Automatic Generation of System-Level Virtual Prototypes from Streaming Application Models

Philipp Kutzer, Jens Gladigau, Christian Haubelt, and Jürgen Teich
Hardware/Software Co-Design, Department of Computer Science
University of Erlangen-Nuremberg, Germany
Email: {philipp.kutzer, jens.gladigau, haubelt, teich}@cs.fau.de

***Abstract*—Virtual prototyping is a more and more accepted technology to enable early software development in the design flow of embedded systems. Since virtual prototypes are typically constructed manually, their value during design space exploration is limited. On the other hand, system synthesis approaches often start from abstract and executable models, allowing for fast design space exploration, considering only predefined design decisions. Usually, the output of these approaches is an "ad hoc" implementation, which is hard to reuse in further refinement steps. In this paper, we propose a methodology for automatic generation of heterogeneous MPSoC virtual prototypes starting with models for streaming applications. The advantage of the proposed approach lies in the fact that it is open to subsequent design steps. The applicability of the proposed approach to real-world applications is demonstrated using a Motion JPEG decoder application that is automatically refined into several virtual prototypes within seconds, which are correct by construction, instead of using error-prone manual refinement, which typically requires several days.**

I. Introduction

Today, modern Multi-Processor System-on-Chip (MPSoC) architectures consist of a mixture of microprocessors, digital signal processors (DSPs), memory subsystems, and hardware accelerators, as well as interconnect components. It is noticeable that the adoption of programmable logic in such electronic systems is more and more increasing. Driven by this rise, the process of software development becomes the dominating part during system design. In the course of software development, software engineers have to cope with operating systems, communication stacks, drivers, and so forth. In order to allow early software development, virtual prototyping is a more and more frequently used technology in Electronic System Level (ESL) design. There, the desired target platform is modeled as an abstract, executable, and often completely functional software model. Hence, the virtual prototype includes all functional properties of the target platform, while non-functional properties, such as timing behavior, are mostly disregarded.

In contrast to FPGA-based prototyping, virtual prototypes are deployed before architectural models on register-transfer-level are available. Due to this early availability, the overall time spent on hardware and software design can be reduced, because software can be implemented, refined, tested, debugged, and verified on realistic hardware models in parallel to the hardware design process. Nevertheless, additional time is needed for implementing such prototypes from the functional and desired architectural system specification. This drawback could be avoided with an automatic virtual prototype generation. This would further speed up the design process and in addition, errors, often made in manual prototype generation, are avoided.

Supported in part by the German Science Foundation
(DFG Project HA 4463/3-1)

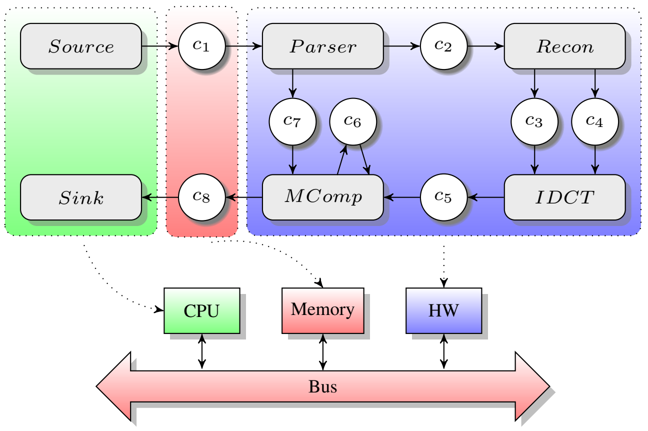

Fig. 1. Application model of a Motion JPEG decoder, clustered and mapped to an architecture template. The architecture template consists of a CPU, a hardware accelerator (HW) and an external memory. All the components are connected via a bus.

Describing a complex application abstracted as an actor-oriented model [1] is a more and more accepted approach in ESL design. Such models are used to describe the functional behavior of the application. Therefore, they consist of concurrently executing actors, which communicate over abstract channels. In our approach, the communication takes place via channels with FIFO semantics. An example is shown in Fig. 1 for a small actor-oriented model, a Motion JPEG decoder, which consists of the actors Source, Parser, Reconstruction (Recon), Inverse Discrete Cosine Transformation (IDCT), Motion Compensation (MComp), and Sink, as well as FIFO channels c_1 to c_8. In order to generate a virtual prototype starting with an actor-oriented model, additional information about the system architecture candidates and the mapping possibilities of the functional components have to be specified. In the lower part of Fig. 1, a possible mapping to an architecture template is given by the dotted arrows.

