SystemCoDesigner—An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications

JOACHIM KEINERT, MARTIN STREUBÜHR, THOMAS SCHLICHTER, JOACHIM FALK, JENS GLADIGAU, CHRISTIAN HAUBELT, and JÜRGEN TEICH
University of Erlangen-Nuremberg
and
MICHAEL MEREDITH
Forte Design Systems

With increasing design complexity, the gap from ESL (Electronic System Level) design to RTL synthesis becomes more and more crucial to many industrial projects. Although several behavioral synthesis tools exist to automatically generate synthesizable RTL code from C/C++/SystemC-based input descriptions and software generation for embedded processors is automated as well, an efficient ESL synthesis methodology combining both is still missing. This article presents SystemCoDesigner, a novel SystemC-based ESL tool to automatically optimize a hardware/software SoC (System on Chip) implementation with respect to several objectives. Starting from a SystemC behavioral model, SystemCoDesigner automatically extracts the mathematical model, performs a behavioral synthesis step, and explores the multiobjective design space using state-of-the-art multiobjective optimization algorithms. During design space exploration, a single design point is evaluated by simulating highly accurate performance models, which are automatically generated from the SystemC behavioral model and the behavioral synthesis results. Moreover, SystemCoDesigner permits the automatic generation of bit streams for FPGA targets from any previously optimized SoC implementation. Thus SystemCoDesigner is the first fully automated ESL synthesis tool providing a correct-by-construction generation of hardware/software SoC implementations. As a case study, a model of a Motion-JPEG decoder was automatically optimized and implemented using SystemCoDesigner. Several synthesized SoC variants based on this model show different tradeoffs between required hardware costs and achieved system throughput, ranging from software-only solutions to pure hardware implementations that reach real-time performance for QCIF streams on a 50MHz FPGA.

Authors' addresses: Hardware/Software Co-Design, Department of Computer Science, University of Erlangen-Nuremberg, Am Weichselgarten 3, 91058 Erlangen, Germany. Forte Design Systems, 100 Century Center Court, Suite 100, San Jose, CA 95112.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*; J.6 [**Computer-Aided Engineering**]: *Computer-aided design (CAD)*

General Terms: Design, Experimentation, Languages

Additional Key Words and Phrases: System design, hardware/software codesign

ACM Reference Format:

Keinert, J., Streubühr, M., Schlichter, T., Falk, J., Gladigau, J., Haubelt, C., and Teich, J. 2009. SystemCoDesigner—An automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming application. ACM Trans. Des. Autom. Electron. Syst. 14, 1, Article 1 (January 2009), 23 pages, DOI = 10.1145/1455229.1455230. http://doi.acm.org/10.1145/1455229.1455230.

1. INTRODUCTION

Due to increasing chip capacities and computational power, processing of huge amounts of data becomes more flexible and less expensive. Corresponding examples can be found in digital signal processing, telecommunications, and network processing. In all cases, developers have to find an implementation that fulfills the user requirements in terms of throughput, latency, and energy consumption, while keeping the costs small. Typically, these exigencies can be best taken into account by hardware/software systems, where time critical operations are executed on special accelerators. However, whereas today the selection of an appropriate hardware platform is performed manually, increasing complexity, development costs, and time to market, pressure demand for an automatic approach.

ESL (Electronic System Level) design is expected to reduce design times by a factor of 10-100. Here, SystemC [Grötker et al. 2002] is becoming an industry de facto standard. The key advantage of SystemC lies in its actor-oriented approach, which is a widely accepted paradigm in system level synthesis [Lee et al. 2003; Lee and Neuendorffer 2004]. Actor-oriented models separate behavior from communication by means of actors and channels.

There still exists a gap, however, from ESL design to RTL (Register Transfer Level) synthesis. Although several commercial C/C++/SystemC-based behavioral synthesis (also known as high-level synthesis) tools are available—for example, *Cynthesizer* by *Forte Design Systems* [Forte Design Systems], *CatapultC* by *Mentor Graphics* [Mentor Graphics Corp.], and *NEC's CyberWorkBench* [NEC System Technologies, Ltd.]—and tools for automatic code generation for embedded processors exist [Leupers 2000; Murthy and Bhattacharyya 2006], efficient synthesis approaches from ESL models to hardware/software SoC (System on Chip) implementations are still missing.

To overcome these limitations, we have developed a tool called SystemCoDesigner, which uses an actor-oriented approach in order to integrate behavioral synthesis into ESL design space exploration tools. Using this approach, SystemCoDesigner is the first tool that reduces the number of manual steps as much as possible and provides a correct-by-construction generation of hardware/software SoC implementations from a behavioral model. This is achieved by starting the design process from an executable SystemC behavioral

model. Integration of Forte's Cynthesizer allows translating SystemC modules from this model into RTL code. The resulting actor implementations are characterized regarding their size (number of look-up tables, number of block RAMs, etc.) and their performance (latency). On the other hand, actors are compiled for embedded processor cores, making it possible to estimate execution times. The obtained values are used during the multiobjective design space exploration, where a highly accurate performance model is automatically generated from the SystemC behavioral model. The result of the design space exploration is a set of so-called *nondominated* solutions. From this set, the designer can select those solutions which are best suited for application and synthesize them, for example, using Xilinx EDK [XILINX 2005].

This article describes the corresponding design flow. A special emphasis is put on the integration of behavioral synthesis that permits the automatic generation of hardware accelerators from a given behavioral model. This capability extends results presented in a previous publication [Haubelt et al. 2007], where hardware accelerators still have to be implemented manually, thus requiring a lot of expertise and user interaction by the designer. Moreover, a Motion-JPEG decoder serves as running example throughout this article in order to illustrate the modeling complexity supported by SystemCoDesigner. In particular, the Motion-JPEG decoder contains complex modules whose manual hardware implementation would be time-consuming and error-prone. Here, the seamless integration of Forte's Cynthesizer high-level compiler into our design flow brings an impressive productivity gain. Finally, this article presents a comprehensive accuracy analysis of the proposed performance simulation, which is particularly important to understand when used in automated decision making (design space exploration). For this purpose, a new method for determining software execution times is proposed. As a result, it will be shown that new approaches in automatic software synthesis presented in this article reduce the schedule overhead by up to 25%.

The rest of the article is structured as follows: After presenting related work in Section 2, Section 3 introduces the Motion-JPEG example used throughout the article. Our design system, called SystemCoDesigner, is presented in Section 4. In order to integrate design space exploration and behavioral synthesis, a synthesizable subset of SystemC has to be specified. This is done in Section 5, which discusses SysteMoC, a SystemC library that strictly separates the data transformation from the communication behavior of a single actor. The automatic actor synthesis is discussed in Section 6. Section 7 is devoted to design space exploration, while Section 8 discusses the bit stream generation for FPGA-based hardware/software SoC implementations. Finally, Section 9 presents the results obtained by applying our proposed methodology to the Motion-JPEG decoder example using our SystemCoDesigner and Cynthesizer from Forte. As a key result, the decoder was implemented with immediate success in various configurations, while the overall development process took only about one month. The obtained implementations show different tradeoffs between hardware costs and achieved system performance, ranging from software-only solutions to real-time systems for QCIF streams targeting a 50MHz Virtex-II FPGA.

2. RELATED WORK

In this section, we discuss related work in the area of ESL design space exploration and synthesis. Tools providing ESL design space exploration are, for instance, *Sesame* [Pimentel et al. 2006], *MILAN* [Mohanty et al. 2002], *CHARMED* [Zitzler et al. 2002], and *WIZARD* [Chantrapornchai et al. 2000]. Furthermore, several publications in the area of design space exploration exist using different optimization strategies and considering different objectives [Kim et al. 2006; Mamagkakis et al. 2006; Gupta et al. 2000]. However, none of these approaches is able to automatically generate an FPGA prototype of the optimized implementations.

