Generalizing Tandem Simulation: Connecting High-level and RTL Simulation Models

Yue Xing, Aarti Gupta, and Sharad Malik
Princeton University, Princeton, USA
yuex@princeton.edu, aartig@cs.princeton.edu, sharad@princeton.edu

Abstract— Simulation-based testing has been the workhorse of hardware implementation validation. For processors, tandem simulation improves test and debug efficiency by cross-level simulating the Instruction Set Architecture (ISA) and RTL models, and comparing architectural-state variables at the end of each instruction rather than at the end of the whole trace. Further, the simulation may start with the ISA model and switch to the RTL model at some point by transferring the values of the architectural variables, thus speeding up the "warm-up" phase. However, thus far tandem simulation has been limited to processor designs as other SoC components lack high-level ISA models and thus the notion of instructions. Even for processors, significant manual effort is required in connecting the two models and constructing the necessary controller to synchronize/check/swap between them. This paper leverages the recently proposed Instruction-level Abstractions (ILAs) for generalizing tandem simulation to accelerators. Further, we use the *refinement-map* that is part of the ILA verification methodology to automate the connection between the ILA and the RTL simulation models for both processors and accelerators. We provide seven case studies to demonstrate the practical applicability of our methodology.

I. Introduction

Modern System-on-Chips (SoCs) comprise CPUs/GPUs and an increasing number of specialized hardware components broadly referred to as accelerators. SoC design flow starts with high-level design models [1,2] for the various components which are then refined into low-level implementations, typically at the Register-Transfer Level (RTL). To ensure the correctness of a low-level implementation, its equivalence against the high-level model is checked using formal verification or simulation-based testing. While formal verification provides guarantees of correctness, the state explosion problem hinders its use on large designs in practice. Thus, simulation-based testing is more generally applied to ensure the conformance of a low-level implementation with a high-level specification.

In general, conformance testing refers to applying the same sequence of test stimulus to the execution models (**EM**) of the high-level specification and the low-level implementation and determining compliance by comparing their traces at the end of simulation time [3]. This can be inefficient, since the first mismatch generally happens much earlier than the end of simulation and is often enough for debugging. To address this inefficiency, an alternative approach – **tandem simulation (or RTL co-simulation with ISA simulator)** – has been proposed for processor designs [4, 5]. It combines the instruction-level execution model (**ILEM**) based on the processor ISA, and the RTL-based execution model (**RTEM**). The ILEM and RTEM are combined into a cross-level execution model (**CLEM**), and the simulation is executed instruction-by-instruction. At the end of each instruction, the corresponding architectural variables (**AV**) are checked (**AV-Check**). This check ensures that the instruction-level architectural variables (**ILAV**s) and the corresponding RTL architectural variables (**RTAV**s) are equivalent. Any deviation signifies a potential bug, which can be analyzed with nearby instructions for debugging. The AV-Check can also be invoked at specific intervals or checkpoints, to further reduce the performance overhead of comparison. In addition, tandem simulation allows swapping in values from ILAVs to RTAVs (**AV-Swap**), which can be leveraged to jump-start the RTEM in the middle of an ILEM simulation. This can significantly reduce simulation time by leaving the "warm-up" part of the test to only the ILEM.

While tandem simulation has been used for processors, other SoC components, especially accelerators, are not thought of as having instruction-level models, which limits the use of tandem simulation to processor designs. Further, tandem simulation thus far requires customization in that human input is needed to establish the connection between the ILEM and RTEM to apply the AV-Check and AV-Swap. This lack of automation also limits practical application.

This paper addresses the above two gaps by leveraging the recently proposed *Instruction-Level Abstraction (ILA)* to extend tandem simulation to accelerators, and by using a *refinement map* that is specified by a user as part of the formal verification methodology [6–8] to enable automation. The ILA has been recently proposed for formally modeling accelerators [9]. Similar to the ISA for processors, an ILA models an accelerator in terms of a set of architectural variables and "instructions" that update the values of these variables. The ILA model can be used to automatically generate an ILEM, which can then be used to perform tandem simulation with an RTL implementation. Further, we automate the tandem simulation flow by using the refinement map, which specifies: (i) the correspondence between ILAV and RTAV, i.e., *what* to compare for verification, and (ii) the instruction start and finish conditions, i.e., *when* to compare for verification. Thus, it provides the required information for monitoring the RTL implementation and checking its compliance against the ILA model.