In the following, we present a method for automatic generation of MPSoC virtual prototypes from actor-oriented models. Our proposed approach performs the virtual prototype generation in two steps: (i) Based on a given resource mapping, communication within the application model is refined to transactions in the virtual prototype, and controllers for intra-resource communication are generated. (ii) The virtual prototype is generated by assembling cycle-accurate processor models, memory models, and models for hardware accelerators using bus models, and synthesizing the software for each processor, according to the given mapping.

The remainder of this paper is structured as follows: Section II reflects related work. In Section III, a brief overview of our approach is given. Section IV describes application modeling. In Section V, the automatic generation of architectural TLM models is discussed in more detail. Section VI describes the architectural refinement in more detail. Section VII presents experimental results from applying the proposed prototype generation approach to a Motion JPEG decoder, a multimedia streaming application mapped onto an MPSoC architecture. Finally, conclusions are given in Section VIII.

II. Related Work

As virtual prototypes are nowadays commonly used in system-level design flows, several commercial as well as free of charge tools exist to build, simulate and evaluate such prototypes. Most prominent are Platform Architect from CoWare, CoMET from VaST and OVPsim [2] from Imperas. The two first mentioned tools were acquired by Synopsys [3] within the last year. Most existing virtual prototyping tools support the integration of transaction level models written in SystemC [4] in the prototypes. However, none of them allows the automatic transformation of a formal description, like an actor-oriented model, to a virtual prototype.

In general, mapping formal models on MPSoCs is a current research topic in system synthesis (e.g., see [5], [6]). There exist several system-level synthesis tools that automatically map formal described applications to a MPSoC target, like Daedalus [7], Koski [8], and SystemCoDesigner [9]. All these approaches want to achieve a common purpose. They target final product generation. This means, they have to cover the complete design flow, starting with an high-level application specification down to the running system. Caused by this, their integration into existing design flows is hard to establish.

In contrast to system synthesis tools, our proposed approach targets automatic virtual prototype generation. In this scenario, important design decisions are reflected in the generated prototype, while support for further manual refinement is retained. Hence, the product quality still could be influenced by a designer and, even more important, our proposed approach could be easily integrated in established industrial design flows.

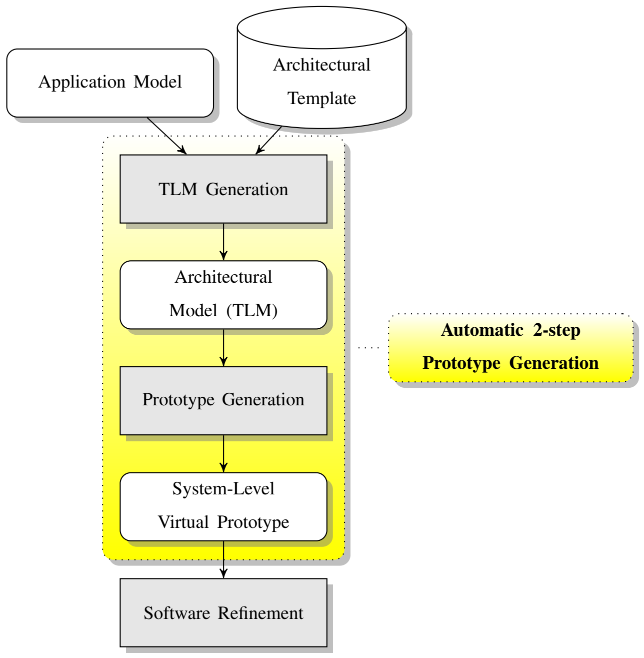
Application Model
Architectural Template
TLM Generation
Architectural Model (TLM)
Prototype Generation
System-Level Virtual Prototype
Software Refinement
Automatic 2-step Prototype Generation

Fig. 2. Design flow from an application model, represented by an abstract executable specification, to a virtual prototype. The flow includes automatic mapping of actor-oriented models to TLM architecture models, as well as virtual prototype generation.

III. Virtual Prototype Generation - Overview

The goal of our system-level design approach is to automatically implement abstract system descriptions written in SystemC as virtual MPSoC prototypes. The associated design flow is depicted in Fig. 2.

At the beginning of our ESL design process, an abstract model has to be derived for the desired application. In our approach, a distinction is drawn between the application model, which describes the functional behavior of the system, and the architecture template, which represents all architecture instances of the system.