An approach supporting automatic design space exploration and synthesis is presented in [Kangas et al. 2006]. It is called Koski, and similarly to SystemCoDesigner, it is dedicated to the automatic SoC design. The input specification is given as a Kahn process network modeled in UML. The Kahn processes are refined using Statecharts. The target architecture consists of the application software, the platform-dependent and platform-independent software, and synthesizable communication and processing resources. Moreover, special functions for application distribution are included, that is, inter-process communication for multiprocessor systems. During design space exploration, Koski uses simulation for performance evaluation. Although many similarities can be identified between Koski and SystemCoDesigner, there are fundamental differences. The most important difference is that SystemCoDesigner integrates behavioral synthesis tools into the design space exploration, removing any manual interaction from the design flow.

Another approach for automatic mapping of digital signal processing applications to FPGA-platforms, is the *ESPAM* design flow [Stefanov et al. 2004; Nikolov et al. 2006]. It automatically converts a Matlab or C loop program into a Kahn process network. The latter can be transformed into a hardware/software system by instantiating processors and IP cores and connecting them with FIFOs. Integration of Sesame allows for automatic design space exploration. The Xilinx *EDK* tool is used for final bit-stream generation. The main difference between *ESPAM* and SystemCoDesigner lies in the modeling approach. Whereas *Compaan/Laura/ESPAM* uses Matlab or C loop programs as input specification, SystemCoDesigner uses SystemC, allowing for both simulation and automatic hardware generation using high level synthesis. Furthermore, SystemCoDesigner uses a more general hardware template for description of communication and module binding, offering various implementation possibilities, whereas the ESPAM hardware description is kept as simple as possible in order to simplify its usage. Moreover, *ESPAM* does not provide support for behavioral synthesis.

In Ha et al. [2006], the *PeaCE* approach is presented. Starting from a Ptolemy application model, it provides a seamless codesign flow from functional simulation to system synthesis. Moreover, PeaCE supports the generation of an FPGA prototype. In PeaCE, the application is modeled by a task graph where tasks are either signal processing tasks or control tasks. Signal processing tasks are modeled through synchronous piggybacked dataflow, a dataflow model with

control tokens. Control tasks are modeled by flexible finite state machines (hierarchical state machines without state transitions crossing hierarchy boundaries). The target architecture is basically a multiprocessor system. However, no support for behavioral synthesis is given.

Recently, the Center for Embedded Computer Systems [Center for Embedded Computer Systems] introduced its Embedded Systems Environment (ESE). ESE starts with a SystemC description and synthesizes an SoC implementation. In a first step, the mapping of operations to resources is done manually. In a second step, a SystemC transaction level model is generated, which is used for synthesis. Here, ESE provides a behavioral synthesis approach to generate hardware implementations from SystemC modules. However, in contrast to SystemCoDesigner, the automatic design space exploration is excluded, that is, the mapping has to be done manually.

Another interesting tool, called Cascade, is provided by CriticalBlue [CriticalBlue]. Starting from C/C++ or assembler code, Cascade generates hardware accelerators and corresponding interfaces to the processor core. However, in contrast to SystemCoDesigner, the generated accelerators are more fine-gained, since these accelerators are expected to replace single assembler instructions.

A different approach, named *Metropolis*, is proposed by Balarin et al. [2003]. Metropolis is a design space exploration framework that integrates tools for simulation, verification, and synthesis. Metropolis provides an infrastructure to help designers cope with the difficulties in large system designs by allowing the modeling on different levels of detail and supporting refinement. The applications are modeled by a metamodel consisting of sequential processes communicating via so called *media*. A medium has variables and functions, where the variables are only allowed to be changed by the functions. From the application model, a sequence of event vectors is extracted representing a partial execution order. Nondeterminism is allowed in application modeling. The architecture again is modeled by the metamodel, where media are resources and processes represent services (a collection of functions). Derivation of sequences of event vectors results in a nondeterministic execution order of all functions. The mapping is performed by intersecting both event sequences. Scheduling decisions on shared resources are resolved by so called *quantity managers*, which annotate the events. That way, quantity managers can also be used to associate other properties with events, like power consumption. In contrast to SystemCoDesigner, Metropolis is not concerned with automatic design space exploration. It supports refinement and abstraction, thus allowing top-down and bottom-up methodologies with a meet-in-the-middle approach. Since Metropolis is a framework based on a metamodel implementing the Y-chart approach, many system level design methodologies, including SystemCoDesigner, may be represented in Metropolis.

3. THE MOTION-JPEG DECODER

The JPEG algorithm [ITU 1992] is a widespread method for compression of images found in many embedded devices, such as mobile phones and digital

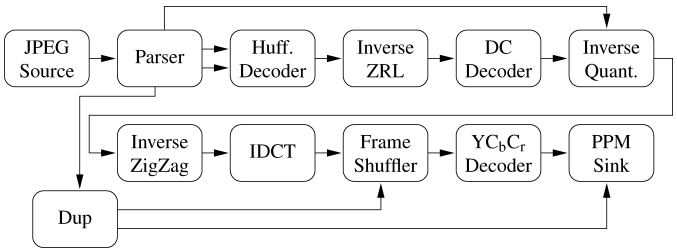


Fig. 1. Block diagram of a Motion-JPEG decoder, transforming a Motion-JPEG data stream into uncompressed images. Each block corresponds to a task performing a particular operation. Communication is illustrated by edges.

cameras. It reduces the required storage space for digital images by entropy encoding, spatial pixel decorrelation, and color space transform. A Motion-JPEG stream can simply be considered to consist of a sequence of individual JPEG images.

Figure 1 shows the block diagram of the corresponding decoder. It consists of several modules processing a stream of data. The *Parser* analyzes the processed JPEG stream and extracts important control information, such as image dimensions, number of colors, quantization strength and Motion-JPEG stream format. This information is forwarded to the other modules by embedding them into the stream of compressed data, or by separate *communication channels*. The *Huffman Decoder*, the zero-run length decoder (*InvZrl*), and the *DC Decoder* are responsible for entropy decoding. The *Inverse Discrete Cosine Transform* (*IDCT*) takes frequency coefficients in the form of 8×8 blocks, assembled by the *Inverse ZigZag* operation, and reestablishes the pixel values. Since JPEG applies lossy compression, by truncating the frequency coefficients in order to eliminate less important details, this operation is reversed as well as possible by the *Inverse Quantization*. The *Frame Shuffler* is responsible for reordering the pixels arriving block by block into a raster scan order required by most display devices and transmission protocols. The YC_bC_r *Decoder* finally converts the image into the RGB color space before the *PPM sink* generates one *Portable Pixmap File* [PPM format specification] per processed image.

In order to map this decoder to an embedded SoC, the designer has to decide which parts need hardware accelerators due to timing constraints, and which modules can run on an embedded processor, which normally is available in any case in such a system. This decision, however, is not easy to make, because the calculation effort for some of the modules shown in Figure 1 depends on the processed JPEG file content. Hence, in classical system design, several implementations have to be generated manually in order to determine whether they satisfy the requirements.

In order to alleviate this problem, SystemCoDesigner provides a new methodology for determining the optimum system architecture, fulfilling the designer's requirements by help of an automatic design space exploration. This is, subject of the next section, which gives an overview of the overall design flow, before the following sections investigate some steps in more detail.

