of these synthesis runs give a very important feedback on potential timing bottlenecks of the design and have to be taken into account for the refinement of the cycle timing of the design.

As depicted in Fig. 8, it is possible to create a Verilog RTL description out of a PVT model by a behavioral synthesis tool. This RTL description could then be used as an input to a logic synthesis tool, which performs the mapping onto gates of the target semiconductors library and indicates the achievable clock frequency of the design. Using high abstraction levels as design entry limits the design effort and increase the productivity. Therefore we decided to introduce behavioral SystemC synthesis from a PVT description into the design flow of the SI. At the time the SI coprocessor is implemented a tool called *Cocentric SystemC Compiler* was still available from Synopsys Inc. This tool supported a SystemC-based design entry and allowed for behavioral synthesis (from PVT or CA level) as well as RTL synthesis. The behavioral synthesis option supported an automatic generation of memory structures, data path elements as well as the required control FSM for a specific design block. Moreover, the tool supported several useful features like operator and memory sharing or automated memory instantiation. In the meantime the tool is discontinued, but other tool suppliers, like Forte Design Systems, entered the SystemC behavioral synthesis arena with tools that offer even more functionality than Cocentric SystemC Compiler. As resource sharing, scheduling of operations as well as the associated insertion of pipeline stages is performed automatically by the behavioral synthesis tools, the PVT entry is very useful for dataflow-oriented designs, aiming at a dedicated implementation of core functionality of a specific application. These designs have typically weak constraints on the latency of the functionality. Thus, a behavioral synthesis tool has a high degree of freedom to schedule operations and data accesses, which lead to several alternative implementations with different data throughput, silicon area as well as power consumption.

The situation is slightly different for microinstruction-controlled designs like the SI coprocessor. In this case one constraint is to start the execution of one microinstruction in each clock cycle. In this case it is desired to control the behavior of the critical parts of the design, like shared embedded memory resources, on a cycle accurate level. Therefore, the SI model is refined by adding more accurate timing information, which finally led to a cycle-accurate model (CA).

Cycle-Accurate (*CA*). An iterative optimization process is applied to create the targeted CA representation of the coprocessor. A first iteration loop was mainly focusing on arithmetic performance and a second iteration loop was used to improve the final silicon area occupied by the design. The before mentioned SystemC synthesis tools are also able to use the CA

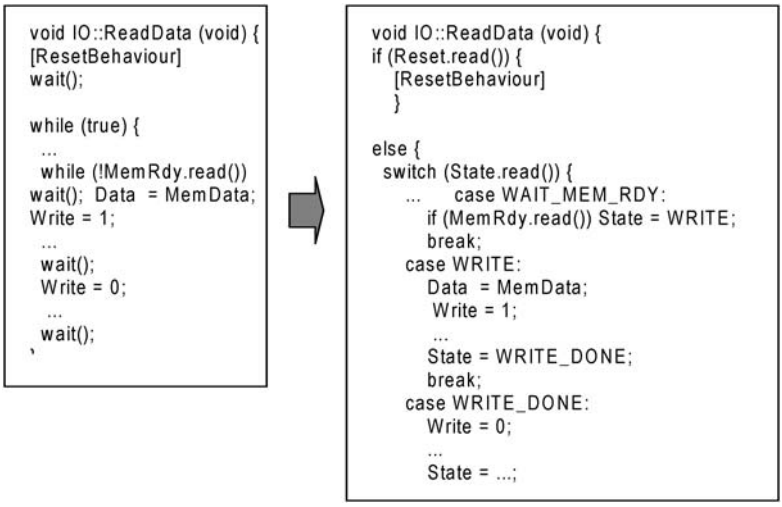

Fig. 10 Migration from a CA model to a SystemC-RTL description

description as entry point for behavioral synthesis. Such tools automatically generate the FSM control for the SI coprocessor.

SystemC-RTL. The implementation of RTL allows the designer to control the functionality on a lower level and thus it can be expected that the optimization with respect to timing and area can be done more easily than by constraining the behavioral synthesis process from CA level. Furthermore the benefit of automated FSM generation for blocks containing small global control functionality is rather limited as the step from the CA representation towards a SystemC-RTL model can be done relatively simple: Code that is placed between two subsequent wait() statements can be moved into one branch of a global state machine of this block. Moreover, a state variable has to be introduced which has to be assigned in every branch of the global state machine. Figure 10 illustrates the conversion from a cycle-accurate behavioral model into a RTL description based on a simple example.

Based on the experiences made throughout the design and implementation of the SI coprocessor the authors conclude that a SystemC based design flow is a promising approach. It supports a smooth iterative refinement of the architecture from a pure functional description of the algorithms down to RTL for logic synthesis. For example, the iterative design flow approach applied for the SI coprocessor enabled the implementation of a fully verified RTL description and has been achieved by spending an effort of roughly one man-year.

On the other hand, the simulation speed of a model reduces drastically with increased timing accuracy. This behavior can become a significant hurdle if a huge number of patterns have to be simulated during the iterative refinement of the architecture. However, it is not an issue for the design of the SI. As the coprocessor aims at the acceleration of LLAs that process small portions of an image, the number of patterns required for simulation can be kept small. Thus, the simulation time during the iterative refinement could be kept at a reasonable limit (see Section 4.3) Therefore, the refinement of the models from PVT to CA and later on to RTL are focused on the performance of the coprocessor and it is acceptable to disregard any optimizations aiming at an increase of simulation speed.

One of the major milestones for a wider acceptance of SystemC synthesis is the de-facto standardization of the SystemC subset supported for synthesis that is now being formalized

by the OSCI Synthesis Working Group. However, from a designer's point of view it is desirable to extend the defined subset by a standardization of pragmas or language extensions for steering the behavioral synthesis process. Moreover, for the implementation of micro-controlled architectures like the described SI coprocessor it is desirable to enable a convenient automated arbitration of shared resources. It can be expected that the vendors of behavioral synthesis tools will solve these issues in the near future. The expected increased functionality of SystemC synthesis tool will lead to an increased acceptance of SystemC not only as modeling language but also as a design entry language aiming at a smooth path from an abstract description to a gate-level implementation.