The system behavior is modeled in form of actor-oriented models, which only consist of actors and channels, as depicted in the Motion JPEG example from Fig. 1. Actors are the communicating entities, which are executed concurrently. For communication, tokens are produced and consumed by actors, and transmitted via dedicated channels.

The architecture template of the system is represented by a heterogeneous MPSoC platform, which is specified by connected cores. Single actors or clusters of actors can either be mapped onto processor elements (CPU) or on dedicated hardware accelerators (HW), as depicted in Fig. 1. Hardware accelerators will typically be used for computationally intensive or time critical parts of the application. In general, System-on-Chips include both processor elements as well as hardware accelerators. Depending on the actor mapping, communication channels can either be mapped on internal memory of data processing units (CPU or HW accelerators), or on shared memory modules. In the Motion JPEG decoder example, all channels except c_1 and c_8 are mapped to the hardware accelerator, as communication takes place internally. Channels c_1 and c_8 represent the communication between the CPU and the dedicated accelerator, and hence have to be mapped to the shared memory.

After modeling the application, the architecture template, and defining a mapping of functional to structural elements, an architectural model will be automatically generated. In this intermediate model, the actors are clustered according to the mapping on architectural resources. Due to the fact that virtual prototypes are usually implemented using transaction level modeling (TLM), we use the OSCI TLM-2.0 [10] standard in our design flow.

For virtual prototyping, parts of the architectural model are subsequently replaced by the corresponding resources from a virtual component library, which consist of cycle-accurate processor models, as well as models of communication entities. Beside the architectural refinement, software is generated and cross-compiled for each CPU, to match its instruction set architecture (ISA).

The resulting virtual prototype can now be used for further software and communication refinement. Moreover, due to the cycle-accurate processor models, performance estimation becomes possible. The steps of architectural mapping as well as prototype generation will be described later in more detail. First, our application modeling approach is described.

IV. Application Model

This section introduces our concept of actor-oriented modeling, which is necessary to understand our proposed mapping approach. In actor-oriented models, actors are potentially executed concurrently and communicate over dedicated abstract channels. Thereby, they produce and consume data (so called *tokens*), which are transmitted by those channels. These models may be represented as bipartite graphs, consisting of channels $c \in C$ and actors $a \in A$. In the following, we use the term network graph for this kind of representation.

Definition 1 (network graph): A *network graph* is a directed bipartite graph $G_n = (A, C, P, E)$, containing a set of actors A, a set of channels C, a channel parameter function $P : C \rightarrow \mathbb{N}_\infty \times V^*$ that associates with each channel $c \in C$ its buffer size $n \in \mathbb{N}_\infty = \{1, 2, 3, .., \infty\}$, and also a possibly nonempty sequence $v \in V^*$ of initial tokens, where V^* denotes the set of all possible finite sequences of tokens $v \in V$. Additionally, the network graph consists of directed edges $e \in E \subseteq (C \times A.I) \cup (A.O \times C)$ between actor output ports $o \in A.O$ and channels, as well as channels and actor input ports $i \in A.I$.
An example of a network graph is already given in the upper part of Fig. 1.

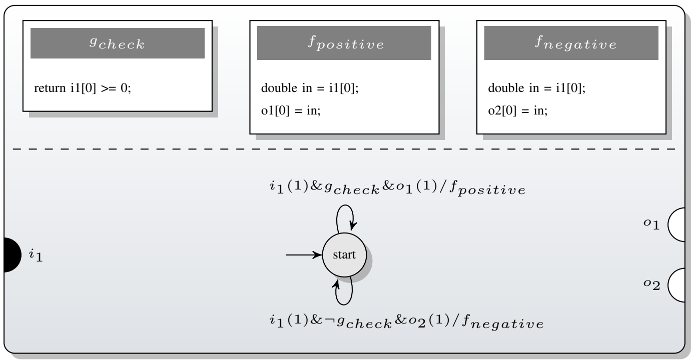
```
return i1[0] >= 0;
```
```
double in = i1[0];
o1[0] = in;
```
```
double in = i1[0];
o2[0] = in;
```
g_{check}
$f_{positive}$
$f_{negative}$
$i_1(1)\&g_{check}\&o_1(1)/f_{positive}$
start
i_1
o_1
o_2
$i_1(1)\&\neg g_{check}\&o_2(1)/f_{negative}$

Fig. 3. Visual representation of an actor, which sorts input data according to its algebraic sign. The actor consists of one input port i_1 and two output ports o_1 and o_2.