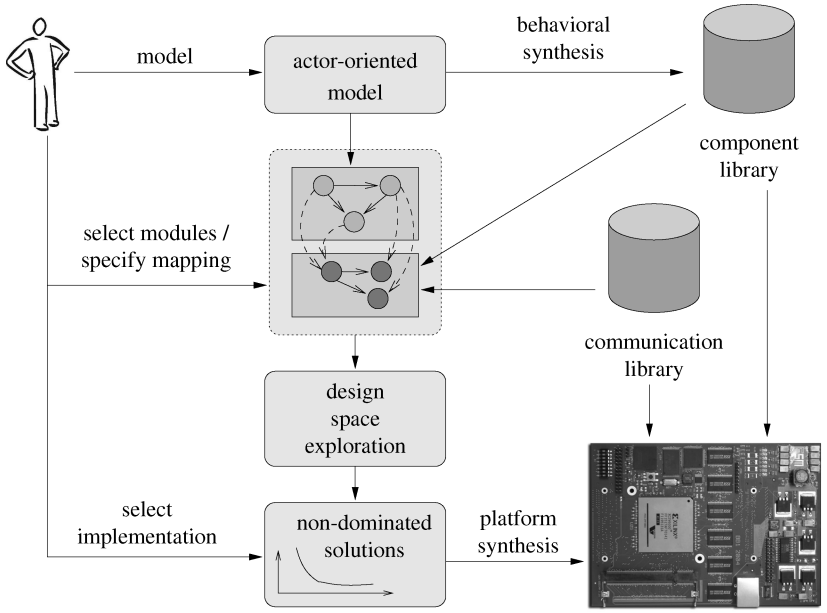


Fig. 2. ESL design flow using SYSTEMCODESIGNER: The application is given by an actor-oriented model described in SystemC. Behavioral synthesis is used to create *architecture templates*. Design space exploration using *multi-objective evolutionary algorithms* automatically searches for the best architecture candidates. The entire SoC is implemented automatically for FPGA-based platforms.

4. DESIGN FLOW

The overall design flow of our ESL technique is based on (i) actor-oriented modeling in SystemC, (ii) hardware generation for some or all actors using behavioral synthesis, (iii) determination of their performance parameters, such as required hardware resources, throughput and latency, (iv) design space exploration for finding the best candidate architectures, and (v) automatic platform synthesis. Figure 2 depicts the cooperation of these steps as implemented in SYSTEMCODESIGNER.

The first step in our ESL design flow is to describe the application by an *actor-oriented model*. This kind of description is particularly well suited for stream-oriented applications occurring, for instance, in the domain of digital signal processing or telecommunications. It naturally extends the block-based algorithm specification given in Figure 1 and exposes the coarse grained parallelism in an intuitive manner. Each module or process in Figure 1 is realized as an *actor* communicating with other actors by help of *communication channels*. The actor specification itself uses a special subset of SystemC, defined by the SYSTEMOC library [Falk et al. 2006]. The latter provides constructs to describe actors with well defined communication behavior and functionality. Furthermore, it offers communication channels by which the actors can be interconnected in order to obtain a complete system specification. This not only delivers an executable specification important for debugging and simulation, but also allows the extraction of the underlying mathematical model of

computation, which is a prerequisite for system analysis and exploration. Further details about these issues can be found in Section 5.

Each actor described in SystemoC can then be transformed into both hardware and software modules. Whereas the latter is achieved by simple code transformations, the hardware modules are built by help of Cynthesizer [Forte Design Systems], a commercial behavioral synthesis tool that we have integrated into our design flow. This allows us to quickly extract important performance parameters such as throughput and required hardware resources in the form of flip-flops, lookup tables, and multipliers. These values can be used to evaluate different solutions found during automatic design space exploration.

The performance information, together with the executable specification and a so called *architecture template*, serves as the input model for design space exploration. The architecture template is represented by a graph that contains all possible hardware modules, processors, and the communication infrastructure from which the design space exploration has to select the ones that are necessary in order to fulfill the user requirements in terms of overall throughput and chip size. The fewer components allocated, the less hardware resources are required. In general, however, this comes along with a reduction in throughput, thus leading to tradeoffs between execution speed and implementation costs. Multiobjective optimization, together with symbolic optimization techniques [Schlichter et al. 2006; Haubelt et al. 2006] are used to find a set of *nondominated* solutions. Each of these has the property to be not dominated by other known solutions in all objectives. For example, the design space exploration might return two solutions showing a typical tradeoff (e.g., throughput vs. flip-flop count). One solution has good performance and a huge effort in area. Another solution may use less area and achieves only lower performance. This means that both solutions do not dominate each other, hence the user may want to select the one serving his needs best, either the faster one or the cheaper one.

Once this decision has been taken, the last step of our ESL design flow is the automatic generation of the corresponding FPGA-based SoC implementation in order to allow for rapid prototyping. Since we use an actor-oriented application modeling, complex optimized SoCs can be assembled by interconnecting the hardware IP cores representing actors and potential processor cores with special communication modules provided in a corresponding library. Furthermore, the program code for each microprocessor is generated. Finally, the entire SoC platform is compiled into an FPGA bit stream using, for example, the Xilinx Embedded Development Kit (EDK) [XILINX 2005] tool chain for Xilinx FPGAs. Further details about the automatic platform synthesis step are given in Section 8.

5. MODEL OF COMPUTATION

To allow for automatic design space exploration, certain restrictions must be imposed on the input model. As mentioned previously, our design flow starts with an actor-oriented executable specification in SystemC. In such models, *actors* $a_1, a_2 \in A$ are communicating entities that are concurrently executed. Communication is restricted to dedicated *channels* $c \in C$. *Tokens* are produced

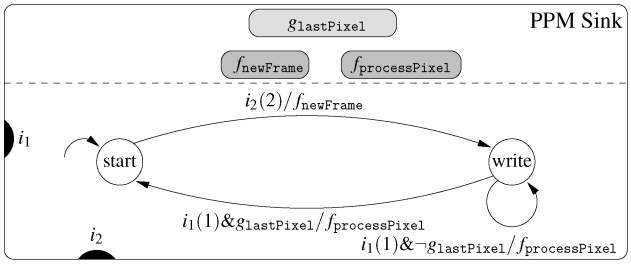
and consumed by actors and transmitted via those channels. Actors contain *ports* to which the channels are connected. Here, we focus on port to port communication media with extended FIFO semantics that connect exactly one *actor output port* with exactly one *actor input port*. In the following, these extended FIFOs will be called SYSTEMOC FIFOs if the extensions are relevant to the discussed matter. Otherwise, we will simply refer to them as FIFOs. Furthermore, we require that the actors of the SystemC design are partitioned into a *data path* for transforming token values, and a *control unit* supervising the *communication behavior* of the actor. In order to meet these requirements, we use the actor-oriented approach provided by SYSTEMOC [Falk et al. 2006], a SystemC library for actor-oriented modeling that requires usage of a well defined subset of SystemC.

In SYSTEMOC, each actor $a \in A$ is divided into three parts: (i) the actor *ports*, (ii) the actor *functionality*, and (iii) the actor *communication behavior* encoded as a *finite state machine* (FSM). Apart from the current state of this FSM, in the following called *communication state machine*, an actor may also possess a *functionality state*, for example, the values of some internal variables an actor may have.