4.2.2 Motion estimation coprocessor

The motion estimation coprocessor is designed using the A|RT tools [14]. In this design method, A|RT-Builder is used for designing the Application Specific Units (ASUs), while A|RT-Designer is used for generating the VLIW ASIP which uses the ASUs apart from standard functional units like ALUs and ACUs. Both tools use C-based specifications enhanced with special C-types such as bit vectors and fixed-point types as input. Especially the C-based specification for A|RT-Designer can easily be wrapped into a SystemC PV model for use in a virtual prototype. A|RT-Builder is simply a language translation and takes a C-based RTL specification of an ASU as input, and creates a synthesizable RTL description in either VHDL or Verilog. The A|RT-Designer tool assists designers in the development of a hardware processor, customized for the C-algorithm that has to be executed on this architecture. The generated processor consists of a set of data path resources, controlled by a VLIW type controller and is created in four steps.

The starting point for A|RT-Designer is a C-based algorithm that is compiled to an internal representation during the first step. In the second step the architecture is generated by A|RT-Designer and is composed of standard resources (like ALU, ACU, MULT, constant ROM/RAM) and application specific resources (ASUs created with A|RT-Builder) from one or more libraries. In the third step the algorithm is mapped onto the generated architecture. Variables and constants are mapped on available memory resources (register files, RAM, ROM), followed by assigning operations to the data path resources and connection of the ASUs with a set of register files including the generation of their interconnects. Finally the fourth step performs scheduling and register assignment in such a way that the global machine cycle count is minimized while keeping the number of necessary registers as low as possible. This step involves a significant manual control of the tool by the designer to optimize the final schedule e.g. through the use of *pragmas*.

In order to use the A|RT tool chain the original C++ behavioral description of the motion estimation algorithm is partitioned and translated into ANSI-C code as required by A|RT Designer (Fig. 11). First the behavioral description of the ME (Fig. 11(a)) is partitioned into a SW task that prepares the motion model candidates and should run on the ARM CPU and a HW task that performs the main processing loop of the motion estimation algorithm (Fig. 11(b)). The input parameters to the HW task consist of two parts namely frame constants (e.g. frame size) and run-time parameters (e.g. motion model candidates and block coordinates). The C-code of the HW task is translated into ANSI-C, as required by A|RT Designer, and an initialization state is introduced such that frame constants are communicated only once. Furthermore several new data-types are introduced to allow communication of run-time parameters on a stripe (eight pixel blocks) basis. Next, the C-code of the processing functions (data-path) in the HW task is modified by integrating behavioral models of the ASUs (Fig. 11(c)). The model resulting from this step is also wrapped into a SystemC PV

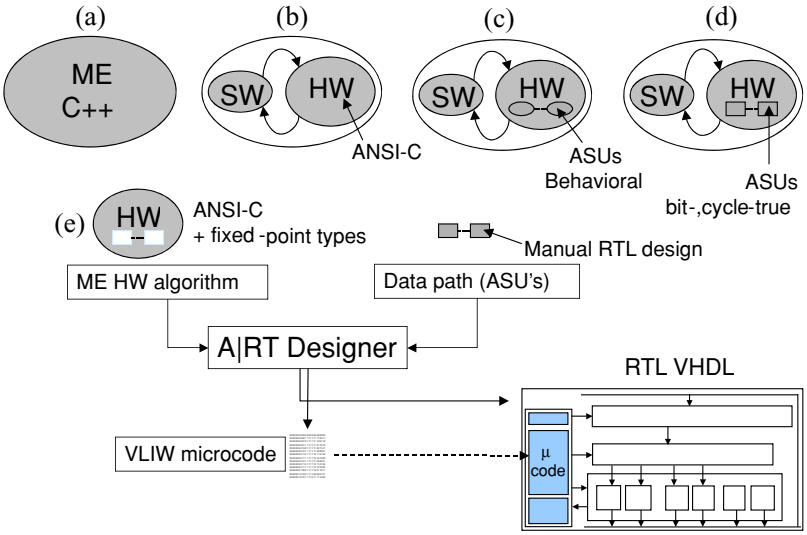

Fig. 11 Motion estimation design flow

model for integration in the virtual prototype as depicted in Fig. 5. Finally bit and cycle true models of the ASUs are integrated replacing the behavioral code of the ASUs (Fig. 11(d)). Each step is verified with the reference C++ code of the ME by comparing the intermediate results of the motion estimator such as the generated candidate motion models and the resulting motion models calculated by the HW task of the ME.

The C-code of the HW task resulting from this last step can directly be used as an input of A|RT Designer (Fig. 11(e)) and in fact is the ME CA model as depicted in the overall design flow in Fig. 5. The result is a synthesisable RTL description of a custom VLIW processor, consisting of a data-path and a controller. The controller contains an FSM that determines the next instruction to be executed, and a micro-code ROM, that contains the scheduled VLIW code of the HW task C algorithm.

4.2.3 Coprocessors synthesis results

The synthesis results for both the FPGA and standard cell implementation are listed in Table 1. In total ten single-ported 256 × 32 bits RAM blocks are used as embedded memory inside the ASUs of the ME. All intermediate and motion model results are mapped into a single RAM with a size of 64 Kbits. Furthermore the controller of the ME integrates several ROM blocks with a total size of 172 Kbit. The SI integrates in total 40 Kbits RAM. With a target clock frequency of 150 MHz, the arithmetic unit of the SI has a peak performance of about

Table 1 FPGA and Standard cell synthesis results

Technology	Altera FPGA		CMOS 90 nm @150 MHZ (mm^2)			
Resources	Logic	Memory	Logic	RAM	ROM	Total
SI	1.2 Mgates	40 Kbits	0.72	0.13	–	0.85
ME	0.8 Mgates	246 Kbits	0.45	0.26	0.14	0.85

3 GOPS and the ME can process 150 frames per second for a frame size of 352*288, using a single scan and 15 motion models per block.

4.3 SI coprocessor verification

For the verification of the SI coprocessor a test bench is developed. This test bench should achieve a reasonable coverage. However, as the SI coprocessor supports a certain range of programmability, i.e. different image segment sizes and instruction parameters, even a reduction of the complete test set to a minimum, e.g. by only checking corner cases of the parameter set, still leads to several hundred of tests that need to be executed. Because the occurrence of potential design bugs may also depend on the values of the input data set, it is important to vary the input patterns applied to the SI coprocessor under test as well. As the resulting large number of tests cannot be executed in an interactive way, it is required to run all the tests automatically, e.g. by execution of a script. Moreover, it is desirable to create a self-checking test bench, which condenses the result of a certain test to a simple 'ok' or 'not-ok' statement.

