The ODYSSEY approach to early simulation-based equivalence checking at ESL level using automatically generated executable transaction-level model

Maziar Goudarzi *, Shaahin Hessabi, Naser MohammadZadeh, Nasim Zainolabedini

Department of Computer Engineering, Sharif University of Technology, Azadi Avenue, Tehran, Iran

A R T I C L E   I N F O

Article history:
Available online 14 April 2008

Keywords:
Hardware–software co-simulation
Simulation-based verification
Transaction-level model
ODYSSEY methodology
Hardware–software co-design

A B S T R A C T

Design technology is expected to rise to electronic system-level (ESL). This necessitates new techniques and tools for synthesizing ESL designs and for verifying them before and after ESL synthesis. A promising verification strategy for future very complex designs is to initially verify the design at the highest level of abstraction, and then check the equivalence of the lower level automatically generated models against that initial golden model. We present one such approach to simulation-based functional verification implemented in our ESL design methodology called ODYSSEY. Our ESL synthesis tool generates a transaction-level model (TLM) at TLM level 2 (i.e., design with partial timing) that corresponds to the input ESL design (which is at TLM level 3; i.e., sole functionality without timing). Both the ESL design and its generated TLM model can be simulated on a host machine with corresponding input stimuli to establish their functional equivalence. The TLM is in SystemC, and hence executable, and also models both hardware and software components in C++ to achieve higher simulation speed. We introduce an implementation of a TLM level 2 model that is tailored to our ESL design methodology and apply our approach to a number of benchmarks to evaluate the TLM simulation performance. Experimental results show that the approach suits early validation of the ESL synthesis process since its simulation performance is more than 4 orders of magnitude higher than simulations at lower levels and it is generated early in the design cycle. Also the co-simulation overhead – compared to simulating the original ESL design in C++ – depends on the partitioning quality in terms of communication to computation ratio.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Starting the design from higher levels of abstraction is inevitable to excel the designers’ productivity [1]. The industry trend has already been toward electronic system-level (ESL) design [2], where software and hardware of the system are co-designed. In such a design flow, the design process starts from an implementation-independent system model that is later partitioned and elaborated to hardware and software components. This elaboration/partitioning, which we hereafter refer to as ESL synthesis, is an essential part of ESL design, and furthermore, is difficult to verify since the pre- and post-synthesis models are in different semantic domains; the pre-synthesis model is implementation-independent whereas the post-synthesis model is a heterogeneous combination of sequential functional software and parallel structural hardware. In this paper we present our approach to tackle this problem in ODYSSEY project [11].

The International Technology Roadmap for Semiconductors (ITRS) suggests that future design technologies should facilitate verification of the system-under-design much earlier in the design flow to avoid costly design iterations from lower levels [1]. Fig. 1 shows past, present, and envisioned future design system architectures as viewed by ITRS. In the past (left hand side of the figure), only the operations from register transfer level (RTL) design down to the final implementation employed equivalence checking (look at the EQ Check rectangle at the bottom left of the figure) while various tools and multiple design files were generated and used for hardware synthesis; hardware/software partitioning and optimization were also manual tasks accomplished by expert system designers (look at HW/SW optimization cloud at middle left of Fig. 1), and moreover, software optimization was a separate process from hardware optimization procedure (see the separate SW opt box at left-middle of Fig. 1). At present (middle part of the figure), hardware/software partitioned model is generated from the high-level system model (compare the top parts of past and present in the figure) which enables some degree of design-space explora-

* Corresponding author. Present address: System LSIs Research Center of Kyushu University, Fukuoka, Japan. Tel.: +81 92 847 5193; fax: +81 92 847 5190.
E-mail addresses: goudarzi@sharif.edu (M. Goudarzi), hessabi@sharif.edu (S. Hessabi), naser@ce.sharif.edu (N. MohammadZadeh), n_zainolabedini@ce.sharif.edu (N. Zainolabedini).
tion; furthermore in the hardware optimization process, multiple design files are converged into one efficient data model that removes many tool interoperability issues and enables efficient iterative optimization methodologies during optimizations from RTL level down to the final implementation (look at the bottom-middle of Fig. 1 and compare it to the past case). In future (right hand side of the figure), software optimization and hardware/software partitioning and optimization are to be combined with hardware optimization process (look at the bottom of the flow) and more importantly, verification tasks all move to earlier, higher levels of abstraction (look at the box labeled “functional, performance, testability verification” at the top-right of Fig. 1) followed by equivalence checking and assertion-driven design optimization (see the third bullet in the text box at bottom-right of Fig. 1). In other words, this vision of ITRS suggests [1] that with the technology move to higher abstraction levels, modern equivalents are sought for the equivalence checking process that is currently done at logic and register transfer level (RTL).