The actor functionality is a set of functions that are only executed during transitions of the communication state machine. These functions are partitioned into *actions* and *guards*, which are distinguished by their ability to transform the functionality state of the actor. While actions may transform the functionality state of the actor and consume or produce tokens, guards are prohibited from doing this. During an execution of a transition of the communication state machine, the *action* annotated to this transition is executed atomically. Furthermore, a transition has an associated boolean expression called *activation pattern*, which must evaluate to *true* to enable the execution of the transition. Activation patterns consist of *guards* from the actor functionality as well as of predicates on the number of available tokens on input ports and on the number of free places on output ports. These predicates correspond to the number of tokens that can be accessed by the consequently executed action in a random fashion. For instance, $i_2(2)$ denotes a predicate that tests the presence of at least two tokens in the SYSTEMOC FIFO connected to the actor input port i_1, whereas $o_1(1)$ checks if at least one free place is available in the FIFO connected to the output port o_1. Actions consume and produce tokens for which the activation pattern assures presence and free slots, respectively.[1]

We will exemplify this via the *PPM Sink* actor from the Motion-JPEG model, which is depicted in Figure 3. Starting in the `start` state, the actor consumes two tokens from port i_2 as indicated by the corresponding predicate $i_2(2)$. These two tokens contain the frame dimensions for the next frame from which the

[1]The MoC underlying our methodology is a generalization of *FunState* (Functions driven by State machines) [Strehl et al. 2001]. In contrast to FunState, we allow all functions of an actor to share a state. Note that this idea is somehow similar to a rule-based model of computation for SystemC proposed in [Patel et al. 2006]. It is based on the idea of *guarded atomic actions* [Rosenband and Arvind 2004]. The complete system behavior is described by rules consisting of guards and atomic actions, where an action is ready for execution as soon as the guard evaluates to true. A behavioral synthesis tool based on these ideas is developed by Bluespec [Bluespec, Inc.].



(a)

```
(1) void newFrame() {
(2)   size_t Height = i2[1];
(3)   site_t Width  = i2[0];
(4)   missingPixels =
(5)     Width * Height;
(6) }
```

(b)

Fig. 3. (a) Depiction of the *PPM Sink* actor from Figure 1 consisting of two input ports i_1 and i_2, a set of *actions* and *guards* and the state machine that determines its communication behavior. The transitions of the state machine are annotated with *activation pattern/action* pairs. (b) shows the source code of the action $f_{\texttt{newFrame}}$.

action $f_{\texttt{newFrame}}$ shown in Figure 3(b) derives the number of pixels in the frame. The corresponding result is stored in the functionality state of the actor, that is, *missingPixels* is updated for later usage by the $g_{\texttt{lastPixel}}$ guard. Lines (2) and (3) show how to access the first and second token in the SystemoC FIFO connected to input port i_2 in random order. The size of the random access area corresponds to the predicate annotated to the transition of the FSM and amounts two for actor port i_2.

After completion of the $f_{\texttt{newFrame}}$ action, the FSM changes into the *write* state. There, the frame is read pixel by pixel from port i_1 and written to an unmodeled output, for example, in the simulation phase to a *ppm image file*. Each execution of the $f_{\texttt{processPixel}}$ action modifies the functionality state by decrementing the number of pixels remaining in the frame. The end of a frame is detected via the guard $g_{\texttt{lastPixel}}$, which evaluates to *true* for the last pixel of a frame. In this case, the state machine reverts to the start state.

6. AUTOMATIC ACTOR SYNTHESIS

After modeling the considered application in SystemoC, the next step in our design flow consists in the automatic generation of hardware modules for some or all actors. This allows for rapid prototyping, as explained in Section 8, and for extraction of performance parameters in form of action execution times and required chip areas. The latter ones are required for evaluation of each implementation during automatic design space exploration.

The actor hardware synthesis applies state-of-the-art behavioral synthesis tools. These behavioral synthesis tools typically do not support all possible SystemC language constructs, for example, recursion, dynamic memory allocation, and thus usage of dynamic data structures is not allowed.

The actor hardware synthesis itself consists of three steps, namely (i) transformation of SystemoC actors into SystemC modules, (ii) behavioral synthesis using *Forte Design Systems Cynthesizer*, and (iii) generation of Verilog gate-level netlists using *Synplify Pro* from *Synplicity* [Synplicity].

The actor transformation into SystemC modules is required in order to abstain from the abstract SystemoC syntax, which cannot be directly processed

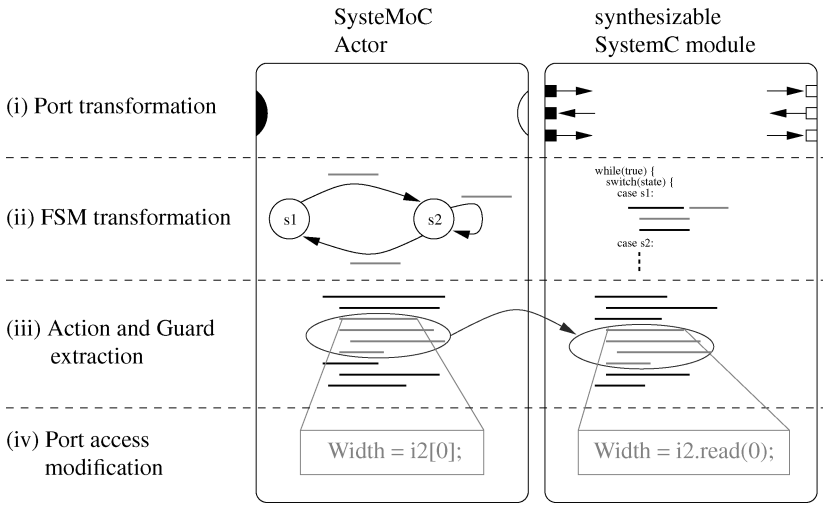


Fig. 4. Transformation steps of a SYSTEMOC actor to a synthesizable SystemC module.

by state-of-the-art behavioral synthesis tools. Especially, the communication state machine is encoded in an object-oriented manner allowing for automatic extraction of important communication properties. For behavioral synthesis, however, classical SystemC modules with parallel processes and well-defined ports driving hardware signals are required. Consequently, the SYSTEMOC actors have to be translated into such SystemC modules.

For this purpose, four steps are required, as depicted in Figure 4, namely (i) port transformation, (ii) FSM transformation, (iii) action and guard extraction, and (iv) port access modification.

The *port transformation* replaces each SYSTEMOC actor port interfacing an abstract SYSTEMOC FIFO by a SystemC hardware signal port. This is necessary in order to be able to drive the control and data signals of the hardware SYSTEMOC FIFO primitive contained in the communication library.

The *FSM transformation* automatically translates the abstract SYSTEMOC syntax of the communication state machine into a code fragment suitable for behavioral synthesis. This allows creating a SystemC module, containing a controlling process including the transformed communication state machine (see Figure 4). The latter checks, by means of a switch statement, for the current state. Within each state, an enabled transition is searched by checking the token counts in the input and output FIFOs and by calling the guard functions. If such a transition is found, the corresponding action function is called and tokens are consumed from the input FIFOs and produced on the output FIFOs. Furthermore, the state variable of the actor is set to the next state.

The *action and guard extraction* phase copies the corresponding functions from the SYSTEMOC specification into the generated SystemC module such that they can be invoked from the controlling communication state machine. The addition of special constraints for the behavioral synthesis tool allows obtaining optimized actor implementations.

The *port access modification* finally replaces all port accesses occurring in the guard and action functions, such as those illustrated in Figure 3(b), by function calls as shown in Figure 4. These functions are implemented by the SystemC hardware signal ports and drive the signals of the hardware primitives of the SysteMoC FIFOs. However, since the latter demand a communication protocol requiring at least one clock cycle, simple sequential function calls would result in bad system performance. Consequently, to allow concurrent writes to different hardware ports, we create processes for each SysteMoC output port that handle the hardware protocol. Concurrent reads from input ports are implemented by splitting the read access into two functions: the first function only starts the read access and does not wait for a clock edge, whereas the second function waits for the result from the input port.

