Just-in-Time Verification in ADL-based Processor Design

Dominik Auras, Andreas Minwegen, Uwe Deidersen
Stefan Schürmans, Gerd Ascheid, Rainer Leupers
Institute for Communication Technologies and Embedded Systems
RWTH Aachen University, Germany
{auras,minwegen,deidersen,schuerma,ascheid,leupers}@ice.rwth-aachen.de

***Abstract*—A novel verification methodology, combining the two new techniques of Live Verification and Processor State Transfer, is introduced to Architecture Description Language (ADL) based processor design. The proposed Just-in-Time Verification significantly accelerates the simulation-based equivalence check of the register-transfer and instruction-set level models, generated from the ADL-based specification. This is accomplished by omitting redundant simulation steps occurring in the conventional architecture debug cycle. The potential speedup is demonstrated with a case study, achieving an acceleration of the debug cycle by 660x.**

I. Introduction

Embedded systems increasingly feature a multi-processor system on chip (MPSoC) architecture at their core. Driven by partly contradicting constraints such as performance and energy efficiency, these MPSoCs are most likely heterogeneous, i.e. comprising processors with different architectures suited for different types of tasks. Examples are general-purpose, digital signal or application-specific instruction-set processors (ASIPs) tailored towards a certain application domain. While for the first two designers can resort to off-the-shelf components by various vendors, ASIPs are by nature custom designs.

The development of programmable processors on register-transfer level (RTL) is an error-prone task which is further aggravated by the necessity of a software toolchain. In order to solve this issue, several model-centric design methodologies, based on a single formal model of the target architecture specified in terms of an architecture description language (ADL), have been proposed [1, 2, 3], some of which have been commercialized [4, 5]. One of the main advantages of such a model-centric design flow is the inherent consistency that stems from the fact that all information about the target architecture is captured in one centralized model from which models on various abstraction levels can be generated automatically. These are typically an instruction-set level (ISL) model (in this paper always cycle-accurate) used for the fast functional exploration of the target architecture and a RTL model which can be used for implementation at a later point of the design flow. The relationship between the models is illustrated by Figure 1.

Although the models on different abstraction levels are generated from the same architecture description, practice has shown that verification of the RTL model against the ISL model is indispensable. Potential mismatches between

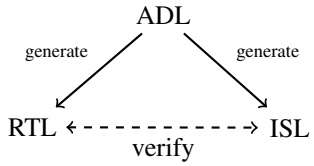

Fig. 1. Relationship between models

models may be caused e.g. by constructs in the ADL model that are allowed on the instruction-set level but have no equivalent hardware implementation. In this paper we focus on verification by simulation-based equivalence checking, which compares traces of the model states obtained by simulating the RTL and ISL models, as described in [6, Ch.5.4.1]. In case a mismatch is detected, the designer has to modify the ADL model, regenerate the ISL and RTL models and perform the equivalence check again. This process, referred to as architecture debug cycle, is part of a typical ASIP development flow, shown in Figure 2. Verification is performed after the

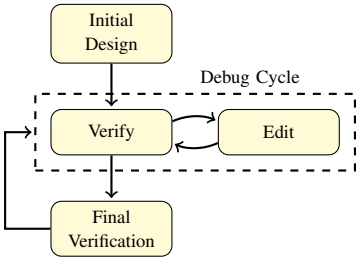

Fig. 2. Architecture debug cycle

initial design. For every mismatch, several debug cycle iterations may be required for locating and fixing the underlying bug. Additionally, a final verification is required, since the design modifications can possibly introduce new bugs. The main bottleneck of such a simulation-based approach is in the relatively low speed of RTL simulation. Furthermore, usually a significant amount of time in the debug cycle is spent on simulating time steps in which no mismatches are encountered, before and after the mismatch occurs. During debugging, these parts are usually re-simulated numerous times, which results in a high degree of redundancy in the debug cycle and thus a high turnaround time.