Solutions to this verification and validation problem fall into two main categories of formal-verification techniques and simulation-based techniques [3,4]; we follow the latter approach. Simulation-based approaches, sometimes instrumented with assertions, are more popular due to easier use and more intuitiveness. The main concern in such approaches is the simulation speed. Two approaches exist for such simulations [5]: (i) interpretive simulation where the simulator, which is already compiled for the host machine, reads in a model of the system-under-simulation and simulates it, and (ii) compiled simulation where the model of the system-under-simulation is itself compiled to execute on the host machine and the results of this execution are the simulation results of the system-under-simulation. We follow the latter since compiled simulation provides the highest speed compared to other approaches because its resulting executable is (i) customized to the system-under-simulation and (ii) optimized for the host computer running the simulation. The general idea of compiled simulation of hardware and logic alone [5] and also compiled co-simulation of software along with hardware [6,7] are not new and have been extensively reported and used in the literature, however, it is new and required by the technology trend toward system-level design that (i) tailor co-simulation model to the specific needs of ESL design flow; (ii) automatically generate it in the design flow, and (iii) realize ITRS-envisioned verification flow at ESL level; these are important contributions of this work and our experimental results show that they successfully reduce the validation time and

Fig. 1. ITRS-envisioned required evolution of design system architecture [1].
increase the designer productivity. In this paper we present a transaction-level model (TLM) [6] in SystemC [7] for the hardware, software, and interface components that our ESL-synthesis tool generates from a given C++ application which is the starting point in our ESL design flow. We compile and run this TLM model to functionally validate the post-synthesis model and make sure that pre- and post-synthesis models are functionally the same. Our co-simulation model works early in the design cycle at orders of magnitude higher speed compared to co-simulation at lower levels of abstraction. Consequently, comprehensive simulation runs are possible before further elaborating now-partitioned hardware and software components of the system.

This paper is an extension of our previous work [8] describing an ESL design framework starting from object-oriented (OO) C++ source code of the target application. Here we have added the following investigations:

- Detailed explanation of the co-simulation model, its components, and the interactions among them.
- Description of the verification and validation procedure that we follow at ESL level.
- Presentation and analysis of experimental results on processing and co-simulating several benchmarks using our ESL synthesis tool and our implemented validation approach along with comparison to conventional co-simulation techniques.
- Examples and explanations of the design flow that we have proposed.

The rest of this paper is organized as follows. Section 2 presents a motivational example to clarify what we are addressing in this paper and motivate the approach. In Section 3 we provide the background concerning our design methodology and synthesis flow. Section 4 presents our validation flow along with the TLM co-simulation model and its technical details. Section 5 presents and analyzes the experimental results. Related works are presented in Section 6, and finally, Section 7 summarizes and concludes the paper.

2. Motivational example

Suppose that a JPEG encoder is to be designed for a digital camera. A full-hardware implementation, although providing the best power-performance figures, is not an appealing choice since it takes a very long time to design and is not easily extendable to future applications (such as an MPEG2 encoder on the same camera). Further assume that the available processor core, say Xilinx MicroBlaze soft processor core [9], cannot meet the performance and power requirements on its own, and hence, the system is to be designed as a mixed hardware–software implementation. One way to design such a hardware–software system is to employ a top-down design flow starting from a high-level functional model of the JPEG encoder, e.g., as a software program in C++, and gradually refine it to the ultimate implementation composed of binary executable for the MicroBlaze processor core, and gate-level hardware ready for layout and fabrication. ESL design methodologies, such as the one we follow in this paper (see Section 3), follow such a design flow and provide corresponding design-automation tools to accelerate the time-consuming and/or error-prone steps involved.

Concerning the validation of the system-under-design, the functional model is certainly validated at the start of such design flow to make sure it is functionally correct; for instance in the above example, raw pictures are given to the JPEG encoder program, which is compiled and run on a host computer, and the resulting JPEG pictures are viewed or compared to the expected ones on the same host. Next steps in the design flow typically involve (i) hardware–software partitioning, where parts of the functional model (e.g., computation of the discrete cosine transform – DCT) are assigned to hardware for performance and/or power efficiency, and (ii) elaborating the hardware and the software partitions and their interface down to their final form (i.e., software running on the MicroBlaze processor connected to the DCT hardware engine); the latter step typically involves usage of several tools, sometimes from different vendors, and risks interoperability issues in addition to being very time-consuming (several hours in our studies). At this final form, again the system is certainly validated; typically this validation is done by either rapid-prototyping on a FPGA board or by co-simulating the hardware and the software parts using commercial tools (such as [10]) on a host computer. The missing part in this flow, however, is a middle-stage co-simulation model to let the designer validate the design before starting the intricate time-consuming tasks of the step (ii) above. Without such a model, the error(s) occurring anywhere in the entire design flow cannot be caught until the end of the flow; this not only results in the loss of precious time, but also even after the error manifestation, it cannot be diagnosed; for instance, assume that the above JPEG encoder system worked correctly at first (i.e., as a C++ program), but failed at the final stage on the FPGA board; how can the designer find out in which part of the design flow the error was initiated? Was it in the tasks associated with the step (i) above or step (ii)? This was particularly important to us in developing our design flow since the tasks in step (i) are done by the tools that we have developed ourselves, but the step (ii) mainly involves third-party tools. Thus, if a middle-stage validation model is available for the system-under-design, and furthermore, if it is automatically generated from the initial software model of the system, not only each validation cycle is cut by several hours by detecting errors sooner in the design flow, but also the error is more easily diagnosed by isolating steps (i) and (ii) above. In this paper we present an automatically generated hardware–software co-simulation model for this checkpoint and furthermore show its usefulness in early assessment of the quality of the hardware–software partitioning.
reused in several related applications, or different generations of the same product. Moreover, programming in software is generally a much easier task compared to design and debug of working hardware. Therefore, programmable platforms not only reduce design risk, but also result in shorter time-to-market [12].