7. AUTOMATIC DESIGN SPACE EXPLORATION

Having the application modeled in SysteMoC on the one hand, and the synthesized hardware modules for each actor on the other hand, an automatic design space exploration (DSE) can be performed. The latter explores optimal (or near optimal) solutions in terms of throughput, latency or required chip size by deciding which actors shall run on a processor core and which ones are better realized by a hardware module.

The first step is to formalize the particular instance of the hardware/software partitioning problem by providing a so called *architecture template*. This specifies all possible hardware modules and processor cores as well as their interconnection. An initial version of such an architecture template can be created automatically by instantiating one hardware accelerator for each SysteMoC actor, together with a configurable number of processors and the corresponding communication infrastructure. The user can then adapt it for the particular application by adding, for example, commercial IP cores.

For the case study performed in this article, the architecture template is restricted to one hardware module for each actor, one MicroBlaze softcore processor, and a certain number of hardware FIFO primitives realizing the abstract SysteMoC FIFOs. Each of these abstract SysteMoC FIFOs can be mapped onto four alternative FIFO primitives: For communication between hardware modules, two alternative hardware implementations exist, namely block RAM (BRAM) based and lookup table (LUT) based. The two other primitives represent hardware/software and software/hardware interfaces, connecting the SysteMoC FIFOs to *fast simplex links* (FSLs) for communication with the Xilinx MicroBlaze softcore processor. This target architecture in principle allows for hardware only, software only, and mixed hardware/software designs of the Motion-JPEG decoder.

Once the designer has decided on the possible architecture template for the considered application, a formal model has to be built that serves as input to the design space exploration. It consists of three parts, namely (i) the application itself, (ii) the architecture template, and (iii) the mapping constraints. The application is represented by a *process graph* $g_p = (V_p, E_p)$, which can be derived from the SysteMoC specification. Its actors $a \in A$ and channels $c \in C$ are

represented as vertices V_p, and their connections are modeled by edges E_p. The architecture template consisting of the hardware modules, processor cores, and communication channels is formally specified by the *architecture model* $g_a = (V_a, E_a)$, where the resources (CPUs, FIFO primitives, etc.) are modeled by vertices V_a and the interconnection topology is defined by edges E_a. The mapping relations between actors and channels in the SystemC model and resources in the architecture model are represented by so-called *mapping edges* $E_m \subseteq V_p \times V_a$. They specify which SysteMoC actors and channels can be bound to which hardware modules or processor cores. Possibly many *mapping edges* $e \in E_m$ for one vertex in the *process graph* onto different vertices in the *architecture model* represent different implementation possibilities of an actor, and thus set up the entire design space for automatic exploration. For each mapping edge, we annotate the action execution times of a SysteMoC actor occurring when the actor is bound to the hardware resource identified by the mapping edge. The action-accurate delay annotation improves the throughput and latency estimations when the execution times for the different actions, and hence actor invocations, are not identical. The hardware sizes are directly associated to the resources of the architecture model.

As soon as the designer has set up the formal model consisting of the process graph g_p, the architecture model g_a, and the mapping edges E_m, the automatic design space exploration can be executed. It searches for optimal implementations, that is, an allocation α of resources ($\alpha \subseteq V_a$) and a binding β of processes onto resources ($\beta \subseteq E_m$). The design space exploration is performed by the multiobjective evolutionary algorithms (MOEA) presented in Schlichter et al. [2006]. An MOEA is an optimization heuristic that is based on the principles of biological evolution. Starting with a set of implementations, an MOEA is performing in two steps: (1) Generating new solutions by variation of the implementations by *mutation* and *crossover*, and (2) a selection of good implementations based on their fitness. In multiobjective optimization, the fitness calculation is typically based on the nondominance property, that is, an implementation that is not worse than any other implementation in all objectives is said to be nondominated.

As a result of the design space exploration, we obtain a set of *nondominated* solutions, which is an approximation of the set of Pareto-optimal solutions. The evaluation of a concrete implementation bases on several *objectives*, such as latency, throughput, and hardware cost in the form of required LUTs, flip flops, and BRAM modules (for FPGA targets).

Whereas the necessary hardware cost can be determined by simply summarizing the module characteristics from behavioral synthesis and IP core data sheets, throughput and latency calculation is more complex. Due to the data-dependent behavior of the JPEG decoder, the latter can only be obtained by simulation of the application using system level performance models, which take the binding to hardware modules and processor cores into account. This is established by the *Virtual Processing Components* (VPC) framework [Streubühr et al. 2006], which simulates a particular architecture modeled in SystemC in order to derive average case estimations for latency and throughput. In order to allow for fast simulation, VPC builds a precompiled simulation binary that

Table I. Results of a Design Space Exploration Running for 2 Days, 17 Hours and 46 Minutes on a Linux Workstation with a 2400 MHz Intel® Core™2 CPU and 3GB of RAM

Parameter	Value
Iterations	300
Solutions evaluated	7,600
Non-dominated solutions	366
Exploration time	$2d\,17h46min$
Simulation time	$30.44s/solution$

is able to simulate all possible bindings for the chosen architecture template. Thus the allocation and binding can be loaded at simulation startup by simple configuration instead of generating multiple simulation binaries, thus leading to fast exploration times. Each time an action of an actor is executed, the execution time needs to be simulated together with effects resulting from resource contention. For this purpose, a scheduling policy for each resource has to be defined. Preemptive scheduling is established by a sophisticated event protocol implemented in the VPC framework [Streubühr et al. 2006].

7.1 Motion-JPEG Exploration Results

For the exploration of the Motion-JPEG example, we created an architecture template using one MicroBlaze softcore processor, 224 FIFO primitives, and 19 modules generated by behavioral synthesis.[2] The complete specification includes 319 mapping edges for the actors resulting in about $5 \cdot 10^{33}$ possible implementation alternatives.

Table I gives the results of a single run of the design space exploration for the Motion-JPEG decoder. The exploration has been stopped after 300 iterations, which correspond to a run time of 2 days, 17 hours, and 46 minutes.[3] The simulation time per solution is about 30 seconds for Motion-JPEG streams consisting of four QCIF frames with 176×144 pixels each. As a result, 366 nondominated solutions were found, each of them representing an arbitrary hardware/software implementation. Moreover, each solution can be automatically synthesized for an FPGA platform, as discussed in Section 8.

Note that it is possible to limit the number of nondominated solutions presented to the designer. This can be established by using bounded archives in the MOEA, which are used to store the best solutions found so far. When using bounded archives, the pruning of the archive always maximizes the diversity of the stored solutions. That way, the designer still is able to perform unbiased decision-making.

[2]Due to complexity, some of the actors shown in Figure 1 are split into several submodules, which are not further detailed.

[3]Each iteration corresponds to the full evaluation of a full population of several so-called individuals, where each individual represents a specific hardware/software solution of the Motion-JPEG example.

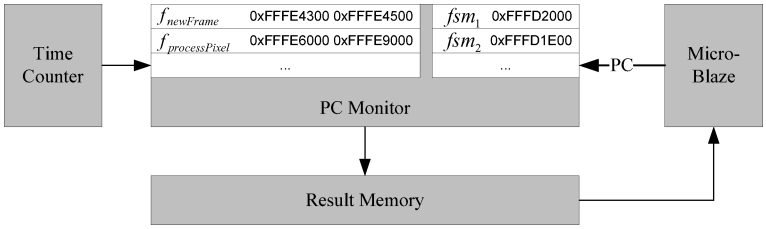


Fig. 5. Block diagram of the hardware profiler. The tables, being part of the *PC Monitor*, specify the program addresses of the action entrance and exit points as well as the start addresses for evaluation of the communication state machines.