Contributions. This paper introduces a novel verification methodology that drastically reduces the redundancy observed in the conventional debug cycle described above. The key idea is to use the fast instruction-set simulation as much as possible and to avoid simulating time steps that are irrelevant for the bug targeted in the debug cycle. This is accomplished by means of two novel techniques combined together. First, a live verification mechanism compares the processor states, i.e. all register contents, in realtime while simulating the ISL and the RTL model and stops right after finding any mismatch, thus avoiding redundant simulation steps after the discovery. The second technique is a processor state transfer mechanism that transfers the state across different abstraction levels, to omit redundant RTL simulation steps before the mismatch while debugging the ADL model. The proposed methodology was implemented as a plug-in to a state-of-the-art commercial ADL-based [1] processor design tool suite [4] in a non-invasive manner allowing the novel methodology to be applied to new as well as legacy models. In principle, the concept is also applicable to other tool flows.

The remainder of this paper is organized as follows: After presenting related work in Section II, in Section III the state-of-the-art verification methodology and the proposed methodology are introduced. Implementation aspects are covered in Section IV. In Section V the advantages of the proposed methodology are then demonstrated with two case studies. Conclusions are given in Section VI.

II. Related Work

Verification methodologies can be roughly categorized into formal, simulation-based and hybrid techniques. Formal verification approaches [7, 8] are based on mathematically proving that an implementation fulfills a formal specification, either by applying deductive methods or by enumerating the whole state space and checking for invalid states. In order to enable a mathematical proof, the implementation is usually represented by a formalized model, which has to cover all relevant aspects of the real implementation. If this is not the case or the formal specification is incorrect or incomplete, the formal proof is not meaningful. However, when checking the compliance of an RTL implementation with an ADL specification, proving the validity of the ADL specification is out of scope, and the RTL code constitutes a formal model amenable to formal techniques. Deductive formal methods are hard to apply to complex designs like modern processor architectures, as the high level of abstraction in the ADL and RTL code does not allow proofs to be found automatically, and thus require manual expert work. Moreover, methods based on state space enumeration suffer from state space explosion resulting in extremely high runtimes and memory needs for the verification of large designs.

Simulation-based verification [9, 10] tries to circumvent those problems by running simulations over a set of test cases. While this approach allows to quickly detect major issues, its error coverage depends very much on the selection of the test cases and the set of input data. Therefore, a high coverage requires running many different test cases, which can result in very long simulation times.

Hybrid methods [11, 12] apply formal methods and simulation at the same time to obtain the benefits of both approaches. They use formal methods to analyze the specification and the model to find a set of test cases with the highest possible coverage proving the correctness if no errors are found in simulating these test cases.

The presented work belongs to the simulation-based verification methodologies and tackles its main limitation, speeding up the typical development cycle by omitting redundant parts of the simulation. To the best of our knowledge, this is the first methodology that shortens the debug cycle during RTL verification in ADL-based processor design. The concept does not make any attempt to replace existing verification approaches with high coverage, which are still recommended as final step, as shown in Figure 2. It rather allows to drive the simulations faster to the current point of interest and can thus allow to reach the state of successful verification faster.

III. Verification Methodologies

A. State-of-the-Art Verification

In the state-of-the-art verification flow as typically supported by commercial tools, the ISL and RTL models are simulated separately with the same stimuli. During simulation the state of the respective model is traced by dumping the evolution of the state to a file, e.g. in the value change dump (VCD) format. The equivalence check is then performed post-mortem by comparing the traces of the entire simulation cycle-by-cycle, as shown in Figure 3.

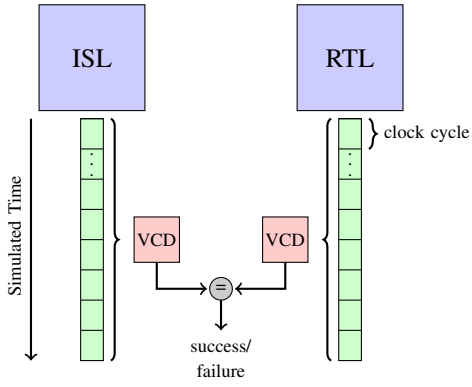

Fig. 3. State-of-the-art verification

While this straightforward approach is widely used, it suffers from two major drawbacks. First, in case of extensive simulations the dump files quickly become very large and difficult to handle, eventually resulting in a slow-down of the entire verification process. More importantly, the simulation is always entirely executed from beginning to end. This leads to the simulation of redundant time steps as depicted in Figure 4(a) since the time steps after the first mismatch are irrelevant for debugging. Moreover after trying to fix the bug that led to the mismatch the previous time steps are

also redundant on RTL in the next iteration of the debug cycle. Hence the state-of-the-art verification flow prolongs the turnaround time unnecessarily, since multiple iterations are required.