ODYSSEY synthesis methodology starts from an object-oriented model and provides algorithms to synthesize the model into an ASIP processor (called OO-ASIP) and the software running on it. The synthesized ASIP corresponds to the class library used in the object-oriented model, and hence, can serve other (and future) applications that use the same class library. This is an important advantage over other ASIP-based synthesis approaches, since they merely consider a set of given applications and do not directly involve themselves with future ones [13].

One key point in the ODYSSEY ASIP is the choice of the instruction-set: methods of the class library that is used in the embedded application constitute the ASIP instruction-set. The other key point is that each instruction can be dispatched either to a hardware unit (as any traditional processor) or to a software routine; consequently, an ASIP instruction is the quantum of hardware–software partitioning. Moreover, it shows that the ASIP internals consist of a traditional processor core (to execute software routines) along with a bunch of hardware units (to implement in hardware instructions as shown in Fig. 2).

An application consists of a class library, which defines the types of objects and the operations provided by them, along with some object instantiations and the sequence(s) of method-calls among them. We implement methods of that class library as the ASIP instructions, and realize the object instantiations and the sequence(s) of method-calls as software running on the ASIP. A simple internal architecture for such an ASIP is briefly presented in Section 3.2.

3.2. ASIP architecture

The internal architecture of the OO-ASIP is shown in Fig. 2. It corresponds to a library comprising two classes, A and B, where B is derived from A and has overridden its \( \text{f}() \) and \( \text{g}() \) methods and has introduced \( \text{h}() \) method. The following C++-like code excerpt demonstrates this. Note that redefinitions of the same method can reside in different partitions (e.g., A::\( \text{g}() \) is a software method while B::\( \text{g}() \) is a hardware one).

```cpp
class A {
    void \( \text{f}() \); // is implemented in hardware
    void \( \text{g}() \); // is implemented in software
    ...;
};

class B extends A{
    void \( \text{f}() \); // overrides A::\( \text{f}() \) and is again in hardware
    void \( \text{g}() \); // overrides A::\( \text{g}() \), but is in hardware now
    void \( \text{h}() \); // new method and is implemented in software
    ...
};
```

All objects’ data are stored in a central data memory accessible through an object management unit (OMU). In the application corresponding to Fig. 2, three objects are defined: \( O_{A1}, O_{A2}, \) and \( O_{B1} \). Objects of the same class (e.g., \( O_{A1} \) and \( O_{A2} \)) have the same layout and size in memory for their attributes. Objects of a derived class keep the original layout for their inherited attributes (the white part of the memory portion of \( O_{A1} \)) and append it with their newly introduced attributes (the gray part of \( O_{B1} \) in box in Fig. 2).

The class methods that are assigned to the hardware partition, i.e., the hardware methods, are implemented as functional units (FU); the other class methods, i.e., the software methods, are software routines stored in the local memory of the traditional processor core (the upper-left box inside the OO-ASIP in Fig. 2).

To efficiently dispatch virtual method-calls to hardware as well as software methods, we have proposed a network-based mechanism [14] that dispatches virtual method-calls as packets sent over an on-chip network to which all possible call targets are connected (the on-chip network in Fig. 2). Further details of the methodology are beyond the space and scope of this paper and are available in [11,15], and [16].

### 3.3. System synthesis flow

We have implemented the above-explained ODYSSEY methodology in a tool-chain [8] comprising some newly developed tools by ourselves and some commercially available ones used for conventional logic synthesis and software compilation. Fig. 3 shows the big picture of the flow of operations in the synthesis process implemented by the tool-chain. Target application (e.g., JPEG decoder) should first be developed as a conventional OO program in C++. This C++ program is input to the tool-chain (the box labeled 1) which produces two outputs at two stages: a hardware–software co-simulation model (at the level tagged 11 in the middle of Fig. 3) and a fully elaborated final system implementation ready to be downloaded to the target FPGA board (labeled 18 at the bottom of the figure). The synthesis process is divided into two layers:

![Fig. 2. Internal architecture of an OO-ASIP corresponding to a class A with \( \text{f}() \) and \( \text{g}() \) methods, and class B derived from A while redefining \( \text{f}() \) and \( \text{g}() \) and introducing \( \text{h}() \) method.](image-url)
the upper layer, i.e., ESL synthesis in Fig. 3, takes the system model and produces software, hardware, and their interface; the lower layer, i.e., downstream synthesis in Fig. 3, takes the above-generated hardware and software partitions and produces gate-level hardware and object code software. The synthesis process is briefly outlined below (full details available in [11]):

Box 1: The target application is coded in OO C++. This is an ordinary C++ program that can be compiled and run on the host computer. The C++ source codes together define the class library as well as the main() function, where the objects are instantiated and the sequence of method-calls among them is specified. The system model at this level can be viewed as TLM level 3 (programmer view) where only the functionality of the system is described and there is no information on timing and structure of the final implementation.