The resulting SystemC test bench created for the SI coprocessor is depicted in Fig. 12 and comprises a functional CPU model, the SI coprocessor model and a memory model, representing the shared system memory accessed by both the coprocessor and the system CPU. The CPU model in fact is a C++ program with access to the system interfaces. Therefore it can also be used to interpret a scripting language controlling the execution of the checks to be performed. Moreover, the CPU model is also able to generate the reference results. This is achieved by integration of the functional SystemC reference implementation available from the coprocessor implementation phase, into the CPU model.

Furthermore the memory model is extended with functionality that allows an on-the-fly comparison of the results generated by the CPU model and results produced by the coprocessor implementation. Moreover, the memory model has the capability to read or write images under control of the CPU model.

A macroinstruction is validated for a certain parameter set by first running the reference code on the CPU model producing the reference data that is written into the memory model. Afterwards the SI is programmed to perform exactly the same macroinstruction with the same parameter set. While the SI is writing its result into the memory model it is compared with the reference data by the checker module. Any deficiency is monitored and can be reported in various ways, depending on the validation settings. The memory model can generate a 'DIFF'-file, which indicates every pixel that differs between the SI and reference implementation including the results from both implementations.

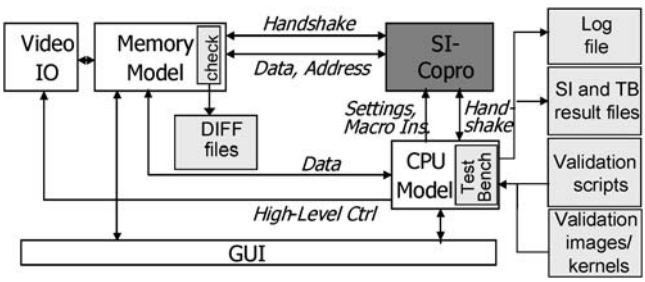

Fig. 12 SI reference model

The described test environment can be used for verification of the PV, PVT, CA as well as the SystemC-RTL model of the SI. Naturally, the simulation speed decreases with an increased accuracy of the applied model. For example, a verification suite that executes about 180 macroinstructions and processes 6 million pixels is executed within 2 minutes when using the PV representation of the coprocessor. The same test executed on the RTL model has a runtime of approximately 4 hours.

5 Hardware/software integration

HW/SW integration aims at joining together both HW coprocessors with the embedded application SW running on the ARM. In order to validate this HW/SW integration an FPGA based prototype is developed. However, instead of developing the FPGA prototype directly, an intermediate SystemC virtual prototype is used as a model of the target architecture. The aim of applying virtual prototyping is to speed up and ease the porting of the embedded application SW. Furthermore for this embedded SW development a layered approach is applied which allows for a seamless migration from the PC environment to the prototypes. Such layered approach provides a separation between the application functionality and the underlying prototyping infrastructure and, therefore, allows easy SW porting and reuse between the Virtual- and FPGA-prototype. Section 5.1 will explain the embedded SW development followed by a detailed description of the VP in Section 5.2. Finally, Section 5.3 will discuss the FPGA based prototype.

5.1 Embedded SW development

Originally, smart imaging algorithms were created using the C++ programming language and verified in a PC-based environment. Adapting from the PC environment to the embedded architecture typically means an arduous task that requires a lot of effort in rewriting and revalidating the application SW. In order to reduce such effort a similar SW structure in both the PC and the embedded prototype is kept, which enables unchanged reuse of most platform independent code and also simplifies the error detection and debug of the SW once ported to the embedded CPU.

As shown in Fig. 13, the smart imaging application developed in the PC environment is structured in three levels as explained in Section 2.1. In the upper level, the HLA communicates with the MLAs by invoking methods on their classes. MLAs, including the motion segmentation algorithm, compose the intermediate level of the software structure. MLAs execute LLAs via the C-IPL API, which in the PC environment is a compendium of SW routines.

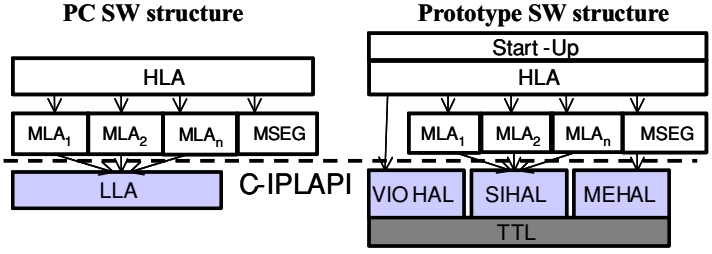

Fig. 13 Software structure

During the embedded SW development a similar structure is kept where the three previous layers are still present. Besides these three layers, start up code is added to the embedded software in order to configure and initialize the coprocessors, as well as to set the SW structures necessary for the HW/SW communication and synchronization. Furthermore, the lower-layer of the prototype SW structure is adapted to execute parts of the smart imaging functionality on the coprocessors instead of the SW algorithms used before.

For that purpose, as depicted in Fig. 13, three different hardware abstraction layers (HAL) are created. These HALs hide the low-level details of the system architecture and provides a well-structured API compliant with the C-IPL library for embedded SW development of the MLAs. The ME HAL controls and communicates with the Motion Estimator coprocessor. A set of functions as part of the C-IPL is provided to the Motion Segmentation algorithm in order to ease its communication with the ME coprocessor. The VIO HAL controls and communicates with a dedicated video input and output unit existing in the prototype architecture. This layer provides a group of functions to read video frames from the input interface and write back results to the output interface. Finally, the SI HAL controls and communicates with the Smart Imaging coprocessor. This HAL provides the same function calls and parameters passing as in the C-IPL to execute the LLAs functionality on the SI coprocessor. Instead of SW routines, the LLAs implemented in the SI HAL are based on sequences of macroinstructions that are executed by the SI coprocessor.

As an example, Fig. 14 shows the C-IPL HAL implementation (right) and its comparison with the original C-IPL used in the PC environment (left). As depicted, functions calls for performing a thresholding operation in a source image are equivalent (i.e. same name and input parameters) in both cases, but the original LLA functionality is implemented in the HAL using macroinstructions that control the SI coprocessor. As explained in Section 4.1.1 several macroinstruction classes are provided. These macroinstructions, which are 64-bits wide, are created and sent to the coprocessor using a set of specific commands. These commands are *SetConfInstruction* to set parameters in the coprocessor, *SetIOInstruction* to load/store blocks of data from/to the shared memory to/from the local memory of the coprocessor, and *SetExeInstruction* to execute specific operations (e.g. thresholding) on blocks of data previously loaded in local memory.