Since the ILA generalizes the ISA, the proposed ILA-based tandem simulation methodology applies uniformly to both accelerators and processors. However, there are a couple of chal-

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. This research is also funded in part by NSF award number 1628926, XPS: FULL: Hardware Software Abstractions: Addressing Specification and Verification Gaps in Accelerator-Oriented Parallelism, and the DARPA POSH Program Project: Upscale: Scaling up formal tools for POSH Open Source Hardware.

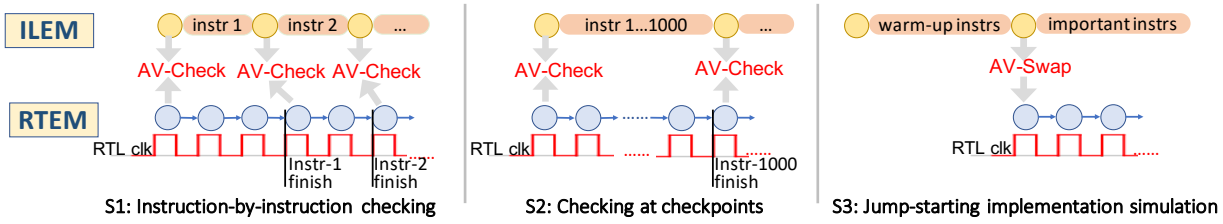


Fig. 1. Three Scenarios for Tandem Simulation

lenges in extending and automating tandem simulation:

- Challenge 1 – Testbenches: The ILEM and RTEM require testbenches in different forms – instruction-by-instruction vs. cycle-by-cycle. Tailoring testbenches for different levels takes extra effort and needs to be automated.
- Challenge 2 – AV-Swapping and micro-architectural variables: The implementation has more variables (micro-architectural variables) than the ILA model. Thus, when swapping AVs from the ILEM to the RTEM, these extra (micro-architectural) variables need to be set properly.

We propose solutions to these challenges based on use of ILAs and refinement maps. We demonstrate the strength of our methodology on seven case studies, covering four hardware accelerator designs (AES-block [10], AES-round [10], GaussianBlur [11], FlexNLP [12]) and three RISC-V processor designs (Pico [13], Piccolo [14], Rocket Core [15]). In particular, GaussianBlur, FlexNLP, Piccolo, and Rocket Core have been integrated into various SoC designs in the broad community, demonstrating that our method is applicable to practical designs. We report the instruction-by-instruction checking time and AV-Swapping time, which are both negligible relative to the simulation time of the ILEM and RTEM. We also provide results for cases when RTL designs are buggy and also for jump-starting, demonstrating the advantage of our methodology in improving debugging and simulation speed.

To summarize, our paper makes the following contributions:

- We extend the tandem simulation methodology to accelerators using ILAs as high-level reference models.
- We describe a fully automated flow to apply tandem simulation to processors and accelerators, by leveraging a refinement map (often available in formal verification).
- We demonstrate the effectiveness of this methodology through seven case studies, including several practical designs at scale.

II. Background

A. Tandem Simulation for Processors

We focus on the following three scenarios of tandem simulation and demonstrate how they can be automatically applied. *Scenario 1: Instruction-by-instruction checking*. Traditional conformance testing applies checking (e.g. AV-Check) at the end of the trace. In contrast, in this scenario AV-Check is applied at the end of each instruction, as shown in Figure 1.
Scenario 2: Checking at checkpoints. The AV-Check is done at predefined points [16] (e.g. Figure 1 applies AV-Check for every 1K instructions). This potentially reduces the performance overhead in doing AV-Check at each instruction.
Scenario 3: Jump-starting implementation simulation. This is useful when a long test stimulus contains both important and unimportant sections (e.g., a warm-up phase in Figure 1). Simulating the unimportant sections only on the ILEM improves the overall simulation speed.