Box 2: The syntax-tree of the above C++ source codes are generated and analyzed to determine classes, class methods, method definitions, and objects.

Box 3: List of the classes and methods are displayed for the user to assign each of them to either hardware or software. We follow manual hardware–software partitioning.

Box 4: Source code of the hardware methods are transformed such that they use our specially designed programming interfaces for (i) calling any other methods and (ii) accessing attributes of the object.

Box 5: The same as Box 4 but applied to software methods and the programming interfaces are also different.

Box 6: Each transformed method code of Box 4 is encapsulated in a separate hardware module with appropriate ports to connect to other hardware components, and with proper SystemC code to implement the above-mentioned programming interfaces.

Box 7: All transformed software methods from Box 5 are together put in a single unit. Other C++ codes are also added to provide the above-mentioned programming interfaces. All of the above items are put in one hardware module (corresponding to a conventional processor core) with appropriate ports to contact other hardware units.

Box 8: These are the bundle of hardware units all in SystemC each corresponding to one hardware method.

Box 9: This is one hardware unit in SystemC corresponding to the cpu (the uniprocessor core) of the system.

Box 10: All other hardware components of the system (including the top-level module, interconnects, and other units to communicate with the outside of the OO-ASIP) which are application-independent are added here.

Level 11: The top-level module composed of Boxes 8-10 is our executable TLM model (detailed in Section 4.3) which isolates the two stages of the design. This is the main focus of this paper. This TLM model is TLM level 2 (programmer view with timing) where the design is hardware–software partitioned and is partially timed; i.e., the coarse-grained structure and the communication architecture of the system are determined and fixed, and the inter-module communication timing (i.e., communication among hardware and software methods) are elaborated – see below. Intramodule timing (i.e., detailed timing inside each module) is determined later when the hardware modules are fully elaborated (Box 13) and connected to the processor (Box 15) through the glue logic (Box 17). We rely on commercially available legacy tools for elaborations needed below this stage.

Since in our design methodology hardware modules (software routines) correspond to hardware methods (software methods), inter-module communication corresponds to method-calls in the input application. Our tool identifies all method-calls in the input C++ code and replaces them with appropriate communication primitive in the produced SystemC code depending on whether the caller is a hardware or software method: for each hardware caller, a separate SystemC interface is used whereas all software methods on the same processor share a common SystemC interface depending on whether the caller is a hardware or software method: for each hardware caller, a separate SystemC interface is used whereas all software methods on the same processor share a common SystemC interface (including the top-level module, interconnects, and other units to communicate with the outside of the OO-ASIP) which are application-independent are added here.

Intra-module timing (i.e., detailed timing inside each module) is determined later when the hardware modules are fully elaborated (Box 13) and connected to the processor (Box 15) through the glue logic (Box 17). We rely on commercially available legacy tools for elaborations needed below this stage.
sequence of inter-module communications remains the same as sequence of method-calls in the original application, and also the hardware and software communication interfaces are designed once by us (the tool developers) but reused for all applications, and hence, their internal timing is fixed and application-independent.

Box 12: Using conventional behavioral SystemC synthesis tools, each hardware module is elaborated to gate-level hardware (Box 13).

Box 14: Based on the target processor core (Box 15) corresponding C++ compiler is used to compile the software part of the system to binary object code (Box 16) executable by the processor core.

Box 17: The system template and parts of the cpu processor module (Box 9) are fixed and are converted to glue logic to integrate all components and generate one integrated final implementation (represented by Box 18).

Box 18: This is the final fully elaborated form of the system. In our experiments, the target platform is an FPGA board; thus, Box 18 corresponds to a bitstream that can be downloaded to the FPGA. In case of chip implementation, this can be further extended to layout and fabrication tapeout steps.

The user of our tools is only involved with Boxes 1 and 3; all the other parts are automatic from the user point of view. In other words, the user develops the OO C++ model of the application (Box 1), feeds it to our tool (Box 2) which gives the list of class methods, and marks each method for implementation either in hardware or software (Box 3); the rest of the process up to final implementation are accomplished by design-automation tools. A few other parts are implementation-dependant and need involvement of the tool developers: Box 10 is application-independent and has been designed once by us, the tool developers, and is used in all runs of the design flow; Boxes 15 and 17 are closely related; each processor core has its own hardware interface and needs corresponding glue logic; we used Xilinx-provided PowerPC and MicroBlaze processor cores and manually designed their appropriate glue logic ourselves; these can be used without modification as long as the processor core is not changed. All other tasks in the flow are automated; the steps above the dashed line marked 11 (i.e., “Transaction-level model in SystemC”) are developed by us and those below it are legacy commercial tools; all of these back-end tools are run and managed by a front-end integrated environment that we have developed.

It is important to note that one main advantage of the ODYSSEY design methodology is extending an existing system to implement further applications via software extensions. In such case, the hardware is fixed and only steps 1, 2, 5, and 7 are done to get the TLM model, and step 14 is further run to get the new final system.