Furthermore, all three HALs are built on top of the TTL interface, see Section 2.3. As an example, Fig. 15 shows how the *SetExeInstruction* command is implemented using TTL. A

```
int
CamThreshold (Image *source, Image *dest, int threshold)
{
  CAM_PIXEL *srcptr,*dstptr,*cpsrcptr,*cpdstptr;

  c1=threshold;
  c2=0;
  c3=255;

  // ROI handling
  …
  srcptr = source->imageData;
  dstptr = dest->imageData;
  …

  for (y=0;y<height;y++)
  {
    cpsrcptr = srcptr; cpdstptr = dstptr;
    for (x = 0; x < width; x++, srcptr++, dstptr++)
    {
        if (*srcptr < c1) *dstptr = c2; else *dstptr = c3;
    }
    srcptr = cpsrcptr + source->widthStep;
    dstptr = cpdstptr + dest->widthStep;
  }

  return 1;
}
```

```
int
CamThreshold (Image *source,Image *dest, int threshold)
{
  c1 = threshold;
  c2 = 0;
  c3 = 255;

  // Parameters adjustments & ROI handling
  …

  SetConfInstruction (EXT_ARI,0,BASE,0, source->imageData);
  …

  for (posy = yOffset; posy < (yOffset + height); posy++)
  {
     // Load a line from memory to SI local memory
     SetIOInstruction (IO_Read,IO_Bank1,LINE_TYPE,width-1,true,0);

     // Execute threshold operation
     SetExe2Instruction (MI_THRESH, 0,LINE_TYPE,width-1,0, 255);

     // Write a line from SI local memory to memory
     SetIOInstruction (IO_Write, IO_Bank1,LINE_TYPE,width-1,true,1);
  }

  return 1;
}
```

Fig. 14 C-IPL implementations: (left) PC environment vs. (right) SI HAL

```
SetExe2Instruction ( /* parameters */ )
{
  MacroInstruction[0] = /* filled with post-processed parameters */ ;
  MacroInstruction[1] = /* filled with post-processed parameters */ ;

  if (number_MI_sent == sync_granularity)
  {
     // Ask for room to write a new group of macroinstructions
     reAcquireRoom (macroPort, sync_granularity);
     number_MI_sent = 0;
  }

  // Write the macroinstruction in the channel (shared memory)
  store (macroPort, number_MI_sent, 1, MacroInstruction);
  number_MI_sent++;

  if (number_MI_sent == sync_granularity)
  {
     // Update channel info and synchronize with the other channel's side
     releaseData (macroPort, sync_granularity);
  }
}
```

Fig. 15 SetExe2Instruction implementation using TTL

TTL logic channel is used to communicate and synchronizes the SW running in the CPU and the SI coprocessor. Physically this channel is mapped on the shared memory available on the system. Channels contain tokens. In this case, a token is a 64-bit macroinstruction (i.e. two 32-bit words). A TTL logic port (*macroPort*) connects the SW task with the logic channel. TTL primitives are executed on this port. The *reAcquireRoom* primitive is used to check if there is enough free space to write in the channel. This is a blocking primitive that only returns when room for *sync_granularity* tokens is acquired. The *store* primitive is used to write the actual macroinstruction in the channel. Finally, the *releaseData* primitive is used to update the status of the channel, enabling the coprocessor to consume the new produced macroinstructions. Note that in order to reduce the overhead due to the synchronization the *reAcquireRoom* and *releaseData* primitives are performed at a coarser granularity than the *store* primitive ($sync\_granularity \gg 1$).

Summarizing, this layered approach allows that further modifications in the system architecture would only require slight changes in the HAL and/or the TTL implementation, keeping the rest of the software application unchanged. This eases significantly the porting of the reference and future applications (i.e. software reuse) to different architectures.

5.2 System-level virtual prototyping

Applying virtual prototyping in the development of the Smart Imaging core aims to: (1) shorten the design time by developing the embedded SW in parallel with the implementation of the SI and ME coprocessors, (2) test and tune early the interactions between the SW running on the embedded CPU and the coprocessors i.e. configuration, communication and synchronization, and (3) use the VP as a system-level test-bench in order to intensively verify the correctness of the applications after their partitioning and mapping on the target system architecture. In general this approach helps us in early bug detection, reducing the risk of having to redesign the system, and served as an intermediate step that smoothed the transition from the original applications to the FPGA prototype.

Due to the nature and complexity of the smart imaging algorithms, validating the functional correctness of the applications running on the prototype requires the execution of hundreds of

frames of a specific scenario. Moreover, multiple scenarios with specific conditions in terms of lighting, number of objects in the scene, behavior of the objects, etcetera, have to be tested to ensure that an application is working properly. This leads to the need of a high simulation speed for our VP (in the order of MHz) to cope with such validation complexity. Since the prototype architecture and the application mapping are already decided in an early stage of the project, there is no need for an extensive architectural exploration and accurate performance analysis at the system-level using the VP. Therefore, timing accuracy is not important at this stage. The focus of the VP is more on assuring that the applications still produce the same results when executed in the prototype architecture. Hence, the VP has to precisely reflect the real memory map of the prototype architecture in order to create the start-up and HAL software. According to these requirements (i.e. fast simulation speed and register-accurate view of the architecture) the complete VP is developed at the PV level. The VP is built using the CASSE modeling and simulation environment [7]. A generic introduction to the CASSE framework is provided in Section 5.2.1. More details about how CASSE is applied to the development of the smart imaging VP are introduced in Section 5.2.2.

5.2.1 CASSE modeling and simulation environment

CASSE is a SystemC-based simulation environment that enables modeling and analysis of complex SoCs early in the design process. The tool combines application modeling, architecture modeling, mapping and analysis within a unified environment, with the aim to ease and speed up these modeling steps. Application modeling is based on the TTL interface. Architectural modeling is based on a group of highly configurable predefined elements provided by the tool libraries. CASSE is structured in three layers as depicted in Fig. 16.