Definition 2 (Channel): A *channel* is a tuple $c = (I, O, n, d)$ containing channel ports partitioned into a set of channel input ports I and a set of channel output ports O, its buffer size $n \in \mathbb{N}_\infty = \{1, 2, 3, .., \infty\}$, and also a possibly empty sequence $d \in D^*$ of initial tokens, where D^* denotes the set of all possible finite sequences of tokens $d \in D$.
In the following approach, we will use SysteMoC [11], a SystemC [4] based library for modeling and simulating actor-oriented models. In the basic SysteMoC model, each channel is an unidirectional point-to-point connection between an actor output port and an actor input port, i.e. $|c.I| = |c.O| = 1$. The communication between actors is restricted to these abstract channels, i.e. actors are only permitted to communicate with each other via channels, to which the actors are connected by ports.

In a SysteMoC actor, the communication behavior is separated from its functionality. The communication behavior is defined as finite state machine (FSM); the functionality is a collection of functions that can access data on channels via ports. These functions are classified in *actions* and *guards*, and are driven by the finite state machine (FSM). So SysteMoC follows the FunState [12] (Functions driven by State machines) approach.

An action of an actor is able to access data on all channels, the actor is connected to, and is allowed to manipulate the internal state of the actor implemented by internal variables. In contrast, a guard function is only allowed to query, not to alter neither the internal state nor the data on channels. A graphical representation of a SysteMoC actor is given in Fig. 3. The actor *Sorter*, which is used to sort input data tokens according to algebraic sign, possesses one input port (i_1) and two output ports (o_1 and o_2). Tokens from input port i_1 will be forwarded to output port o_1 by the function $f_{positive}$, if the activation pattern $i_1(1)\&g_{check}\&o_1(1)$ of the state transition from the state $start$ to the state $start$ evaluates to true. This pattern determines under which conditions the transitions may be taken. In SysteMoC, the activation pattern can depend on

```
class Sorter : smoc_actor {
public:
  smoc_port_in<double>  i1;
  smoc_port_out<double> o1;
  smoc_port_out<double> o2;
  smoc_firing_state start;
  Sorter(sc_module_name name) : smoc_actor(name,
      start) {
    start =
        (i1(1) && GUARD(check) && o1(1))   >>
        CALL(positive)                     >> start
      |
        (i1(1) && !GUARD(check) && o2(1)) >>
        CALL(negative)                     >> start;
  }
private:
  bool check(void) const {
    return i1[0] >= 0;
  }
  void positive(void) {
    double in = i1[0];
    o1[0] = in;
  }
  void positive(void) {
    double in = i1[0];
    o2[0] = in;
  }
};
```

Listing 1. SysteMoC code for the actor Sorter. The FSM of the actor is defined in the constructor of the actor class, whereas the functionality is encoded as private member functions.

some internal state of the actor, on availability and values of tokens on input channels, and on availability of free space on output channels. In our example, the state transition will be taken, if at least one token is available on input port ($i_1(1)$), the guard g_{checks} evaluates to true (data on input channel has positive algebraic sign), and output port o_1 has space for at least one additional token ($o_1(1)$). Analog to this, the second transition is taken if input data is negative. The corresponding SysteMoC code is given in Listing 1.

To summarize, the transition-based execution of SysteMoC actors can be divided into 4 steps: (i) Evaluation of all activation patterns k of all outgoing state transitions in the current state $q_c \in Q$. (ii) Non-deterministically selecting and taking of one activated transition $t \in T$. (iii) Execution of the corresponding action $f \in a.F$. (iv) Notification of token consumption/production on channels connected to corresponding input and output actor ports after completion of action as well as transition to the next state.

During system synthesis from actor-oriented models, actors $a \in A$ and the communication channels $c \in C$ are mapped to components of a system architecture. To reflect architectural structure in network graphs after mapping, nodes can be clustered. For representation of clustering, we define a clustered network graph.