Some C++ constructs are not synthesizable by our current toolset. Some of these limitations can be eliminated by revising and extending the tool-set; this category includes method overloading, class methods with more than 3 arguments, and dynamic object (de)allocation. The other category includes constructs that the ODYSSEY methodology prohibits; this includes recursion involving a hardware method (since it implies expensive stack mechanisms to save the hardware state before re-entry), and passing non-object pointers to hardware methods (since non-object pointers are allocated in the local memory of the processor which is not accessible to hardware modules – see Fig. 2). Since our currently used commercial SystemC synthesis tool does not support synthesis of code containing pointers, pointers cannot be used in hardware methods. Other pointers (i.e., pointers to objects, method-calls on object pointers, and pointers in software methods) are all supported.

4. Validation and verification at ESL level

4.1. Design and validation flow

We follow a simulation-based verification approach. Fig. 4 shows ODYSSEY design and verification flow. The application is developed in C++ and after debugging is transformed (by our ESL synthesis) to the TLM model which is again in C++. Since the TLM model is still in C++ and the entire C++ source code of the application (which can also contain some parts only added for debugging purposes; e.g., file I/O or interacting with the user on console) is transformed by our tool-chain, all same facilities are still available when debugging the TLM model.

Our synthesizer tool generates the complete TLM in addition to appropriately transformed hardware and software method implementations. The TLM co-simulation model is in SystemC and can be readily compiled and run. Since the input C++ program is entirely synthesized into this TLM, the complete test-bench (typically, code to read some stimuli and write outputs on screen or to file), which comprises a part of the input C++ program, is also transformed; consequently, the result of executing post-synthesis co-simulation model must be exactly the same as the pre-synthesis C++ program if the ESL synthesis tasks (see Fig. 3) are correctly accomplished. This gives a simulation-based equivalence checking scheme that can be used to efficiently check functional equivalence of pre- and post-synthesis models at ESL level. Similarly, this equivalence checking scheme can also be used to verify correct operation of the ESL-synthesizer tool. We indeed followed this technique in [8] to verify our synthesizer tool while developing it.

As mentioned above, the synthesizer is capable of transforming the C++ test-bench so that it suits the post-synthesis model; this is accomplished in the same way that other class methods of the model are transformed. This is an automatic test-bench transformation for lower level models that allows to efficiently validate the post-synthesis model before further elaborating software and hardware components. In this way, design re-spins due to hardware–software inconsistencies are avoided sooner in the design cycle.

In terms of timing information, relative timing (i.e., sequence) of the communication among hardware and software modules are verified at this level, but detailed timing simulation is not possible at this level since system components are not yet fully elaborated. We use two traditional co-simulation methods at lower levels of abstraction (i.e., after elaborating hardware and software components) to verify timing, but as will be shown in the experimental results (Section 5) they perform orders of magnitude slower than

![Fig. 4. The design and validation flow.](image-url)
our TLM level co-simulation; this confirms the need to and the success of our early validation technique.

4.2. Correspondence between input and output of the ESL synthesis process

The input and output of the ESL synthesis process and the correspondence between model and implementation elements are depicted in Fig. 5. The input consists of a set of classes along with a main() function in C++ (the left hand side of Fig. 5). The output is a hardware architecture composed of some hardware units and a processor module, along with a software architecture comprised of several routines (the right hand side of Fig. 5).

Implementations of hardware methods are individually put in the hardware modules in the middle of the OO-ASIP box in Fig. 5, whereas implementations of all software methods are put in the processor module. The input main() function is transformed to the thread__main() routine in the processor software. The objects’ data are put in an object-identifier-addressable memory unit, which is the same OMU of Fig. 2. The “on-chip network” is used to dispatch method-calls among software and hardware method implementations (detailed in [14]).

4.3. The executable transaction-level model

The transactions in our TLM model correspond to method-calls and method-returns among method implementations in hardware and software. The TLM model is entirely in SystemC. The processor module contains the thread__main() routine, and hence, is the initiator of co-simulation. The thread__main() routine first initializes objects by calling their corresponding class constructor. It does not allocate memory for the objects since this is statically done by our synthesizer tool. The thread__main() then continues as in the input main() function and invokes methods of objects. If the called method is in software (i.e., a software method is called) the corresponding routine in the cpu module is executed. A method invocation can also be dispatched to a hardware unit in which case the simulation continues with that hardware unit while the processor model (the cpu module) waits for the results of the call. An invoked method implementation may also call other methods; all different cases of hardware and software caller and callee are supported. The co-simulation finishes at the end of the thread__main() routine. Details of this big picture are provided below.

4.3.1. Sequence of events in the co-simulation

The structure of the TLM is shown in Fig. 6. The OO-ASIP module has two network lines, net_in and net_out, to connect to other OO-ASIPs on a network. The reset input is used to restart the OO-ASIP. The main() input is required by the SystemC synthesis tool (it only supports SC_CTHREAD, i.e., clocked thread, processes for synthesis) and it would not be needed if only co-simulation were desired. However, since we pass this same simulation model (except for the cpu module) to downstream synthesis, the clk input is connected to all to be synthesized hardware modules. Each hardware method implementation has a SC_CTHREAD net_recevier() process that manages invocation of the module as well as calling other methods. Similarly, the processor module (the SC_MODULE(cpu) box in Fig. 6) has an SC_CTHREAD net_recevier() process for method-calls and invocation. All modules also have a SC_METHOD restarter() process sensitive to the reset signal to force them to the initial state.