AES ILA		
W	Input	addr_in, data_in, cmd
S	Architectural Variables	key, length, addr, status, data_mem, output_data
S_0	All AVs are initialized as 0	
Instructions (I)		
I_0	set_key	Set encryption key **decode function:** $D_0 = (addr\_in == 0xff10)$ && $(cmd == 2)$ **state update function:** $N_0(key) = data\_in$
I_1	set_length	Set length of text to encrypt
I_2	set_address	Set address of text to encrypt
I_3	start_ encrypt	child instructions: • I_0^c Load data block • I_1^c Encrypt data • I_2^c Store data block
I_4	get_status	Poll status for output

(a) ILA model for AES

AV map		
AES-ILA	**AES-RTL**	
key	top.aes_key.reg_out	
length	top.aes_length.reg_out	
addr	top.aes_addr.reg_out	
status	top.status_reg	
data_mem	top.xram.mem	
	wr_en	top.xram_wr
	addr	top.xram_addr
	data	top.xram_data_in
output_data	top.out_data_reg	
instruction map		
Instruction	**start condition**	**finish condition**
set key	decode	1 cycle
set length	decode	1 cycle
set addr	decode	1 cycle
start encrypt	decode	status == 0
get status	decode	1 cycle
interface map		
AES-ILA	**AES-RTL**	
cmd	wr	

(b) AES Refinement Map

Fig. 2. AES Example

B. Instruction Level Abstraction (ILA)

Recently, the ILA was introduced as a uniform instruction-level formal model for both processors and accelerators [9] [1] Similar to the ISA, the ILA for accelerators specifies a set of instructions and AVs. (The ISA can be viewed as a special case of an ILA.) Each instruction is a command at the interface of the accelerator. For instance, accelerators that are accessed through MMIO (Memory-Mapped Input/Output) are controlled by loads/stores issued by the SW/FW (firmware) on the host processor. The ILA model considers these load/stores appearing at the interface as "instructions" for the accelerator.

Formally, an ILA model [9] is defined as a five-element tuple $\langle S, W, S_0, D, N \rangle$, where S and W are the sets of state variables and inputs, and S_0 denotes initial values. The set of instructions I is defined by sets D and N, which represent the decode functions (the triggered condition) and the state update functions, respectively. An example fragment of an ILA model of a cryptographic (AES) accelerator is shown in Figure 2a (this figure is similar to an ILA example figure from [17], the AES example is from [9]). The AVs include the encryption key, the text length etc. The inputs are the MMIO interface signals. Figure 2a shows the list of instructions and the definition of the instruction `SET_KEY`. *As the state update function is a state transition function for the architectural variables, this lends itself to direct translation to an ILEM for co-simulation.*

The ILA allows for hierarchy to model complex instructions using child-instructions defined in a **child-ILA** (like micro-instructions for complex processor instructions), e.g., the `START_ENCRYPT` instruction in AES is described using child-instructions for loading, encrypting, and storing the data.

C. Refinement Map

A key issue limiting automation of tandem simulation even for processors is the lack of a general approach that connects

[1] This section provides an overview of the ILA modeling and verification methodology and uses examples and figures for exposition here that are similar to those in previous ILA papers (e.g. [9, 17]) with appropriate attribution.