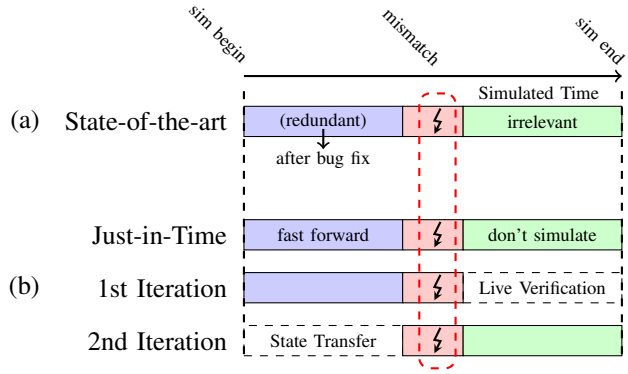

Fig. 4. Simulation procedure

B. Just-in-Time Verification

The novel verification methodology proposed in this paper, referred to as Just-in-Time Verification, significantly reduces the turnaround time. This is achieved by reducing the time spent for simulating redundant time steps in the simulation. Figure 5 depicts the Just-In-Time Verification. In contrast to the state-of-the-art verification methodology the simulation execution of the ISL and RTL models is now interleaved. The reduction of simulated time steps is then accomplished by employing two novel techniques referred to as Live Verification and Processor State Transfer.

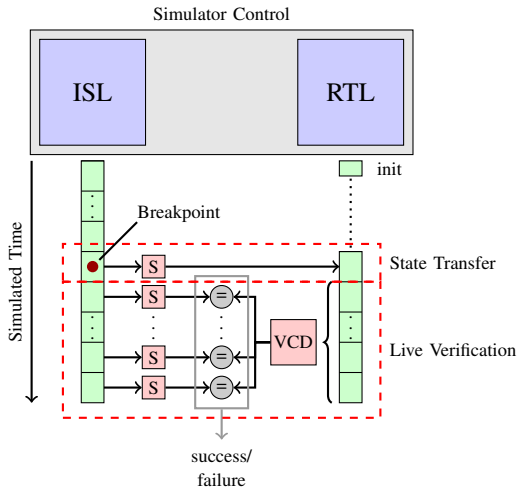

Fig. 5. Just-in-Time Verification

1) Live Verification: Live Verification is based on the idea to perform the comparison of states on-the-fly during simulation execution. The basic mechanism is depicted in Figure 5. The RTL simulation is run for an episode comprising several time steps with tracing enabled and hence the corresponding state trace is dumped to a VCD file. Subsequently the same episode is executed in the instruction-set simulator (ISS) and the ISL state is compared to the corresponding state of the RTL simulation at each time step. This procedure is repeated until the first state mismatch is detected. By running the verification episode-wise the simulation of irrelevant time steps after the first mismatch is minimized, thus reducing the turnaround time (Figure 4(b)). Moreover, large dump files are avoided as traces are only generated for a single episode and solely the RTL simulation is traced.

2) Processor State Transfer: While Live Verification avoids simulating irrelevant time steps on both ISL and RTL, the Processor State Transfer technique is based on the observation that the simulation of an arbitrary episode requires significantly more time on RTL compared to the ISL. Thus, in order to avoid the RTL simulation bottleneck, only the ISS executes from the beginning until it reaches a dedicated breakpoint that indicates the begin of the current region of interest, as shown in Figure 5. Subsequently the processor state on ISL is transferred to the RTL model and the Live Verification is enabled just in time. In this way, the simulation is rapidly forwarded leveraging the speed of the ISS, thus reducing the turnaround time (Figure 4(b)). Note that still a final verification is required to make sure that the bug fix did not introduce new mismatches.