Assertion of the reset signal starts the co-simulation. This event triggers the restarter() process of the cpu module; this process indirectly invokes the thread__main() routine by generating a method-call packet destined for this routine. The thread__main() routine initializes objects and starts to call object methods. To make a virtual method-call or a normal call to a hardware method, the thread__main() routine assembles a packet, sends it to the net_out output of the cpu module, and waits for the method completion that is indicated by receiving a packet labeled METHOD_DONE from the callee. In the current model, net_in and net_out ports of the OO-ASIP module are externally connected together while all hardware modules and the cpu itself are listening to the network and get invoked by any new packet sent there. Everybody checks the packet destination against its own address and takes it if they match. An alternative approach would be to have a separate network module that routes the packet to its unique destination. We implemented the former since the network routing was not of particular interest here to justify the additional burden.

Each hardware module is responsible for only one hardware method, whereas the cpu can contain several software methods. Thus, if the recipient is a hardware module, its single method code (transformed by the synthesizer) is called and executed, but if the packet is destined for the cpu module, a de-multiplexing routine is called that dispatches the packet internally to the appropriate software routine implementing the called method.

Every hardware or software method may also call another method; the procedure of method dispatching is the same in all cases. All software and hardware methods may also access attributes of an object; this is redirected to the OMU which implements a memory addressable by the object-identifier.

4.3.2. Hardware–software synchronization

In our ESL design methodology, synchronization between hardware and software may only take place at the boundaries of method-calls; i.e., method invocation and method completion. The

Fig. 5. The correspondence between components of the model and its implementation.

Fig. 6. Our transaction-level co-simulation model.
synchronization is provided by the network-based method dispatching mechanism. A method-call is accomplished by sending a packet labeled METHOD_CALL from the caller to the callee while method completion is notified by a METHOD_DONE packet from the callee back to the caller. Our system synthesizer tool intentionally preserves semantics of the input C++ model. Consequently, all method-calls are synchronous and the caller is blocked until the callee returns, as defined by the C++ language. In the current implementation, a packet sent on the network invokes all modules connected to the network, but one and only one of them shall be the destination of the packet and shall take and process it. This uniqueness of the call destination is guaranteed by the synthesizer, who assigns network addresses, and ensures single thread of execution in OO-ASIP in the entire co-simulation run. To improve co-simulation efficiency, a separate network router module could have been used to invoke only the unique call destination module instead of invoking all modules. We implemented the former solution since network routing was not of particular focus here.

The synthesizer does not add any timing information inside method implementations other than the synchronization points at method-call boundaries; elaborating internal timing information is left to downstream synthesis. This and the blocking semantics of method-calls eliminate the need to synchronize internal timing of hardware and software modules.

4.3.3. Passing parameters and returning values

For software-to-software calls, traditional parameter passing mechanisms can be used. However, for a method-call involving a hardware method (including virtual method-calls that may be dispatched to hardware), the method invocation packet also carries parameters of the call. Pass-by-value parameters are just copied to the packet data payload. Pass-by-reference is supported only for parameters that are objects, since only objects are globally accessible to both hardware and software methods.

4.3.4. Hardware method implementation

A separate module is generated to implement each hardware method. All such modules have a common template and differ only in their network address and method code. The common template contains port and process definitions, a programming interface to the method dispatching network, and another one to contact the modules outside the cpu. Required code transformations are similarly performed by the synthesizer.

The cpu module is a Bus Functional Model (BFM) of the processor core of the final implementation, but it does not model any particular processor core since it abstracts the detailed interaction between the processor core and the on-chip network. Downstream synthesis steps use appropriate glue logic to connect buses of a selected processor core to this network. The BFM also abstracts fetch-decode-execute phases of the final processor by keeping all software routines in their C++ form. These abstractions result in very high simulation performance presented below.

5. Experimental results

As mentioned in previous section, we follow a simulation-based equivalence checking approach to validate the hardware–software partitioned system early in the design cycle (i.e., before the fully elaborated implementation). Thus, high co-simulation performance is desired here to allow applying more and more stimuli. To evaluate the performance of co-simulation, we synthesized several benchmark programs and measured the simulation time before and after synthesis. We used the Bench++ benchmark suite [17] since it is coded in object-oriented C++, and furthermore, includes large applications as well as traditional Whetstone and Dhrystone benchmarks. We also developed some other benchmarks by ourselves: the first one, “Data Encoder”, encodes a thousand of data benchmarks. We also developed some other benchmarks by ourselves: the first one, “Data Encoder”, encodes a thousand of data numbers to a system-wide controller when synthesized. Direct and indirect recursions are statically detected by the synthesizer and the designer is warned to modify the code and use iteration instead.

4.3.5. Bus functional model of the processor

The cpu module is externally indistinguishable from other modules: it has the same ports and processes as others and operates similarly. Internally, however, it can contain implementations of several methods and specifically the thread::main() routine, which corresponds to the main() function of the original C++ application. Each of these routines is assigned a unique network address, and hence, cpu listens to multiple network addresses. Similar to hardware method modules, the cpu module provides its encapsulated software routines with sufficient programming interfaces to contact modules outside the cpu. Required code transformations are similarly performed by the synthesizer.
however since the comparison is made at a high-level of abstraction, this seems the only simulation performance measure that can be used in this early stage of the design flow.