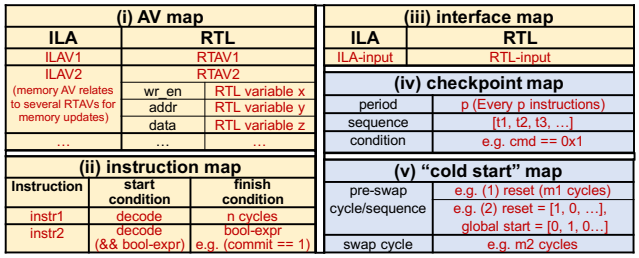
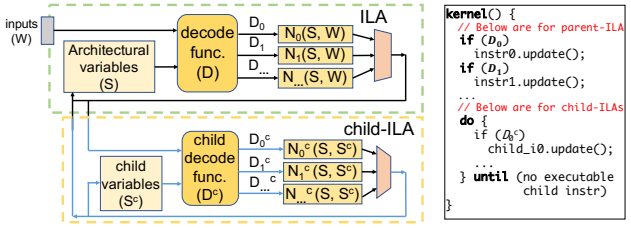


Fig. 3. Refinement Map (sketch)



(a) ILA execution model (b) Kernel template

Fig. 4. ILEM generated from ILA

the ILEM/RTEM and checks the corresponding AVs at the end of instructions. We address this by leveraging the notion of a refinement map, used in formal verification for processors [6,7] and accelerators [8]. As sketched in Figure 3 (similar to a figure in [17]), the ILA refinement map (template shown as black text; refinement map info shown as red text) defines two main fields: (i) *Architectural Variable (AV) map*: defines the mapping from the ILAVs to the corresponding RTAVs. This provides the information about what to check, e.g., `RTAV1` in Figure 3 corresponds to `ILAV1`, and thus they are checked for equivalence. (ii) *Instruction map*: defines the time or condition when each instruction starts (e.g., decode function is true) and finishes (e.g., after n cycles, or after a commit variable is true) in the RTL implementation. This indicates the correspondence points at which the RTAVs should be checked against the ILAVs.

The above two fields specify the key information needed for verification – what to check and when to check. The refinement map also uses the following optional fields as needed: (iii) *Interface map*: provides the correspondence between the ILA inputs and RTL inputs (when not identical). (iv) *Checkpoint map* and (v) *"cold start" map*: are not part of the original ILA refinement map [8] and have been added as part of this work to support tandem simulation. Their use will be discussed in §III.

Figure 2b (similar to a figure in [17]) shows an example refinement map used for AES-ILA and AES-RTL (partially derived from [8]). It shows that, for instance, `top.aes_key.reg_out` from AES-Block implementation corresponds to AES-ILA's `key`. The `set_key` row shows that the instruction starts when the corresponding decode function is true and it finishes after executing one RTL cycle.

III. Generalized Tandem Simulation

In this section, we first introduce ILEM generation, followed by an overview of the proposed methodology for automating tandem simulation. Then we show how the specific challenges discussed in §I are addressed.

A. ILAtor: Automatic Generation of an ILEM

The ILA model is written in a domain-specific language embedded in C++, supported in the ILAng platform [8]. We have developed a tool named ILAtor to automatically generate an ILEM from an ILA model. (The name ILAtor is based on the corresponding tool Verilator [18], which generates an RTEM from a Verilog RTL model.) The ILEM of an ILA model is similar to that of an executable ISA-level processor model. As shown in the upper part of Figure 4a, the ILEM does the following: when a new input instruction is presented, it executes the instruction whose decode function evaluates to true, i.e., its state update function will be applied. The lower part shows the execution model of child-ILAs, which is defined similar to that for the ILA.

The generated ILEM execution kernel is a single thread program representing the ILA execution semantics. ILAtor synthesizes the ILEM in both C and SystemC (as needed) so that it can be easily integrated with RTEM for tandem simulation. The inputs (W) and AVs (S) of an ILA directly correspond to the I/O and member variables of ILEM. For instructions, ILAtor uses the program template shown in Figure 4b to automatically generate the execution kernel. It decodes and executes instructions as defined in its ILA.