7.2 Determination of Software Execution Times

Determination of precise action execution times for both hardware and software implementations is an important prerequisite for accurate performance evaluation. Whereas in Haubelt et al. [2007] these times have been measured using a Modelsim simulation, this approach is not feasible anymore due to the huge complexity of the deployed Motion-JPEG algorithm leading to extremely large simulation times. Alternative solutions would be to use an instruction set simulator or a software profiler such as *GNU gprof*. However, both approaches show severe deficiencies: the instruction set simulator provided by the Xilinx Embedded Development Kit (EDK, version 8.1) is not able to take the timing of external memories into account which we expect to have significant impact due to usage of an external SRAM for instruction and data storage. Software profilers have the inconvenience of modifying execution times by interrupt generation and possible cache influence. Furthermore, they only have limited precision, especially when rather fine-grained actions must be profiled, as in our case.

Due to these reasons, we have implemented a hardware profiler running on the FPGA that observes the program counter of the deployed MicroBlaze processor. The latter offers a corresponding interface, making this approach feasible.

Figure 5 shows the corresponding block diagram of the implemented hardware profiler. It obtains the entrance and exit points of each actor action, which can be obtained by special compiler tools. Furthermore, it disposes of the start addresses of the functions evaluating the communication state machines. By observation of the program counter, the hardware profiler can detect when the processor starts, and respectively terminates the evaluation of a communication state machine and the corresponding actor action. Together with the time counter acting as a clock device, the hardware profiler can derive the execution time spent in the considered action as well as in the required guard evaluation. These times are accumulated in the *result memory*, which can be read out by help of a special MicroBlaze program. This method hence realizes a real-time and side-effect free determination of the action execution times, allowing for accurate software exploration.

7.3 Reduction of Schedule Overhead

Despite an accurate determination of the action execution times, the overall system throughput and latency derived by VPC simulation and measurement

Table II. Schedule Overhead Reduction

	Round-Robin		Improved Round Robin		
M-JPEG Stream	Run-Time	Overhead	Run-Time	Overhead	Overhead reduction
QCIF 176 × 144	48.17 s	14.89 s (30.9%)	42.93 s	11.17 s (26.0%)	25%
Lena 256 × 256	166.24 s	41.67 s (25.1%)	151.04 s	35.17 s (23.2%)	16%

of the concrete software implementation differ. In Haubelt et al. [2007], we identified the *schedule overhead* as the underlying reason.

The latter occurs because all actors bound to a common MicroBlaze are checked in a round-robin fashion, whether one of their transitions can be taken or not. Since as the time for evaluation of the corresponding activation patterns (including guard functions) is not negligible, a schedule overhead results whenever an actor is polled, that is, checked for execution, while no transition is possible. This leads to a time consumption which cannot be taken into account by the VPC framework, as the latter uses an event-based scheduling strategy. In other words, VPC does not need to poll the actors in order to determine which ones can be executed, but it obtains a list with all enabled transitions. The latter is generated by the SystemC kernel in zero simulation time, thus leading to a discrepancy between the execution times predicted by the VPC and those measured in the final software implementation.

In order to alleviate this problem, we have modified the scheduler of the software implementation such that an actor that has been vainly polled for execution is not taken into account anymore until the number of tokens stored in at least one input or output FIFO has changed. Although this scheduler is rather trivial, we measured reductions in the schedule overhead of up to 25%. This is shown in Table II for two different Motion-JPEG streams. It compares the run-time to process four images on a softcore processor by deploying both a simple round-robin polling scheme and by taking the FIFO fill levels into account.

The schedule overhead has been obtained by help of the hardware profiler described in Section 7.2. Each time a communication state machine is executed without calling an action, the hardware profiler determines the spent time and adds it to a schedule overhead section in the result memory, which is accessible to the MicroBlaze for evaluation.

8. BUILDING THE COMPLETE SYSTEM

After termination of the design space exploration, automatic platform synthesis is the last step in our design flow (see Figure 2). Starting from the results obtained by the automatic design space exploration, it performs all steps required to generate an FPGA configuration file by interconnecting the previously generated hardware modules and processor cores with the required communication primitives. Therefore, as depicted in Figure 2, the designer selects those solutions from the automatic design space exploration that best fits his requirements. These solutions are used as input to our platform synthesis for FPGA-based platforms.

The platform synthesis itself works fivefold: first, for each allocated CPU resource, a MicroBlaze subsystem including memory and bus resources is

instantiated. Second, the allocated hardware modules are automatically inserted from the component library as Verilog netlists. The latter might either have been generated manually in order to allow for hand made optimizations, or they are automatically derived from the SysteMoC model as explained in Section 6.

Next, the communication resources are inserted from the communication library. The central element in this communication library is a generic synthesizable Verilog RTL module representing the SysteMoC FIFO primitives [Haubelt et al. 2007] for communication between hardware modules. This FIFO primitive implements the SysteMoC FIFO interface described in Section 5 and can deploy both embedded block RAM or distributed RAM for data storage on Xilinx FPGAs. Both versions only show a small overhead in comparison to native Xilinx COREGEN FIFOs [Haubelt et al. 2007]. The size of the FIFO is directly specified in the SysteMoC model.

The FIFO primitives representing hardware/software and software/hardware communication are derived by augmenting the hardware/hardware FIFO primitive with special *bridges* modules connecting the SysteMoC FIFO primitive to the MicroBlaze FSL links. Support of additional CPUs and buses requires the implementation of the corresponding bridge modules in the communication library. Software/software communication is implemented in a software library allowing the exchange of data within the MicroBlaze by reads and writes to local memory buffers. Special care is taken for efficient code generation in order to achieve low communication overhead. Template based code for instance permits powerful compiler optimizations which lead to fast data exchange. The result of these three steps is a hardware description file (*.mhs*-file in case of the Xilinx Embedded Development Kit [XILINX 2005]).

In the fourth step, the code required for each MicroBlaze processor is generated by an automatic transformation of the SysteMoC actors into C++ code (see Haubelt et al. [2007]). If several actors are bound to the same processor, then a round-robin schedule policy is deployed as described in Section 7.3. Finally, the platform specific bit stream is generated by using several Xilinx synthesis tools performing place and route and software compilation.

9. RESULTS

After having described the overall design flow of SystemCoDesigner, this section presents and discusses the quality of results obtained when applied to our SysteMoC Motion-JPEG Decoder implementation. The latter consists of approximately 8000 lines of code and has been restricted to color-interleaved base line profile without sub sampling for synthesis purposes. A Xilinx Virtex II FPGA (XC2V6000) has been selected as target platform for the implementations running at a clock frequency of 50 MHz. The objectives taken into account during design space exploration have been (i) throughput, (ii) latency, (iii) number of required flip flops, (iv) look-up tables and (v) block RAMs resp. multipliers.[4]

[4]In Xilinx Virtex-II FPGAs, hardware multipliers and BRAMs partially share communication resources and can hence not be allocated independently. Consequently, they are combined to one exploration objective.