The measured co-simulation performance is consistently in the multi-millions range which is orders of magnitude higher than co-simulation in lower levels of abstraction such as instruction-set simulation (ISS) and gate-level simulation – see the following subsections. This confirms our claim on efficiency and practicality of our executable co-simulation model for ESL validation and verification explained above.

The co-simulation overhead, presented in Table 1, varies from 1.7% to 84.2% due to different quantities of major constructs that our synthesizer modifies in the benchmarks. These major constructs are method-calls and accesses to object attributes; partitioning necessitates simulating them after synthesis whereas C++ compiler directly supports them before synthesis. Table 2 provides quantity and frequency (before synthesis) of these elements in the benchmarks and confirms that a higher frequency of these constructs results in a higher co-simulation overhead; this is further illustrated in Fig. 7.

To further analyze this overhead, we measured the penalty of each construct separately. Table 3 shows co-simulation performance of two test programs each containing only one of these constructs. It shows that the impact of a method-call is three orders of magnitude higher than an attribute-access on reducing co-simulation performance, and hence, the frequency of method-calls is in general the dominant factor that determines co-simulation overhead.

Although extremely high frequency of method-calls effectively slows down the co-simulation, it also signals a poor model for mixed hardware–software implementation; an extremely high amount of method-calls per second in the model means that the communication (method invocation) to computation (method execution) ratio is extremely high. Such a model is unlikely to effectively benefit from hardware acceleration since very little computation is available in the class methods to accelerate. Therefore, a poor co-simulation performance can be a warning to the designer to revise the model so as to decrease the method-call frequency (e.g., by combining multiple methods into one).

5.1. Implementation results of real-world applications

We [18] also used the tool and the co-simulation model to fully implement some real-world applications and successfully validated them on an FPGA-equipped development board (Xilinx ML-410 board equipped with XC4VFX60 FPGA). Table 4 shows the applications and their characteristics and Table 5 gives the time spent synthesizing each of them (on a 3GHz Pentium IV processor with 1GB of memory) along with the area (in terms of logic slices and memory bits) that they occupy on the target FPGA. Table 6 compares the simulation performances. The downstream synthesis column in Table 5 (also see Fig. 3) gives the time required to fully elaborate the hardware and software partitions using available commercial tools, and the gate-level simulation column in Table 6 gives the simulation time afterwards.

The downstream synthesis time is very low for the motion JPEG decoder and encoder since they are respectively software extensions of JPEG decoder and encoder without any hardware modification, and hence, they do not need any new hardware synthesis; the user notifies our tool of this fact and the tool bypasses the hard-
Table 4
The real-world applications realized on FPGA board

<table>
<thead>
<tr>
<th>Application</th>
<th>#Methods</th>
<th>Lines of C++ source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG Encoder</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Motion JPEG Encoder</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>JPEG Encoder</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Motion JPEG Encoder</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 5
Results of implementing real-world applications on Xilinx ML-410 board using our toolchain

<table>
<thead>
<tr>
<th>Application</th>
<th>Synthesis time</th>
<th>Synthesis results</th>
</tr>
</thead>
</table>
|                             | ESL synthesis (s) | Downstream synthesis (min) | Logic blocks (slices) | Memory (bits)
| JPEG Encoder                | 16.3           | 585               | 1330 + 13,251          | 23,512 + 209,600 |
| Motion JPEG Encoder         | 17             | 9                 | 1330 + 13,251          | 24,812 + 209,920 |
| JPEG Encoder                | 10.4           | 530               | 1330 + 13,565          | 12,438 + 223,456 |
| Motion JPEG Encoder         | 10.6           | 5                 | 1330 + 13,565          | 13,758 + 223,712 |

Table 6
Comparison of co-simulation time of our co-simulation model and that of gate-level simulation

<table>
<thead>
<tr>
<th>Application</th>
<th>Our co-simulation model (ms)</th>
<th>Xilinx Gate-level simulation (h)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG Encoder</td>
<td>98.152</td>
<td>5</td>
<td>183,389x</td>
</tr>
<tr>
<td>Motion JPEG Encoder</td>
<td>289.300</td>
<td>9.5</td>
<td>37,962x</td>
</tr>
<tr>
<td>JPEG Encoder</td>
<td>37.962</td>
<td>4</td>
<td>379,326x</td>
</tr>
<tr>
<td>Motion JPEG Encoder</td>
<td>109.780</td>
<td>7</td>
<td>229,550x</td>
</tr>
</tbody>
</table>

Table 7
Comparing our co-simulation model to ISS-based co-simulation

<table>
<thead>
<tr>
<th>Application</th>
<th>Our co-simulation model (ms)</th>
<th>ISS+HDL co-simulation (min)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG Encoder</td>
<td>98.152</td>
<td>133</td>
<td>81,005x</td>
</tr>
<tr>
<td>Motion JPEG Encoder</td>
<td>289.300</td>
<td>345</td>
<td>71,552x</td>
</tr>
<tr>
<td>JPEG Encoder</td>
<td>37.962</td>
<td>108</td>
<td>170,697x</td>
</tr>
<tr>
<td>Motion JPEG Encoder</td>
<td>109.780</td>
<td>280</td>
<td>153,033x</td>
</tr>
</tbody>
</table>

Other simulators shall be needed to verify the detailed timing. Thus our TLM model is complementary to them.