IV. Implementation Aspects

A. Formalized Description

The ISS state comprises the ISL model state $S_{\text{ISL\_proc}}$, which is partly made up of registers modelled in the ADL, like the register file, pipeline and status registers. Let the state space created by them be denoted as $S_{\text{ISL\_regs}}$. Additionally, the ISL model state contains implicit states denoted as $S_{\text{ISL\_impl}}$, which do not correspond to register contents but rather to internal states such as a stall condition of a pipeline stage. Furthermore, not all registers available in the ISL model are also available on RTL, hence the ISL register state space is further divided into subspaces:

$$\begin{aligned} S_{\text{ISL\_proc}} &= S_{\text{ISL\_regs}} \times S_{\text{ISL\_impl}} \\ &= S_{\text{com\_regs}} \times S_{\text{ISL\_only\_regs}} \times S_{\text{ISL\_impl}} \end{aligned} \tag{1}$$

where $S_{\text{com\_regs}}$ is the state space of the registers common to the ISL and RTL model, and $S_{\text{ISL\_only\_regs}}$ comprises the states of those registers that have been removed from the RTL model by optimizations during the HDL generation. Typically, the common registers cover all registers explicitly modelled in the ADL specification, amongst others the register file, pipeline and status registers of the processor.

Considering RTL, the RTL model state is defined as

$$S_{\text{RTL\_proc}} = S_{\text{RTL\_regs}} = S_{\text{com\_regs}} \times S_{\text{RTL\_only\_regs}} \tag{2}$$

where $S_{\text{RTL\_only\_regs}}$ describes the registers only existing in the RTL model. Those register states correspond to the implicit ISL states as the implicit information on ISL is stored in actual registers on RTL. There is a one-to-one mapping t from the implicit ISL states to the RTL only register states:

$$t : S_{\text{ISL\_impl}} \rightarrow S_{\text{RTL\_only\_regs}} \tag{3}$$

This function depends on the architecture model and also requires explicit knowledge of the ADL tools, e.g. about the HDL generation details.

B. Live Verification

The central idea of Live Verification is to extract the state of the common registers from both the ISL and RTL states using observer functions

$$\begin{aligned} b_{\mathrm{ISL}} : S_{\mathrm{com\_regs}} \times S_{\mathrm{ISL\_only\_regs}} \times S_{\mathrm{ISL\_impl}} &\to S_{\mathrm{com\_regs}} \\ (s_{\mathrm{com\_regs}}, s_{\mathrm{ISL\_only\_regs}}, s_{\mathrm{ISL\_impl}}) &\mapsto s_{\mathrm{com\_regs}} \\ b_{\mathrm{RTL}} : S_{\mathrm{com\_regs}} \times S_{\mathrm{RTL\_only\_regs}} &\to S_{\mathrm{com\_regs}} \\ (s_{\mathrm{com\_regs}}, s_{\mathrm{RTL\_only\_regs}}) &\mapsto s_{\mathrm{com\_regs}} \end{aligned} \tag{4}$$

and executing the equivalence check:

$$b_{\mathrm{ISL}}\left(s_{\mathrm{ISL\_proc}}\right) \stackrel{!}{=} b_{\mathrm{RTL}}\left(s_{\mathrm{RTL\_proc}}\right) \tag{5}$$

where e.g. $s_{\mathrm{ISL\_proc}} \in S_{\mathrm{ISL\_proc}}$ denotes a specific state of the ISL model. It is assumed that any value mismatch caused by a bug is visible at least once in one of the common registers.

Using the above definitions and assumptions, the Algorithm 1 defining Live Verification can be introduced. For every simulated time step, the common register states are extracted from the simulator states and compared. As stated previously, performing the comparison in every cycle avoids redundant simulation steps after the bug discovery.

Algorithm 1 Live Verification

```
while not end_of_program do
  simulate one cycle
  if b_ISL(s_ISL_proc) ≠ b_RTL(s_RTL_proc) then
    return error
  end if
end while
```

C. State Transfer

Regarding the state transfer technique, it is essential to cover the complete model state. Cores, memory subsystem and peripherals have to be considered. For the following considerations, a simulation of a single-core pipelined processor is assumed.

1) Processor State Transfer: The processor state transfer process exploits the fact that there is a formal link between the generated models given by the ADL specification. This covers $S_{\mathrm{com\_regs}}$ and the mapping from $S_{\mathrm{ISL\_impl}}$ to $S_{\mathrm{RTL\_only\_regs}}$. For example in models described with the Language for Instruction-Set Architectures (LISA) [1], the RTL only registers comprise bubble state, activation and delayed stall registers.

The transfer of the processor state is described by the state transfer function

$$\begin{aligned} T : S_{\mathrm{ISL\_proc}} &\to S_{\mathrm{RTL\_proc}} \\ (s_{\mathrm{com\_regs}}, s_{\mathrm{ISL\_only\_regs}}, s_{\mathrm{ISL\_impl}}) &\\ \mapsto (s_{\mathrm{com\_regs}}, t\left(s_{\mathrm{ISL\_impl}}\right)) & \end{aligned} \tag{6}$$

which uses the model- and tool-dependent mapping function t from Equation 3.