Definition 3 (clustered network graph): A *clustered network graph* $G_{cn} = (G_n, T)$ consist of a network graph G_n and a rooted tree T such that the leaves of T are exactly the vertices of G_n. Each node x of T represents a cluster $X(x)$ of the vertices of network graph G_n that are leaves of the subtree

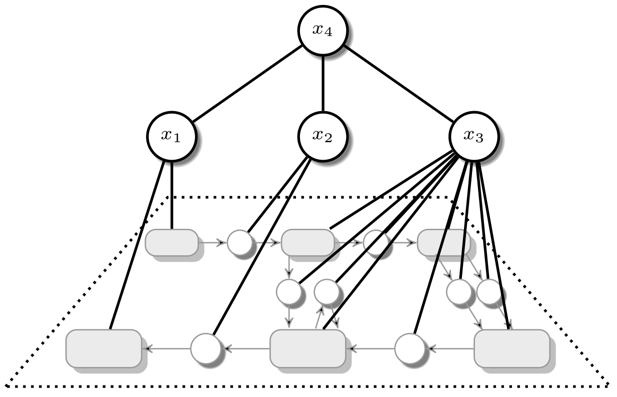

Fig. 4. Clustered network graph of the Motion JPEG example. The cluster $X(x_1)$ represents the CPU, $X(x_2)$ the communication bus and $X(x_3)$ the hardware accelerator. Cluster $X(x_4)$ represents the whole system.

rooted by x.
The representation as tree illustrates the hierarchical structure of the system. This means, the root of T represents the whole system, whereas nodes $x \in T$ with $height(x) = 1$ represent the components of the system. As reuse of parts of models is common in the design process, hierarchical structures with more than two levels are possible. The clustered network graph of the example from Fig. 1 is depicted in Fig. 4.

Although we used SysteMoC, our approach is not restricted to this framework and can be adapted to other frameworks for actor oriented design, e.g. pure SystemC FIFO channel communication. A deeper insight into SysteMoC is given in [11].

V. Generating the TLM Architecture

Transaction level modeling (TLM) with SystemC has become apparent as de-facto industry standard for virtual prototyping and architectural modeling [13], [14]. These models are characterized by an encapsulation of low-level communication details. Due to abstraction, very fast simulation speed can be achieved. To enable fast simulation, details of bus-based communication protocol signaling are replaced with single transactions. In the course of releasing a TLM standard (OSCI TLM-2.0) to enforce interoperability of models, the Open SystemC Initiative defined two coding styles [15]: the loosely-timed (LT) and the approximately-timed (AT) coding style. The loosely-timed coding style allows only two timing points to be associated with each transaction, namely the start and the end of the transaction. This timing granularity of communication is sufficient for software development using a virtual prototype model of an MPSoC. A transaction in an approximately-timed model is broken down into multiple phases, with timing points marking the transition between two consecutive phases. Due to the finer granularity of timing, approximately-timed models are used typically in architectural exploration and performance analysis. As our approach targets software development, or more precisely the refinement of parts of the application in software, by means of virtual prototyping, the loosely-timed coding style is adequate [15].

As described, actors $a \in A$ and the communication channels $c \in C$ are partitioned to clusters $X(x)$ and mapped to components of a system architecture. Due to the mapping, the channel communication can either be internal, in case both communicating actors a_a and a_b mapped onto the same resource ($a_a \in X(x_y)$ and $a_b \in X(x_y)$), or external, in case communication crosses cluster boundaries ($a_a \in X(x_y)$ and $a_b \notin X(x_y)$). For intra-resource communication, FIFOs can be put in private memory of the architectural component, whereas FIFOs of inter-resource communication, like c_1 and c_8 from Figure 1, have to be placed in an external memory model. Either way, actor communication semantics through ports are not altered, in order to reuse the existing actors written in SystemC based SysteMoC. So, the challenge of this step in design flow is to map the FIFO-based communication via dedicated channels to a memory-mapped bus-based communication with global and local shared memory. Since our abstract communication semantics (read, write, commit) calls for uniform channel access, access transparency has to be ensured after mapping to architectural template, resulting in the transaction level architectural model. As communicating actors on different resources are concurrently executed, simultaneous access to FIFO storage has to be avoided. This means that memory coherence as well as cache coherence has to be guaranteed. To cope with actor clustering and to ensure synchronized channel access, independent of communication mapping, we use aggregators and adapters in our approach that implement a suitable communication protocol [16]. Adapters, by which the SysteMoC ports ($i \in A.I$ and $o \in A.O$) are substituted, serve as links between the actors and the transaction level. Due to the fact that more than one actor can be mapped to one resource, and actors can possess multiple ports, an aggregator is needed for each transaction level component ($X(x_i) : height(x_i) = 1$) to encapsulate the desired number of adapters. These aggregators perform transaction level communication and implement the interface of the component to the rest of the architectural model. There is no need to connect adapters for internal channels with the aggregator, because no communication will take place over component boundaries. In Fig. 1, communication between the actors Parser, Recon, MComp and IDCT is internal and can be implemented using, e.g., internal memory. In our approach, we use a transaction level memory model for each communication channel. In the following, we will describe the functionality of adapter and aggregator in more detail.