6. Related works

With the trend toward higher levels of abstraction, operating system (OS) models have been added to hardware and software models for co-simulation [20–22]. We considered only hardware and software in this research.

A variety of approaches exist that co-simulate hardware and software. The simplest solution is to simulate hardware model of the processor along with other hardware of the system in a general hardware simulator. This gives the highest precision but clearly incurs very low simulation speed. Emulation is sometimes used to effectively accelerate this approach. It, however, requires complete elaboration of hardware and software of the system and cannot be used in early stages of the design process. Another approach is to connect an ISS, for the processor, to a hardware simulator using some sort of inter-process communication (IPC) mechanism provided by the host OS [23–29]. The speed of simulation is still a concern here especially since IPC mechanisms are typically time-consuming due to the OS involvement. Emulation is again used to speed up hardware simulation in this case [30]; however, verification of early (unelaborated) hardware and software, where our work is focused at, is not provided.

Compiled co-simulation [31–33] provides a significant speedup (up to three orders of magnitude [31]) compared to ISS-based approaches. Semeria and Ghosh [33] present a BFM model in SystemC for a given processor and use it throughout the design flow from the initial functional model down to detailed implementation where an ISS is encapsulated inside the BFM to determine timing of the software execution. Our approach differs from theirs in three regards. First, we generate the co-simulation model automatically during system synthesis. The second difference is that our BFM model is processor-independent and abstracts details of the processor buses and interrupt lines. Finally, our model is customized to our object-oriented ESL design methodology. A number of other high-level co-simulation environments exist, such as Ptolemy [34] and N2C [35], that are capable of early hardware–software co-validation, but these do not readily match our OOD design methodology and require additional effort to capture our model of computation. It is noteworthy that we could have alternatively tried to take advantage of such tools, instead of SystemC, by devising corresponding design flows and intermediate models such that an automatic tool would generate those models from an input C++ application, however, that (i) would have required to translate the input C++ application to another language suitable for processing by the above-mentioned tools, and (ii) would have complicated and potentially introduced errors into the design flow due to the addition of yet another tool and the corresponding tool interoperability issues. Thus, we decided to rely on the freely available SystemC that, further to its C++ advantages (such as high simulation performance and portability to various hosts and operating systems), relieves us from tool interoperability issues.
SPRINT tool [36] generates transaction-level SystemC model from user-annotated sequential C code so as to enable users to rapidly explore and evaluate various partitioning alternatives. The partitioning unit can be a C function or a single-entry single-exit portion of it, and the inter-module interfaces can be unidirectional point-to-point (default) or shared data (if user directs) depending on the required inter-module communication determined from a dataflow analysis. SPRINT is a generic framework for producing concurrent models from sequential C code, but since OO modeling paradigm is a fundamental choice in our ESL design methodology, our synthesis tool works on C++ code, and furthermore since our goal is to provide a customized solution tailored to our design methodology (not necessarily a generic framework as in SPRINT), we restrict the unit of partitioning to class methods since this importantly enables design of efficient method dispatching solutions [14] among hardware and software methods such that future extensions of the class library are also captured on the same synthesized system. Moreover, SPRINT provides only point-to-point automatic interface synthesis [36], and hence, cannot automatically generate parallel modules from functions (or other portions of code) that are called from multiple functions in the application [36] since this would not be a point-to-point communication, while our TLM model supports such cases by providing a general network-based communication structure. Nevertheless, since SPRINT is a general framework, the dataflow analysis techniques of [36] can be added to our work and also its partitioning approach can be applied to individual hardware or software methods so as to optimize for a certain objective such as performance.

7. Summary and conclusions

We presented an executable co-simulation TLM model and the simulation-based ESL verification procedure that we pursue in our ESL design methodology. The main thrusts of this paper are firstly to propose and implement a simulation-based equivalence checking strategy at ESL level that follows the guidelines that ITRS envisions for future design technology, secondly to introduce an automatically generated executable hardware–software transaction-level model tailored to our ESL design methodology, and finally to demonstrate that the partitioning quality (in terms of communication to computation ratio) determines the co-simulation performance, and hence, the co-simulation overhead can be an early indicator of the quality of the final implementation. Moreover, mechanical test-bench transformation capability of our synthesizer allows the user to more intensively verify the TLM model before further elaborating the design into the gate-level hardware and binary-code software. This prevents costly and time-consuming design iterations from lower level synthesis steps that could have become necessary due to undetected hardware–software inconsistencies or faults in the ESL synthesizer. Such a verification strategy proved essential when developing our design-automation tools [8] as well as when developing our real-world case studies [18]. Envisioned future work includes exploring automatic hardware–software partitioning techniques so as to optimize certain quality metrics such as performance, power consumption, or system cost.

References