Upon triggering state transfer, the HDL simulator is instructed to set the state in the RTL simulation to the translated ISL state

$$s_{\mathrm{RTL\_proc}} := T(s_{\mathrm{ISL\_proc}}) \tag{7}$$

by overriding the drivers of registers during a single clock cycle.

2) Peripherals State Transfer: The memory subsystem is a peripheral external to the ADL-based specification. Its a state must be included in the state transfer. Being external, no formal information on its internal structure is available on ISL which can be exploited for the state transfer technique. In principle, the state only depends on the reads and writes issued by the core, therefore by replaying the memory transaction history, the subsystem can effectively be driven into the desired state. Assuming that the subsystem comprises the storage and an attached controller, the required number of transactions can be significantly reduced if the storage content is transferred, i.e. all committed requests are summarized into a single transfer. Only a few transactions need to be replayed in order to restore pending requests within the controller part of the memory subsystem.

In general, the state of any passive peripheral can be restored by recording all transactions on the core interface to the peripheral and subsequently replaying the transaction history. Additionally many peripherals do not require the complete history to be replayed, if the previous cycles can be summarized into a single update or if the state is simply not dependent on the previous cycles. For passive peripherals with a synchronous interface, tracing the interface pins during simulation is sufficient. Upon state transfer, peripheral and processor core are decoupled in the RTL simulation, then the last cycles are replayed on the interface. The handling of active peripherals is more involved and beyond the scope of this paper.

V. Case Studies

We have implemented the proposed methodology as proof-of-concept plug-in for LISA [1] ADL models. Two case studies demonstrate the potential speedup for the ASIP development. The first case study features a customized ASIP for image processing, where Just-in-Time Verification is used to shorten the debug cycle. The second case study was performed for a RISC core and demonstrates another gain of Processor State Transfer which avoids implementing debug-only functionality on RTL. We used Synopsys Processor Designer F-2011.06-SP1 [4] for ASIP development and Synopsys VCS MX 2011.03 [13] for RTL simulations. In the following, the proof-of-concept implementation for Synopsys tools shown in Figure 6 is introduced before the case studies are presented.

A. Proof-of-Concept for Synopsys Tools

Synopsys Processor Designer provides APIs to augment the ISL model using custom functions. The additional code can be included either outside or inside the processor model class. Code inserted into the processor model class can attach to the

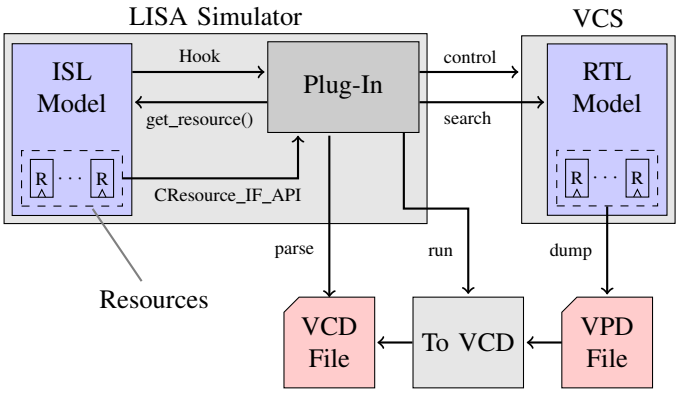

Fig. 6. Implementation of Just-in-Time Verification

main control step function that executes for every cycle, via pre- and post-operation hooks. Furthermore, it can access the generic simulator API to obtain information about the model and to control it (e.g. stop the simulation, set a breakpoint). Our implementation attaches to the simulator in order to execute after every cycle and at the reset event. Initialization takes place upon reset. The plug-in forks a new process which starts the RTL simulation in VCS. Communication with VCS is performed using POSIX pipes.

Resource Detection and Matching. At initialization, the plug-in enumerates all ISL and RTL resources, then finds a matching of the instances. On the ISL side, the LISA API function get_resource() provides information about all available model resources. All registers are stored in the resource list. Regarding the RTL simulator, the implementation uses commands sent over the POSIX pipes to search for all signals following a dedicated naming convention (e.g. all registers have the prefix "REG_") and stores them. Then for every RTL resource, a corresponding ISL resource is identified using simple pattern matching based on the naming convention followed by the HDL generator.