Table III. Motion-JPEG Development Effort

Development Activity	Person Days
Specification and interface definition	4
Module implementation	16
Integration	3
Debugging	11
Code Adaption for Synthesis	4

Table IV. VPC Simulation Results

Nbr. of SW Actors	Latency	Throughput	LUTs	FFs	BRAM/MUL
0	12.61 ms	81.1 fps	44 878	15 078	72
1	25.06 ms	40.3 fps	41 585	12 393	96
8	4 465 ms	0.22 fps	17 381	8 148	63
all	8 076 ms	0.13 fps	2 213	1 395	29

Table III illustrates the development effort spent for realization of the complete Motion-JPEG decoder and its different hardware/software implementations. The first item represents the activity of breaking the JPEG standard [ITU 1992] down into a SysteMoC graph of actors. Module implementation encompasses the encoding of the actor's communication state machine as well as the corresponding actions and guards. During the integration phase, the actors have been connected to the complete system. Thanks to the actor-oriented methodology, this step could be accomplished very quickly, since the interface specification is relatively simple due to the applied FIFO communication semantics. Then, debugging has been required in order to detect errors in both the specification and the module implementations. Finally, coding constructs not supported by the behavioral synthesis tool (see Section 6) have been identified and replaced by equivalent instructions. Additionally, although not mandatory for obtaining a working system, we improved the performance of selected actors by means of synthesis constraints. The YCbCr conversion, for instance, could be accelerated by help of loop-unrolling.[5] Thanks to the integration of Forte Cynthesizer, the hardware accelerators for the different actors could be obtained directly from the SysteMoC specification. Hence, whereas traditional system design mostly requires the creation of both, a so-called golden model and the corresponding RTL implementation, behavioral synthesis helped us avoid this redundant effort. Furthermore, since SysteMoC offers a higher level of abstraction compared to RTL, the designer can progress more quickly. Taking the number of lines of code as a measure for complexity, we figured out that RTL design would have been 8-10 times more costly than the SysteMoC specification. The latter can be translated into a huge number of different hardware-software systems, which would not have been possible when directly recurring to RTL implementation.

Table IV shows the properties of some solutions found by design space exploration. Latency and throughput are VPC simulation estimates for JPEG streams consisting of four QCIF images (176×144 pixels). Table V provides the

[5]Its impacts are directly taken into account by the automatic design space exploration, because both the resource requirements and the execution times are determined after synthesis of the actors.

Table V. Hardware Synthesis Results

Nbr. of SW Actors	Latency	Throughput	LUTs	FFs	BRAM/MUL
0	15.63 ms	65.0 fps	40 467	14 508	47
1	23.49 ms	43.0 fps	35 033	11 622	72
8	6 275 ms	0.16 fps	15 064	7 540	63
all	10 030 ms	0.10 fps	1 893	1 086	29

Table VI. Influence of the MicroBlaze Cache for Software Only Solutions

	Processing Time for Four Images
without cache	146.3 s
with cache	42.9 s

performance values of the corresponding hardware implementations in order to give an idea about the achievable accuracy of the VPC estimations. Whereas it is not possible to give an upper bound of the occurring discrepancies, because the objectives used during design space exploration are nonmonotonic in the mathematical sense, the values typically observed are situated in the ranges resulting from Tables IV and V.

The differences in the required hardware sizes occurring between the predicted values during automatic design space exploration and those measured in hardware, can be explained by postsynthesis optimization and influence of the MicroBlaze configuration, in particular the number of FSL links. The difference in the block RAMs, for instance, we traced back to the fact that our SYSTEMOC specification uses 32-bit communication channels which, however, are not always entirely required. This fact offers a possibility for the Xilinx tools to trim some BRAM FIFO primitives.

The discrepancy between the VPC estimations for latency and throughput and those measured for the hardware-only solution could be traced back to the time spent in complex guards occurring, for instance, in the Huffman Decoder. The underlying reason is that VPC deploys an event-based simulation where the evaluation of guards is performed in zero-time by the simulation kernel, which is not true for the final hardware implementation. The discrepancy between the VPC estimations for latency and throughput and those measured for hardware-software solutions is due to the schedule overhead discussed in Section 7.3. Furthermore, we observed a tremendous influence of the cache enabled on the MicroBlaze processor.

This is illustrated in Table VI means help of a software-only solution. It shows the overall execution time for processing four QCIF frames.[6] As can be seen, the simple enabling of both an instruction and a data cache with 16 kBytes each results in a performance gain of factor 3.4. This is due to the fact that both instruction code and program data are stored on an external memory with relatively large access times. Since the cache behavior, however, depends on the program execution history, a simple reordering of the program code can lead to

[6]Note that this value divided by four does not result in the latency for one image. This is due to the round-robin scheduler, which allows that processing of an image might start before the previous one has been finished.

significant changes in the action execution times, which cannot be taken into account by the VPC framework.

Consequently, in order to obtain more precise simulation results an instruction set simulation taking caches and guard execution times into account would be necessary. Similarly a time-consuming hardware synthesis would be necessary in order to reduce the discrepancy between the VPC hardware estimations and the exact implementation results. This, however, is prohibitive during automatic design space exploration, since the number of solutions that can be investigated, and hence the possibility to find all optimal implementations, strongly decrease. Instead, we use the fast VPC simulations in order to find a manageable set of good implementations, which can then be further investigated by more precise simulation techniques. By providing an automatic synthesis path, SystemCoDesigner significantly simplifies this opportunity, and thus helps to evaluate a relatively large number of design points. This compensates for the fact that the VPC estimates might not be dominance-preserving.

9.1 Influence of Input Motion-JPEG Stream

As motivated in Section 3, determination of the best hardware configuration is a difficult task due to the data dependency of some of the involved actors, like the Huffman decoder. In other words, the achievable system throughput does not only depend on the chosen hardware implementation, but also on parameters such as image size and image quality, which is related to the compression ratio or file size.

The previous results have shown how the VPC framework can be used to quickly estimate the performance of a given implementation, based on accurate action execution times. The latter are derived by application profiling based on a particular Motion-JPEG stream, which is a laborious process. However, in contrast to other work, VPC not only simulates a fixed and constant execution time for each actor invocation, but an individual delay can be associated to each action. The latter should be far less dependent on the processed JPEG file than the actor invocations themselves. That means, performance data obtained from one JPEG stream can be reused for other streams as well, which significantly simplifies the evaluation of a given implementation, and which helps to efficiently estimate the expected execution times for given input streams. This is particularly beneficial for soft real-time systems containing data dependent decisions like the considered Motion-JPEG decoder. These scenarios are difficult to cover by WCET analysis methods, since the latter typically use more restricted models of computation than can be represented in SysteMoC. Consequently, largely overestimated WCET values would result for the entire application in the presence of data-dependent control flow.

Table VII shows the achieved results for a software-only implementation when processing different JPEG streams, whereas only the first one has been used to derive the actor execution times. The first two JPEG streams only differ by the encoded image quality and thus deliver different workloads for the data dependent actors like the *parser* and the *Huffman decoder*. The third JPEG stream uses a completely different image. In both cases, there is a pretty good

Table VII. Comparison Between Simulated and Measured Execution Times for Four Images and Different JPEG Streams

Input JPEG Stream	VPC Estimation	SW Implementation (without schedule overhead)	Rel. Error
QCIF (176 × 144) (5.5 kb)	31.54 s	31.76 s	0.7%
QCIF (176 × 144) (36.2 kb)	57.34 s	55.29 s	3.7%
Lena (256 × 256) (55.6 kb)	116.37 s	115.87 s	0.4%

match between the predicted and the measured values, thus allowing for highly accurate system evaluations. This is thanks to our action-accurate exploration, where the data dependency can be expressed in the actor's communication state machine. Consequently, it can also be taken into account during exploration, whereas the action execution times themselves ideally do not depend on the processed input image.