A. Adapter

An adapter adapts between transactions in the virtual prototype and the asynchronous FIFO channel communication used in the application model. Hence, the communication adapter implements two different interfaces. The interface towards the actor is equivalent to the abstract channel, which has to be replaced. To sustain abstract communication semantics, the adapter needs to access tokens in a random manner and to commit completed transitions via this interface. Therefore, a conversion of the token data type (e.g., serialization and deserialization) has to be performed in adapters. An adapter also has to respect the abstract channel synchronization mechanism. This means, the adapter has to provide an interface through which the adapter can be notified when tokens on channel are produced or consumedi, respectively. This notification can be used to trigger the corresponding actor waiting for free space or tokens on channel.

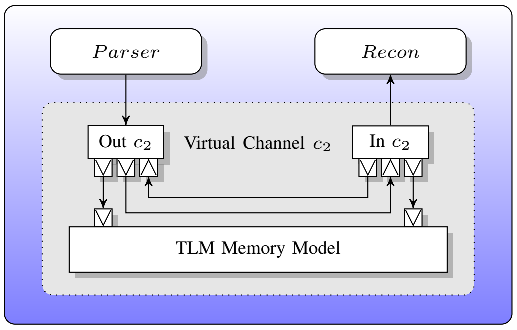

Fig. 5. Mapping of parts of cluster $X(x_3)$ from the model, depicted in Fig. 1, to a architectural component. The internal communication takes place over a virtual channel, which substitutes the abstract channel. Therefore, adapters adapt between the abstract model and the transaction level model. The FIFO queue semantics are implemented using a TLM memory model.

The transaction level interface consist of three transaction level communication sockets (see Fig. 5). One is used for data transmission. The actor, which is connected to the adapter, can read or write data from a memory through this socket. The other two sockets are needed to sustain the channel synchronization. For synchronization, the adapters communicate among each other over arbitrary TLM communication resources. Therefor, a dedicated address has to be assigned to each adapter.

Due to the fact that the SysteMoC channels possess memory, the FIFO storages have to be mapped to resources. As different locations are possible, we allocate the storage in a memory, to which the adapters are connected to. For internal communication, the sockets of the adapters can be directly coupled with each other, as depicted in Fig. 5. The synchronization sockets of the two communicating adapters are directly coupled, whereas the data sockets are connected with the memory.

The memory of external communication is accessible over a bus system, to which the aggregator is connected (see Fig. 6). Allocation of storage in one adapter or splitting and distributing the storage over both communicating adapters is also possible. Independent of the chosen implementation and mapping, each adapter needs to know to which address space his buffer is mapped to, in order to read or write tokens.

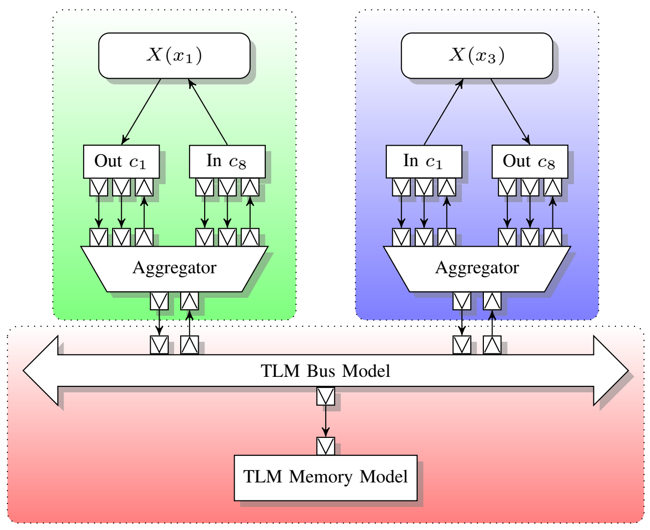

Fig. 6. Mapping of the cross component communication between HW and CPU from Fig. 1. For the sake of clarity, internal communication structure is omitted.