State Retrieval. Before the comparison, the models need to be queried for the state. For the ISL model, the resource interface API enables easy access to all registers. For every resource, the corresponding interface instance is retrieved and polled for the current value. The special states are retrieved, for example the pipeline flush state by calling get_pipe_stage_info() for the appropriate pipeline stage. On the RTL side, VCS is instructed to dump traces of the selected signals. The HDL simulator advances the simulation for one episode comprising a configurable number of cycles. VCS generates a VPD dump file that is first converted to a VCD file. While parsing the VCD file, the plug-in tracks the RTL states for every time step.

Live Verification. When Live Verification is enabled, the plug-in, triggered after every ISL cycle, compares all matched resources. The queried ISL states are compared against the corresponding RTL states. All mismatches are reported immediately. In case of at least one mismatch, the plug-in stops the simulation execution. The contents of the external memories are not compared explicitly, as the implementation assumes that any mismatch should always be observable in at least one register.

Processor State Transfer. When a state transfer occurs, the plug-in basically injects the queried and subsequently translated ISL states into the RTL model. To this end, it has to override the drivers of the signals corresponding to the RTL states. Using VCS, the command "force -deposit" can be used (VPI [14] or VHPI [15] are other possibilities). It overrides the signal value until the original signal driver changes the value again. The implementation generates a script that automatically performs the state transfer, including a copy of the memory content. Furthermore, for the external peripherals like e.g. the memory subsystem, a replay script is generated that drives the controller part of the peripheral into the desired state.

Memory Subsystem. For the plug-in targeting LISA models, the approach of tracing the interface pins is easily implementable with the recently introduced Busport API. All interface pins and the internal controller state are accessible as model resources. However, for the sake of backward compatibility, the implementation also supports the legacy memory interface for LISA models in terms of memory interface description files (MIDF) [16] using a small modification of the ADL model. The plug-in translates the accesses to the appropriate RTL interface transactions using the information available from the MIDF.

B. ASIP For Image Processing

The proposed methodology has been applied to the Retinex processor [17], an ASIP for image processing. The core has seven pipeline stages and uses pipeline bypassing. Its memory subsystem, customized for the application, comprises four memories. The ISA contains 42 instructions. The core uses a simple register file with 16 general purpose 32-bit registers and a few additional special registers. In summary, the architecture design is customized to a great extent and the model complexity is high.

We have run the target application, which processes a color image of size 640x480 pixels, on a reference host that has an AMD Athlon64 Processor 3500+ and four GB of memory. The ISL simulation takes about two minutes to complete, while the RTL simulation needs 11 hours in total. The state-of-the-art verification produces two dump files of 15 GB each.

During the development of the published architecture, the designer encountered a hardware bug in the processor model that was dependent on the input data (i.e. the image) and resulted from a mismatch between the ISL and the RTL model. In this case study, the model before the bug fix is used to demonstrate the impact of Just-in-Time Verification. The errors occurred at the image center. Some pixels that are supposed to be white after processing become black. When debugging the architecture with the state-of-the-art verification, the verification run after each trial fix would have taken 11 hours. Using the Live Verification technique, the verification stops right after processing the image center. Still, after every trial fix, we need to wait for about six hours. With Processor State

TABLE I
IMAGE PROCESSING ASIP CASE STUDY

Verification Methodology	Runtime for N debug cycles	Asymptotic Speed-up of debug cycle
State-of-the-Art Verification	$N \cdot 11$h	1x
JIT w/o Proc. State Transfer	$N \cdot 6$h	1.83x
JIT with Proc. State Transfer	$6\text{h} + N \cdot 1\text{min}$	660x

Transfer in addition, the simulation state can be fast forwarded to the relevant image region, to enable the verification and check if the fix was successful. Using the proposed verification methodology, one iteration of the debug cycle takes about one minute instead of six or 11 hours respectively. This is an asymptotic speedup of 360x to 660x. The results for this case study are summarized in Table I. Note that usually, there will be several debug cycle iterations for each bug.