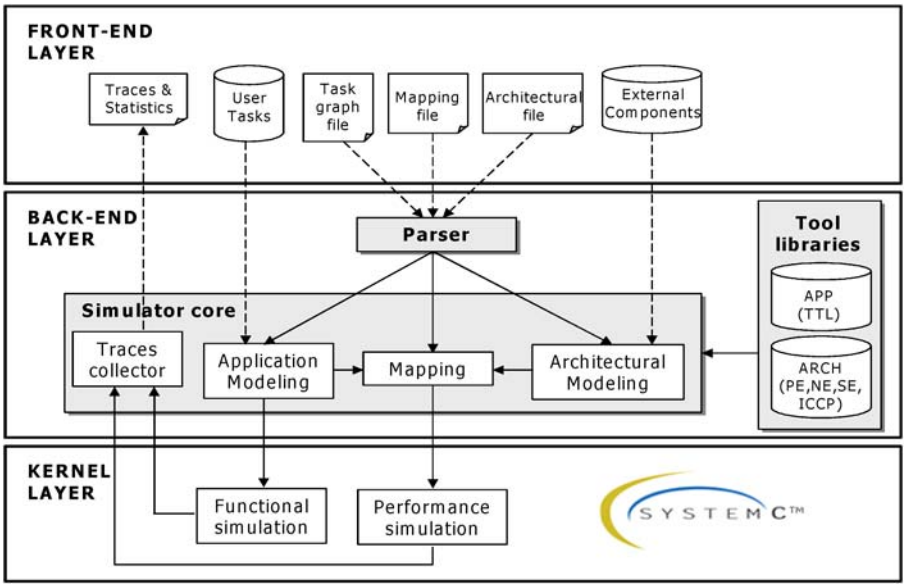

Fig. 16 CASSE internal structure

Front-end layer: the front-end layer serves as a user interface that controls the tool. This layer is composed of the user libraries and the description files. There are two user libraries: the *tasks library* contains TTL-compliant tasks composing the application and the *external components library* that contains user specific SystemC models to be added on the architecture model. There are three description files: the *task-graph file* that describe the structure of the application, the *architectural file* that describes the structure and configuration of the architecture, and the *mapping file* that describes how tasks and channels are allocated on specific elements of the architectural model.

Back-end layer: the back-end layer implements the core functionality of the tool. Besides a parser that reads and interprets the description files, this layer contains also two specific libraries. The application library (APP) where the TTL protocol is implemented and the architecture library (ARCH) where the group of predefined elements are implemented using the IEEE 1666 SystemC and the OSCI TLM standards [18]. These predefined elements are: processing elements (PE), storage elements (SE) and network elements (NE). All elements can be connected together in a 'plug and play' fashion by means of a generic TLM interface, called ICCP, provided also in the ARCH library.

CASSE is able to carry out two kinds of simulations: functional simulations and performance simulations. During functional simulations the tool only requires the *task-graph file*. Based on the information of that file the tool automatically instantiates and bind together tasks and channels (from the user and tool libraries, respectively), creating an executable model of the application. During performance simulations the tool read and parses the *task-graph*, *architectural* and *mapping* files. Predefined elements (PE, NE, SE) and external components (EC) are automatically instantiated (from the respective libraries) and connected together following a modular approach according to the *architectural* file. Tasks and channels are allocated on specific PE and SE elements according to the *mapping* file. All elements are configured according to the *task-graph* structure and the parameters specified in the description files. The outcome of this process is an executable model of the system instance.

Kernel layer: these executable models are then run by means of the SystemC kernel, which constitutes the third layer of the tool. During SystemC simulations execution traces and statistics can be recorded and dumped to output files for later inspection and analysis. This analysis might guide further iterations where both the application and the architecture models are tuned, or a new mapping is selected.

More details about the PE, NE and SE predefined elements and the ICCP interface available in the ARCH library are discussed next.

ICCP is a generic communication protocol, which defines a point-to-point TLM interface and a group of communication primitives between two entities named *Initiator* and *Target*. As shown in Fig. 17, the ICCP protocol provides two basic methods for communication between the *Initiator* and *Target* entities: *read* and *write*. The execution of any of the two basic methods is started in the *Initiator* module. During such execution all information related to the transaction is passed from the *Initiator* to the *Target* within the *RequestGrp* structure using the standardized bidirectional transport interface (*tlm_transport_if*). The *transport* function is in fact executed in the *Target* module, where the transaction is split in three different phases (*Request*, *Read-/Write-Data* and *Response*) that are executed in the slave module connected to it. Once the transaction finishes the *Target* returns from the *transport* function and sends back to the *Initiator* information related to the completion of the transaction within the *ResponseGrp* structure. The timing of the operations carried out over the ICCP depends on the combination of the latencies programmed in the *Initiator* and *Target* modules and may vary from completely untimed (PV) to cycle-accurate at the transaction boundaries (PVT).

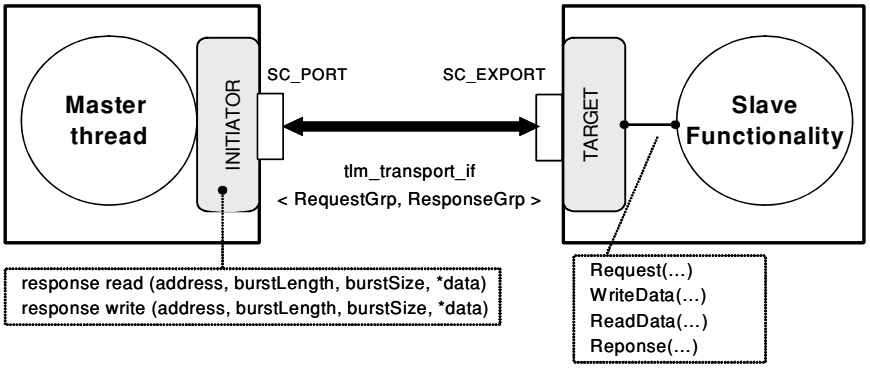

Fig. 17 ICCP interface implementation

PE's are used to model generic computational units. By default, PE's do not contain any functionality, but they are simply placeholders where the task's functionality and timing is executed. As depicted in Fig. 18, a PE is composed of several modules. An arbitrary number of tasks can be assigned (mapped) in a single PE via the multitask container module (MTC) that implements a dynamic vector of SystemC threads. However, only one task can be active at a certain time on a PE. This is assured by means of the *Task Scheduler* module that implements several scheduling policies (e.g. priority-based, cooperative multitasking, TDMA) and supports advanced features such as preemption and interrupts handling. PE's also contains a TTL shell that implements the TTL primitives and translates the logical communication via ports to device level communication via the ICCP *Initiator* interfaces of the PE.