B. Aggregator

As real computational resources like CPUs or DSPs have a limited number of connection pins, each node $x \in T$ besides the root node needs a mechanism that aggregates the children connected to x. For nodes that represent data transferring units, like buses (x_2), this is done by arbitration and address translation. Unlike the communication resources (data transferring units), the computational resources (data processing units) need an aggregator for this purpose. The aggregators contain TLM ports to perform transaction level cross component communication. Therefore, they implement the communication protocol for the connected adapters at the transaction level. For communication, aggregators communicate among each other over arbitrary TLM communication resources. For this purpose, each aggregator is assigned a dedicated address-range. Its size depends on the number of adapters registered to the aggregator. So each adapter is assigned a single address, to which it is accessible for event-based synchronization. Beside his own address range, each aggregator has to know addresses of peer adapters, which are associated with registered adapters, and addresses of the corresponding FIFOs in memory.

VI. Virtual Prototype Generation

In the final step of our automatic design flow, a virtual prototype is generated based on the transaction level architectural model.

A. Architectural Refinement

In order to allow for an early software development, parts of the architecture have to be substituted by virtual component models. In our approach, all resources except the hardware accelerators are replaced. As our approach focuses on software development, the inserted processor models must provide an instruction set simulator, in order to simulate or furthermore debug the software running on the models. Therefore, we use a commercial virtual component library [3], which provides the opportunity to integrate TLM. This feature is necessary to couple the hardware accelerators with the virtual components. In order to sustain the abstract channel synchronization mechanism, an interrupt controller is added for each processor element. By the use of this controller, the processor element can be informed about channel data modification by another processor or hardware accelerator.

B. Target Software Generation

During the process of target software generation, the actor description in SystemC is transformed into standard C/C++ code. Therefore, the ports for communication of the actor are replaced by pointers to FIFO interfaces, and the finite state machine is encoded as switch-case statement. The FIFO interfaces represent the communication interface equivalent to the TLM communication adapters, described in Section V. Moreover, scheduling strategies have to be implemented, in case multiple actors are mapped on the same processor element.

TABLE I
Measurement terms of the 5 different virtual prototypes.

VP	Instructions	Simulation		VP Performance	
		Host[s]	VP[ms]	CPI	MIPS
I	4944835683	1997	44285	1.79	111.66
II	5319738192	521	30494	1.15	174.45
III	5726625319	1791	29222	1.02	195.97
IV	5765601708	660	26993	0.94	213.59
V	6188808202	1760	7224	0.23	856.66
VI	3492102237	550	30870	1.77	113.12

VII. Experimental Results

In order to show the applicability of our approach, we present our first results on generating virtual prototypes from an actor-oriented Motion-JPEG model. Therefore, we use a more fine-grained model than given in Fig. 1, which consists of 19 actors, interconnected by a total of 56 FIFO channels. In Table I, the results of several test cases in terms of different mappings are presented. Since the architecture template contains 19 processors, 19 hardware accelerators, and a shared memory, which all are connected by bus, many architecture instances exist. With our approach, it is possible to generate virtual prototypes from all of them. To show the applicability, we consider only a few mappings serving as representatives.

Our first prototype (`I`) consists of a single processor (ARM926), onto which all actors are mapped. For the next two test cases, two processors are allocated and connected via a bus. For this architecture instance, two mappings are tested, respectively: (i) The IDCT actors are mapped to one processor, all remaining actors to the other one (`II`). (ii) The actors are mapped to the processors alternately, i.e. the

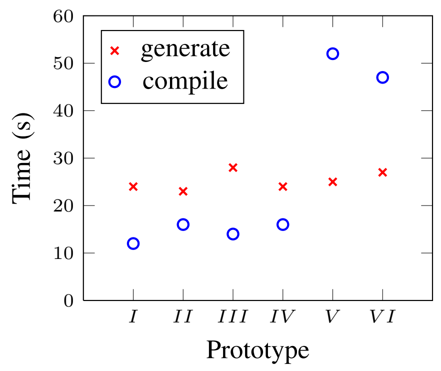

Fig. 7. Times measured for generation and compilation of the different configurations.

neighbor of each actor in the decoding pipeline is mapped to the processor different than the processor to which the actor itself is mapped (`III`). For the FIFO communication between the two processors, a memory is additionally allocated and connected to the bus. In the fourth prototype (`IV`), three processors and a memory are allocated. Here, actors Source and Sink are clustered to one processor, IDCT is mapped to the second one, and the remaining actors are mapped to the third. To take the full advantage of pipelined execution, 19 processors are allocated in the fifth prototype (`V`). In the last test case, `VI`, one processor and one hardware accelerator are allocated. This test case is analog to the second prototype, except the functionality of the IDCT actors is swapped to the hardware accelerator.