C. RISC Core

With this case study, the potential speedup for the ASIP development via the functional gain of Processor State Transfer is demonstrated. Assuming that a H.264 video decoder ASIP has to be developed, the first step could be to compile and run the reference C application on a verified RISC core model. Subsequently profiling information recorded during the simulation could be used to extend the ISA. Assume that the memory of the core is too small to store the complete test video file. File handling is required, but it is not desired to attach peripherals to the core nor run an operating system, because the input will be provided by other cores in the final system. A simple solution just for debugging is required. In the implementation of the RISC, the simulated core can read the files of the simulation host by executing a special trap instruction. The implementation of this instruction is not synthesizable and hence not available in the RTL simulation. The designer may implement the same functionality on RTL manually, however there is an easier solution. With the Processor State Transfer technique, part of Just-in-Time Verification, the state can be transferred after every system call. Whenever the simulated cores get out of synchronisation, e.g. due to functionality available only in the ISL model, the state transfer resynchronizes them. This approach saves a lot of development effort. Functionality that is only needed during the development phase does not have to be implemented for the RTL model, as Processor State Transfer allows to use the ISL model.

VI. CONCLUSIONS

A new verification methodology has been presented. Just-in-Time Verification combines two novel techniques, Live Verification and Processor State Transfer, that significantly shorten the architecture debug cycle occurring in ADL-based processor design. During verification in the debug cycle, the ISL and RTL models generated from the single ADL model are checked for equivalence on the basis of simulations. This technique is a common practice to debug ADL specifications. The state-of-the-art methodology compares state traces after completing the ISL and RTL simulations, thereby discovering bugs only post-mortem. This verification approach is very slow, especially for complex modern processor architectures, because the RTL simulation has a long runtime. Live Verification enables early discovery of ISL versus RTL mismatches. It avoids redundant simulation steps after bug discovery by stopping right at the occurrence, because the state comparison is performed on-the-fly cycle-by-cycle. Furthermore, the Processor State Transfer technique is used to omit redundant RTL simulation steps before the region of interest while fixing the bug, leveraging the ISS speed to fast forward the simulation state before enabling Live Verification. The benefit of the proposed methodology was demonstrated in a case study, where a speedup of 660x compared to the state-of-the-art methodology is achieved.

REFERENCES

[1] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[2] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proceedings of the European conference on Design and Test*, 1995.

[3] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: a language for architecture exploration through compiler/simulator retargetability," in *Design, Automation and Test in Europe Conference and Exhibition. Proceedings*, 1999.

[4] Synopsys, "Processor Designer," http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx, 2011.

[5] Target, "IP Designer," http://www.retarget.com/products/ipdesigner.php, 2011.

[6] P. Mishra and N. Dutt, *Processor Description Languages, Volume 1*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[7] P. Camurati and P. Prinetto, "Formal verification of hardware correctness: introduction and survey of current research," *Computer*, 1988.

[8] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Trans. Des. Autom. Electron. Syst.*, 1999.

[9] M. Levinger, M. Molcho, Y. Lichtenstein, Y. Malka, C. Metzger, D. Goodman, G. Shurek, and A. Aharon, "Test program generation for functional verification of PowerPC processors in IBM," *Design Automation Conference*, 1995.

[10] R. E. Bryant, "A methodology for hardware verification based on logic simulation," *J. ACM*, 1991.

[11] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal, "Coverage-directed test generation using symbolic techniques," in *Formal Methods in Computer-Aided Design*. Springer Berlin / Heidelberg, 1996.

[12] H. mo Koo and P. Mishra, "Coverage-driven functional test generation for processor validation using formal methods," 2006.

[13] Synopsys, "VCS," http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx, 2011.

[14] S. Sutherland, *The Verilog PLI Handbook*. Kluwer Academic Publishers, 2002.

[15] "IEEE standard VHDL language reference manual amendment 1: Procedural language application interface," *IEEE Std 1076c-2007 (Amendment to IEEE Std 1076-2002)*, 2007.

[16] D. Kammler, B. Bauwens, E. Witte, G. Ascheid, R. Leupers, H. Meyr, and A. Chattopadhyay, "Automatic generation of memory interfaces," in *System-on-Chip, 2009. SOC 2009. International Symposium on*, 2009.

[17] S. Saponara, L. Fanucci, S. Marsi, G. Ramponi, D. Kammler, and E. Witte, "Application-specific instruction-set processor for retinex-like image and video processing," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 2007.