SE's model generic random access memory elements, such as register files or static RAM memories. Storage elements can be configured with an arbitrary number of *Target* ICCP interfaces. This allows emulating the behavior of single, dual or multi-port memories existing on the system architecture.

NE's model generic shared interconnections, such as on-chip shared busses. NE's can be configured with an arbitrary number of *Target* (input) and *Initiator* (output) interfaces. The main functionality of a NE is to interconnect architecture elements. NE's include configurable input buffers, an arbiter module, an address decoder module and a controller module. Basically, the controller module routes transactions from the *Target* interfaces to the *Initiator* interfaces.

5.2.2 Smart imaging virtual prototype

The created smart imaging VP, as shown in Fig. 19, is composed of an embedded CPU, three dedicated coprocessors (i.e. ME, SI and VIO), several shared memories and a communication network that in turn is composed of several busses and bridges. This setup reflects the internal FPGA architecture, which is partially based on an ARM9 subsystem as explained in Section 5.3. Shared memories are modeled using generic SE components configured with the right size and number of interfaces. Busses, bridges and memory controllers are modeled using generic NE components configured with the right number of interfaces, arbitration policy, and addressing range for all their output interfaces. Such addressing ranges are selected to reproduce the memory map used in the real prototype. Moreover, communication latencies

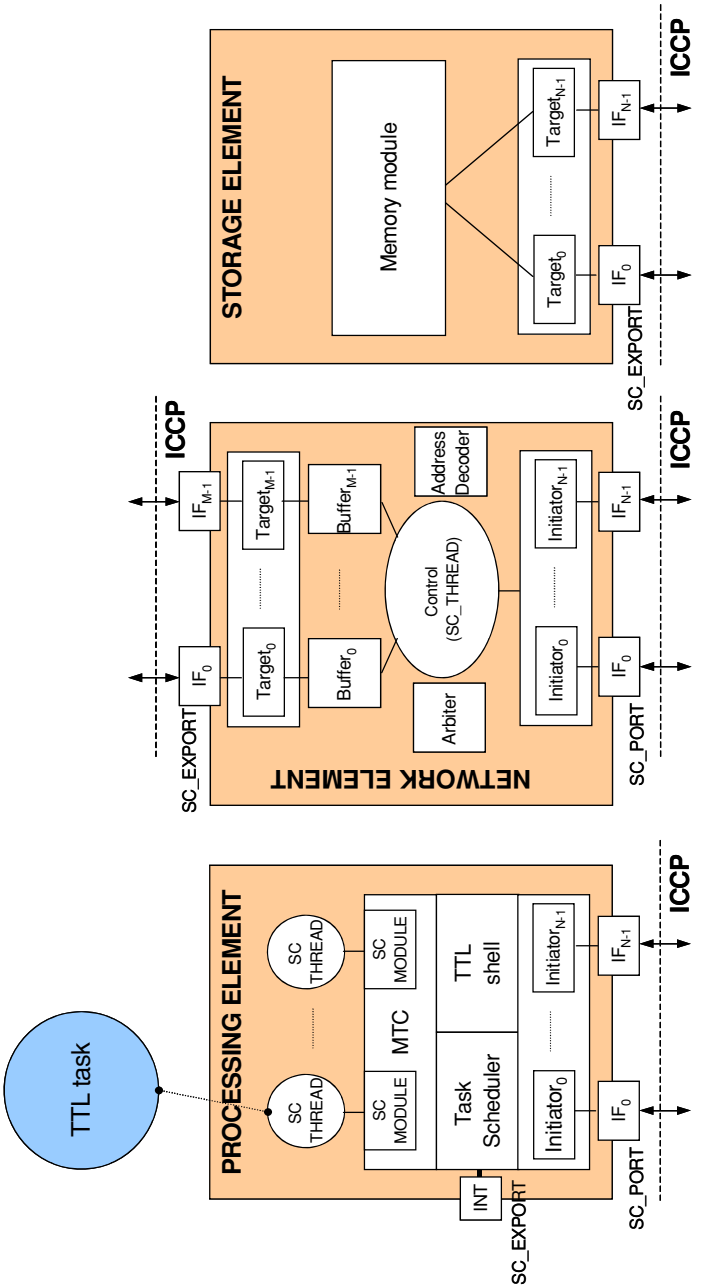

Fig. 18 PE, NE and SE structure

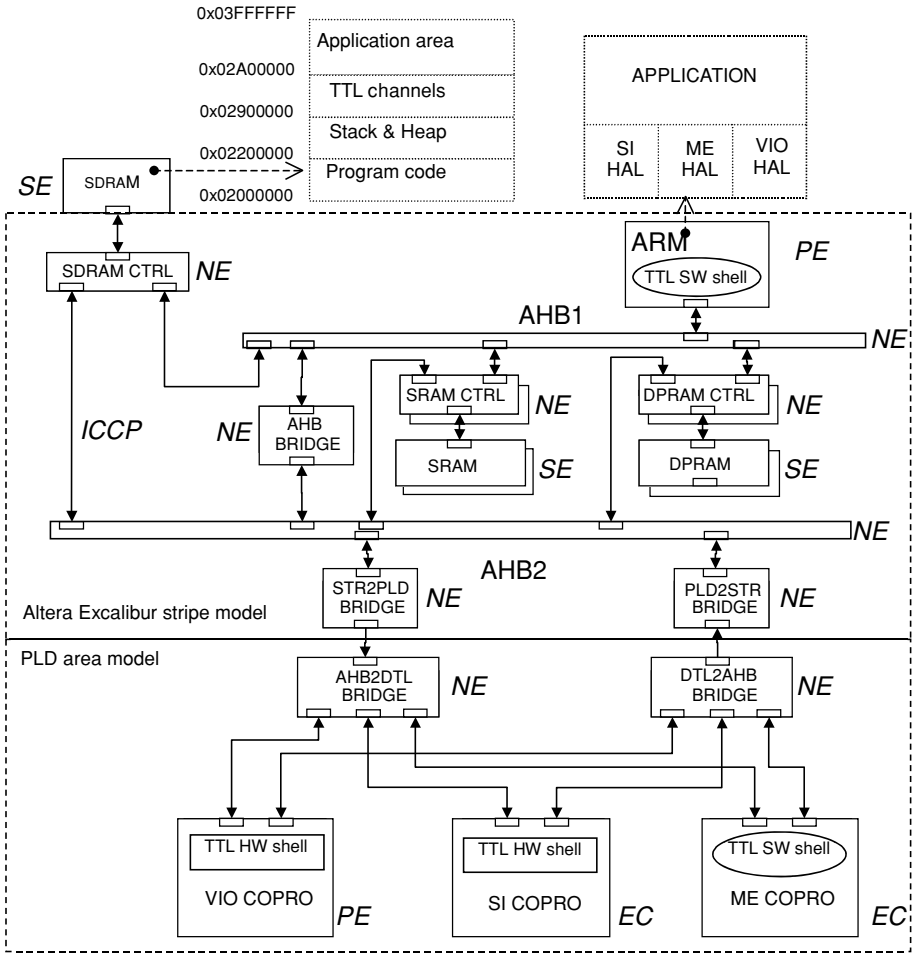

Fig. 19 Smart imaging virtual prototype

in all components are set to zero (i.e. untimed communication) in order to achieve the fastest simulation speed possible. Using CASSE such complex architectural model is quickly created and configured by means of an *architectural description file* that only takes 155 lines.

Instead of an instruction set simulator (ISS) to execute the embedded SW, an abstract CPU model is used. Although much more accurate, an ISS would slow down the simulation speed considerably to the range of hundreds of KHz, making the HAL development and functional validation tasks unfeasible within a reasonable amount of time. For that reason, the source code for the embedded SW is encapsulated into a task and mapped on a PE component conforming the abstract CPU model. These Host Code Emulation (HCE) techniques are applied in order to allow the encapsulated SW running on the PE to access all relevant data structures using exactly the same memory map of the real prototype. Such relevant data structures (e.g. image data and TTL channels) are mapped on the emulated memory models (i.e. SE), and the abstract CPU model accesses them through the NE elements and ICCP interfaces provided by the CASSE libraries. These HCE techniques also allow that the same

source code executed on the abstract CPU can be reused later on for the embedded ARM without any change. The only difference between both prototypes is the underlying TTL implementation, which in case of the VP is included in the PE and in case of the FPGA has to be customized for the ARM.