Figure 7 shows the time needed for prototype generation and compilation. It can be seen that the time spent for prototype generation is nearly independent of the mapping, whereas the time for compiling depends on the components of the prototype. On the one hand, it is obvious, that the more processors are allocated, the more time is needed for compiling. On the other hand, the code for the transaction level hardware accelerators is more complex than the code running on processors, so more time is needed for compiling hardware accelerators. However, in summary it can be seen that all virtual prototypes have been generated within seconds, instead of hours. In the following, 5 measurement terms will be tested in order to decode 10 images (176x144): total instructions executed; cycles per instruction (CPI); million instructions per second (MIPS); simulation time (host time); simulated time. In order to make a statement of system performance, not of simulator performance, the terms CPI and MIPS relate to the simulated time. The corresponding values are given in Table I.

It can be seen that the performance of the prototypes behave as expected. The more processors are allocated, the better the pipeline of the decoder can be exploited. This means less cycles are needed for one instruction, what causes a higher MIPS and a lower CPI rate. The small difference between `II` and `III` is based on a better workload distribution.

As different developer teams implement different parts of the application, it is often unneeded to refine all components of the TLM architectural model to virtual processor models. Prototype `VI` shows that there is no appreciable difference in simulated and host time in contrast to the completely refined model (`II`).

VIII. Conclusion

In this paper, we have presented a two-step methodology for automatically generating virtual system-level prototypes from an abstract system specification. Our main goal was to provide a methodology to remove the dependency on hardware availability, needed for software development, in an early phase of the design flow, which starts with an abstract and executable application model. For this purpose, design decisions are first represented in SystemC TLM, which is typically supported by all commercial virtual prototyping tools. Second, the TLM generation is used to assemble the virtual prototype and generate the embedded software. To show the applicability of our approach to real-world applications, we presented first simulation results for an actor-oriented Motion JPEG model.

References

[1] E. A. Lee, "Overview of the ptolemy project, technical memorandum no. ucb/erl m03/25," Department of Electrical Engineering and Computer Science, University of California, Berkely, CA, USA, Tech. Rep., Jul. 2004.

[2] OVPworld, http://www.ovpworld.org.

[3] Synopsys, http://www.synopsys.com.

[4] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[5] O. Moreira, F. Valente, and M. Bekooij, "Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor," in *Proceedings of EMSOFT*, 2007, pp. 57–66.

[6] P. K. F. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. M. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSoC)," in *Proceedings of DATE*, 2008, pp. 212–217.

[7] M. Thompson, T. Stefanov, H. Nikolov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," in *Proceedings of CODES+ISSS*, 2007, pp. 9–14.

[8] T. Kangas et al., "UML-based multi-processor SoC design framework," *ACM TECS*, vol. 5, no. 2, pp. 281–320, May 2006.

[9] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SYSTEMCODESIGNER - An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications," *TODAES*, vol. 14, no. 1, pp. 1–23, 2009.

[10] Open SystemC Initiative (OSCI)., "OSCI SystemC TLM 2.0," http://www.systemc.org/downloads/standards/tlm20/.

[11] J. Falk, C. Haubelt, and J. Teich, "Efficient representation and simulation of model-based designs in systemc," in *Proceedings of FDL*, Sep. 2006, pp. 129–134.

[12] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "Funstate—an internal design representation for codesign," in *Proceedings of ICCAD*. Piscataway, NJ, USA: IEEE Press, 1999, pp. 558–565.

[13] F. Ghenassia, *Transaction-Level Modeling with SystemC*. Dordrecht: Springer, 2005.

[14] B. Bailey and G. Martin, *ESL Models and their Application*. Dordrecht: Springer, 2010.

[15] *OSCI TLM-2.0 user manual*, Open SystemC Initiative, Jun. 2008.

[16] J. Gladigau, C. Haubelt, B. Niemann, and J. Teich, "Mapping actor-oriented models to TLM architectures," in *Proceedings of Forum on specification and Design Languages, FDL 2007*, Barcelona, Spain, Sep. 2007, pp. 128–133.