10. CONCLUSIONS

In this article, we presented a methodology to automatically optimize and generate an SoC—starting with an abstract actor-oriented model written in SystemC. Our design flow allows the automatic generation of highly optimized and accurately modeled SoC implementations, determined by advanced design space exploration, in very short time. Even for larger SoC designs, first working SoC implementations can be obtained from the SystemC behavioral source model within a few days or even hours. Moreover, by automatic generation, the synthesized system is *correct by construction*; no manual refinement steps are needed, and so a major source of errors is eliminated. This is achieved by integrating a state-of-the-art behavioral synthesis tool into a design space exploration tool. We demonstrated the efficiency of our design flow using an industrial grade case study, a Motion-JPEG decoder. The automatically generated hardware implementations run at 50 MHz and achieve more than 60 QCIF frames per second. No manual optimizations were made, which probably would further speed up the implementation.

Future work will focus on the improvement of the accuracy of our simulation models in order to better match the synthesis results.

REFERENCES

BALARIN, F., WATANABE, Y., HSIEH, H., LAVAGNO, L., PASSERONE, C., AND SANGIOVANNI-VINCENTELLI, A. 2003. Metropolis: an integrated electronic system design environment. *IEEE Comput. 36,* 4, 45–52.

Bluespec, Inc. http://www.bluespec.com.

Center for Embedded Computer Systems. http://www.cecs.uci.edu/.

CHANTRAPORNCHAI, C., SHA, E. H.-M., AND HU, X. S. 2000. Efficient design exploration based on module utility selection. *IEEE Trans. CAD Integ. Circ. Sys. 19,* 1, 19–29.

CriticalBlue. http://www.criticalblue.com.

FALK, J., HAUBELT, C., AND TEICH, J. 2006. Efficient representation and simulation of model-based designs in SystemC. In *Proceedings of the Forum on Design Languages*. Darmstadt, Germany.

Forte Design Systems. http://www.forteds.com.

GRÖTKER, T., LIAO, S., MARTIN, G., AND SWAN, S. 2002. *System Design with SystemC*. Kluwer Academic Publishers.

GUPTA, T., SHARMA, P., BALAKRISHNAN, M., AND MALIK, S. 2000. Processor evaluation in an embedded systems design environment. In *Proceedings of the 13th International Conference on VLSI Design*. 98–103.

HA, S., LEE, C., YI, Y., KWON, S., AND JOO, Y.-P. 2006. Hardware-software codesign of multimedia embedded systems: the PeaCE approach. In *Proceedings of the Conference on Embedded and Real Time Computing Systems and Applications (RTCSA)*. 207–214.

HAUBELT, C., FALK, J., KEINERT, J., SCHLICHTER, T., STREUBÜHR, M., DEYHLE, A., HADERT, A., AND TEICH, J. 2007. A SystemC-based design methodology for digital signal processing systems. *EURASIP J. Embed. Syst. (Special Issue on Embedded Digital Signal Processing Systems)*. DOI:10.1155/2007/47580.

HAUBELT, C., SCHLICHTER, T., AND TEICH, J. 2006. Improving automatic design space exploration by integrating symbolic techniques into multi-objective evolutionary algorithms. *Int. J. Comput. Intell. Res. 2,* 3, 239–254.

ITU. 1992. *Digital Compression and Coding of Continuous-Tone Still Images—Requirements and Guidelines*, T.81 ed. CCITT.

KANGAS, T., KUKKALA, P., ORSILA, H., SALMINEN, E., HÄNNIKÄINEN, M., HÄMÄLÄINEN, T. D., RIIHIMÄKI, J., AND KUUSILINNA, K. 2006. UML-based multiprocessor SoC design framework. *ACM Trans. Embed. Comput. Syst. 5,* 2, 281–320.

KIM, M., BANERJEE, S., DUTT, N., AND VENKATASUBRAMANIAN, N. 2006. Design space exploration of real-time multi-media MPSoCs with heterogeneous scheduling policies. In *Proceedings of the International Conference on Hardware/Software Codesign (CODES+ISSS)*. 16–21.

LEE, E. A. AND NEUENDORFFER, S. 2004. Actor-oriented Models for Codesign: Balancing Re-Use and Performance. In *Formal Methods and Models for System Design*. Kluwer Academic Publishers, Norwell, MA, 33–56.

LEE, E. A., NEUENDORFFER, S., AND WIRTHLIN, M. J. 2003. Actor-oriented design of embedded hardware and software systems. *J. Circ. Syst. Comput. 12,* 3, 231–260.

LEUPERS, R. 2000. *Code Optimization Techniques for Embedded Processors—Methods, Algorithms, and Tools*. Kluwer Academic Publishers.

MAMAGKAKIS, S., ATIENZA, D., POUCET, C., CATTHOOR, F., SOUDRIS, D., AND MENDIAS, J. M. 2006. Automated exploration of pareto-optimal configurations in parameterized dynamic memory allocation for embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. 874–875.

Mentor Graphics Corp. http://www.mentor.com.

MOHANTY, S., PRASANNA, V. K., NEEMA, S., AND DAVIS, J. 2002. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*. 18–27.

MURTHY, P. K. AND BHATTACHARYYA, S. S. 2006. *Memory Management for Synthesis of DSP Software*. CRC Press.

NEC System Technologies, Ltd. http://www.cyberworkbench.com.

NIKOLOV, H., STEFANOV, T., AND DEPRETTERE, E. 2006. Multi-processor system design with ESPAM. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*. 211–216.

PATEL, H. D., SHUKLA, S. K., MEDNICK, E., AND NIKHIL, R. S. 2006. A rule-based model of computation for SystemC: integrating SystemC and bluespec for co-design. In *Proceedings of International Conference on Formal Methods and Models for Co-Design*. 39–48.

PIMENTEL, A. D., ERBAS, C., AND POLSTRA, S. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput. 55,* 2, 99–112.

PPM format specification. http://netpbm.sourceforge.net/doc/ppm.html.

ROSENBAND, D. L. AND ARVIND. 2004. Modular scheduling of guarded atomic actions. In *Proceedings of the 41st Annual Conference on Design Automation*. 55–60.

SCHLICHTER, T., LUKASIEWYCZ, M., HAUBELT, C., AND TEICH, J. 2006. Improving system level design space exploration by incorporating SAT-solvers into multi-objective evolutionary algorithms. In *Proceedings of the Annual Symposium on VLSI*. IEEE Computer Society, 309–314.

STEFANOV, T., ZISSULESCU, C., TURJAN, A., KIENHUIS, B., AND DEPRETTERE, E. F. 2004. System design using kahn process networks: the compaan/laura approach. In *Proceedings of Design Automation & Test in Europe (DATE)*. 340–345.

STREHL, K., THIELE, L., GRIES, M., ZIEGENBEIN, D., ERNST, R., AND TEICH, J. 2001. FunState—an internal design representation for codesign. *IEEE Trans. Very Large Scale Integ. Syst. 9,* 4, 524–544.

STREUBÜHR, M., FALK, J., HAUBELT, C., TEICH, J., DORSCH, R., AND SCHLIPF, T. 2006. Task-accurate performance modeling in SystemC for real-time multi-processor architectures. In *Proceedings of Design, Automation and Test in Europe*. IEEE Computer Society, 480–481.

Synplicity. http://www.synplicity.com.

XILINX 2005. *Embedded SystemTools Reference Manual—Embedded Development Kit EDK 8.1i*. XILINX.

ZITZLER, E., LAUMANNS, M., AND THIELE, L. 2002. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Proceedings of the Conference on Evolutionary Methods for Design, Optimisation, and Control*. 19–26.

Received August 2007; revised March 2008; accepted July 2008