The PV models for the SI and ME coprocessors, described in Section 4, are now integrated into the VP by means of external components (EC). These models are functional equivalent and simulate hundred of times faster when compared with their SystemC CA counterpart models. The complete VP is able to process a frame in the range of 30–180 seconds depending on the complexity of the scenario. Such processing might take several hours in a more conventional HW/SW co-verification environment at the RTL level. The use of the VP significantly reduced the total SW development effort. It took around three months to finish the development of the three HALs and to port the embedded SW for four reference smart imaging applications targeted in the project. This software could later on be integrated directly in the FPGA prototype.

5.3 FPGA prototype

The FPGA prototype is built using a PCI based prototyping board with two Altera FPGA devices: an Excalibur XA10 device with 1 million logic gates and an APEX-1500 with 1,5 million logic gates. The Excalibur also embeds an ARM9 subsystem that is used to run the embedded software parts of the applications. The FPGAs are used to implement the hardware coprocessors and the top-level communication infrastructure. This FPGA prototype is very close to an actual chip implementation. Since the size of the SI logic after synthesis and place&route exceeded 1 million gates, the most likely partitioning of the smart imaging architecture on the prototyping board is to map the SI co-processor and its local memory on the APEX1500 FPGA device. The ME and the communication infrastructure are mapped to the Excalibur device. The infrastructure comprises the multiple DTL, AHB and PCI bridges.

Instead of integrating the SI and ME coprocessors, as well as the embeddedSW, on the FPGA prototype in one go, a different approach is followed. Our approach is based on the communication capabilities between the host PC and the FPGA via the PCI interface. In the FPGA side, the PCI interface is connected to the top-level communication infrastructure through a special logic that served as a bridge from PCI to the DTL communication

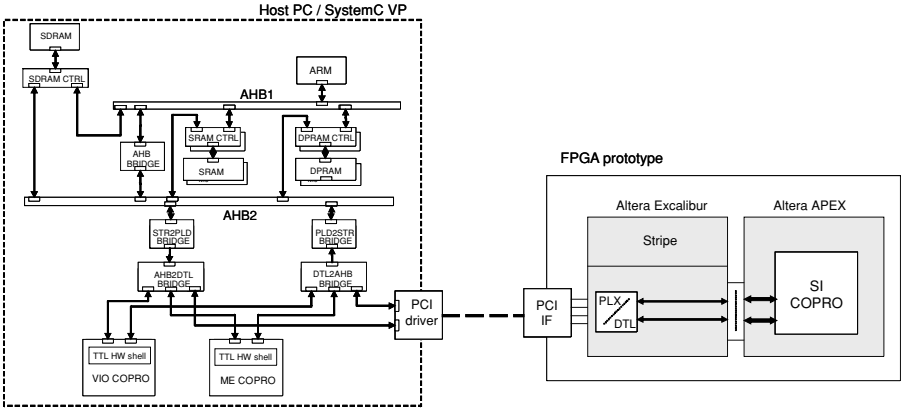

Fig. 20 Host PC/FPGA co-simulation

protocol used in the prototype. Likewise, in the PC host side, the PCI driver shipped with the prototyping board is encapsulated in a SystemC component and added to the VP. This new component serves as a bridge from the ICCP protocol used in the VP to PCI communication. Thanks to this, it is possible to migrate individual components from the VP, such as the SI coprocessor, into the FPGA board while keeping the rest of the architecture on the PC as a SystemC VP, see Fig. 20. The part running on the PC serves as system test bench for the component integrated in the FPGA. This helps significantly to manage the integration and verification complexity by gradually moving components from the VP into the FPGA.

6 Conclusions

In this paper the development of a complex smart imaging architecture following a SystemC-based design flow is presented. The smart imaging core integrates an ARM processor and two specific hardware blocks for image processing: a smart imaging coprocessor and a motion estimation coprocessor. A SystemC-based design flow is applied, comprising the design, synthesis and verification of the two coprocessors, as well as the development and integration of the embedded SW on the smart imaging core.

The two coprocessors are successfully modeled and refined from C/C++-based algorithmic descriptions down to architecture reference models using SystemC and TLM concepts. For the RTL implementation of the hardware coprocessors high-level synthesis tools are used. The applied SystemC based design flow enabled the iterative refinement of the architecture towards an optimal RTL implementation.

Furthermore, the use of SystemC TLM supported the integration of fast functional models of the coprocessors on a virtual prototype platform of the target architecture. This virtual prototype is beneficially used during the embedded SW development phase, which comprised the creation of several HW abstraction layers to communicate and synchronize the SW with the coprocessors. The usage of the SystemC virtual prototype, allowed shortening the design time of the entire system since the SW development is carried out in parallel with the implementation of the coprocessors.

The major advantage of a SystemC-based design flow, compared to traditional approaches, is the smooth transition from the algorithm representation (written in C/C++) to the actual implementation both for HW and SW design within a unified environment. The key element that has enabled such design possibilities is the emergence of the TLM modeling style together with the increasing acceptance of SystemC as a standard for system level modeling, design and synthesis. Hence, such methodology is becoming an attractive approach to be applied in actual design projects within the Semiconductors industry.

Acknowledgments We like to thank Bruno Steux from École des Mines de Paris, Thomas Hinz from Philips Semiconductors Hamburg, Jörn Jachalsky from University of Hannover, Pablo Santos from University of Las Palmas GC and Ghiath Alkadi from Philips Research Eindhoven for their contributions to the development of the smart imaging core. This work was partly sponsored by the European Commission in the IST-2001-34410 CAMELLIA project.

References

1. Camellia Image Processing Library http://camellia.sourceforge.net
2. Steux, B., and Y. Abramson. Robust Real-Time on-Board Vehicle Tracking System Using Particles Filter. In *IFAC IAV'04*, July 2004.

3. Abramson, Y., and B. Steux. Hardware-Friendly Pedestrian Detection and Impact Prediction. In *IEEE IVS'04*, June 2004.
4. Kyo, S., et al. A 51.2GOPS Scalable Video Recognition Processor for Intelligent Cruise Control Based on a Linear Array of 128 4-Way VLIW Processing Elements. In *IEEE ISSCC'03*, February 2003.
5. Raab, W., N. Bruels, U. Hachmann, J. Harnisch, U. Ramacher, and C. Sauer. A 100-GOPS Programmable Processor for Vehicle Vision Systems. In IEEE Design & Test of Computers, 2003.
6. Imagawa, K., K. Iwasa, T. Kataoka, T. Nishi, and H. Matsuo. Real-Time Face Detection with MPEG4 Codec LSI for a Mobile Multimedia Terminal. In *ICCE'03*, June 2003.
7. Reyes, V., T. Bautista, G. Marrero, P.P. Carballo, and W. Kruijtzer. CASSE: A System-Level Modeling and Design-Space Exploration Tool for Multiprocessor Systems-on-Chip. In *DSD'04*, August 2004.
8. van der Wolf, Pieter, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer, and Gerben Essink. Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach, CODES + ISSS '04, Stockholm, Sept. 2004.
9. Gehrke, Winfried, Joern Jachalsky, Martin Wahle, Wido Kruijtzer, Carlos Alba, and Ramanathan Sethuraman. Flexible Co-Processor Architectures for Ambient Intelligent Applications in the Mobile Communication and Automotive domain. In *Proc. SPIE Vol. 5117, VLSI Circuits and Systems*, April 2003, pp. 310–320.
10. Jachalsky Jörn, Martin Wahle, Peter Pirsch, Winfried Gehrke, and Thomas Hinz. A Coprocessor for Intelligent Image and Video Processing in the Automotive and Mobile Communication Domain. In *ISCE2004*, Sept. 2004.
11. Peleg, A., and U. Weiser. The MMX Technology Extension to the Intel Architecture. In *IEEE Micro*, vol. 16, no. 4, Aug. 1996.
12. Lanneer, D., et al. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Processors*, P. Marwedel (ed.), Kluwer Academic Publishers, 1995.
13. Hoffmann, A., et al. A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIP) Using a Machine Description Language. In *IEEE TCAD*, Nov. 2001.
14. A|RT Designer and A|RT Builder tools, formerly from Adelante Technologies, Now Marketed by ARM Ltd. as OptimoDE, http://www.arm.com/products/CPUs/families/OptimoDE.html.
15. Peters, H., et al. Application Specific Instruction-Set Processor Template for Motion Estimation in Video Applications. In *IEEE TCSVT*, vol. 15, no. 4, April 2005.
16. Cai, Lukai and Daniel Gajski. Transaction Level Modeling: An Overview. In *CODES + ISSS'03*, California, USA, October 2003.
17. Synopsys CoCentric System Studio, Home page, http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html.
18. Transaction Level Modelling Standard 1.0, June 2005, http://www.systemc.org.
19. Grötker, T., S. Liao, G. Martin, and S. Swan. *System Design with SystemC*, Kluwer, 2002.
20. Henkel, J. Closing the SoC Design Gap. *IEEE Embedded Computing*, 2003.
21. Paulin, P., and Magarshack, P. System-on-Chip Beyond the Nanometer Wall. In *Proceedings of the 40th IEEE/ACM Design Automation Conference*, ACM Press, 2003, pp. 419–424.
22. Martin, G., and F. Bacchini. System Level Design: Six Success Stories in Search of an Industry. In *Proceedings of the Design Automation Conference*, ACM press, San Diego, California, USA, June 2004.
23. Rose, A., S. Swan, J. Pierce, and J. Fernandez. Transaction Level Modeling in SystemC, SystemC TLM whitepaper, 2005.
24. Forte Cynthesizer, Home page, www.forteds.com.
25. Pandita, R., M. Leclercq, and J. Speros. Enabling Performance Evaluation of SoCs with SystemC Model for the TI OMAPTM Platform. In *Proceedings of the GSPx'04 Conference*, Santa Clara, California, USA, Sept. 2004.
26. Bruschi, F., and F. Ferrandi. Synthesis of Complex Control Structures from Behavioral SystemC Models. In *Proceedings of the Design Automation and Test in Europe*, Munich, 2003.
27. Portero, A., O. Navas, and J. Carrabina. Hw-Sw Design Methodologies Used for a MPEG Video Coprocessor Synthesis. In *Proceedings of the 16th International Conference on Microelectronics*, vol. 3, pp. 1688–1693, 2004.
28. Catapult, C., Home page, www.mentor.com/products/c-based_design/.
29. Toshiba R-CUBE project, 2005, www.semicon.toshiba.co.jp/eng/r_cube/.
30. ARM AMBA AXI protocol specification, June 2003.