B. Methodology Overview

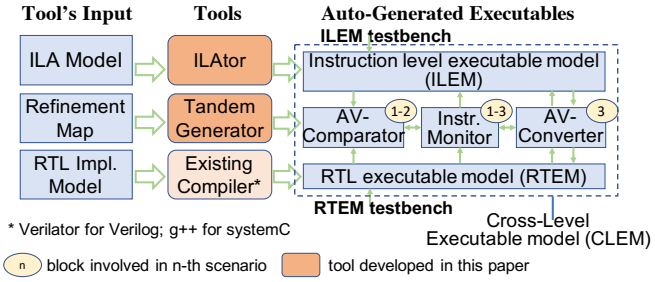


Fig. 5. Tandem Simulation Flow

Figure 5 shows the flow of tandem simulation. The ILEM is generated by the ILAtor, and the RTEM is generated by an RTL simulator-generator (e.g. Verilator [18]). Our tandem tool creates three additional blocks – an instruction monitor, an AV-Comparator, and an AV-Converter – from the refinement map. The instruction monitor uses the instruction map to detect instruction boundaries (if any instruction starts or finishes) in the RTEM. Depending on the scenario, it will invoke the AV-Comparator (Scenario 1-2, Figure 1) for checking AVs, or the AV-Converter (Scenario 3, Figure 1) for swapping AVs and jump-start. Both of these are based on the AV map, which provides the correspondence between ILAVs and RTAVs.

Our methodology augments the refinement map with the checkpoint map (as in Figure 3) to support the following three types of checkpoints for Scenario 2. (1) Checkpoint period (P): invokes checking for every P instructions, (2) Checkpoint sequence ([t1, t2, ...]): invokes checking at the tn^{th} instruction, and (3) Checkpoint condition (C): invokes checking when condition C holds. According to the refinement map, the tandem generator will augment the instruction monitor block to appropriately invoke the AV-Comparator.

A testbench (either for ILEM or RTEM, as in Figure 5) is needed to drive the overall tandem simulation and we assume that such a testbench is given.

C. Challenge 1: Single ILEM Testbench

Unlike processors which fetch instructions from memory, accelerators receive commands/instructions at their interface. Thus, for accelerators the ILEM and RTEM require testbenches in different forms – for ILEM it is usually a sequence of instruction inputs, while for RTEM it is the cycle-by-cycle input

stimulus. The ILEM executes one instruction in a step, while RTEM typically executes an instruction in multiple cycles – during these cycles, the RTEM may block a following instruction if it is not ready to process it.

This tailoring of testbenches for different levels requires additional effort and thus needs automation. In the case when only the ILEM testbench is available, we automate this similar to how processor instructions are simulated from instruction memory. We add an auxiliary "program counter" to the accelerator ILEM and RTEM for accessing an external memory that stores the test instruction sequence. For ILEM, the program counter simply increments by one in each step. For RTEM, the program counter is guarded by the "start condition" (from the refinement map) of the current instruction it points to, i.e., the current instruction will be executed only when the start condition is true. Thus, both ILEM and RTEM run the same test instructions from the ILEM testbench.

In the other direction, when only the RTEM testbench is available, the tandem simulation of Scenarios 1 and 2 can be directly automated since the RTEM is monitored for the instruction it executes. For Scenario 3, an ILEM testbench is still needed, since the ILEM has to execute independently (e.g., prior to the jump-start) without monitoring the RTEM.

D. Challenge 2: Jump-Starting RTEM Simulation

Jump-starting requires the conversion of AVs: from RTL to ILA, and from ILA to RTL. The former is straightforward, since the AV map contains all the information about restoring ILAVs from RTL variables. The other direction is more challenging because the ILA is an abstracted model, and there are RTL micro-architectural variables that are not in the ILA, such as internal counters and pipeline registers. Their values need to be handled carefully.

We address this similar to processor tandem simulation [4,5] by applying a "cold start" to set the RTL micro-architectural variables to their reset values. We then use the AV map to set the RTAVs with the corresponding ILAVs. We automate this with the additional "cold start" map field of the refinement map. In the "cold start" map (Figure 3), the pre-swap cycle/sequence section specifies the input sequence for RTEM reset; the swap cycle describes the holding time for swapped RTAVs to account for designs where RTAVs take multiple cycles to propagate to micro-architectural variables. As shown in example (1) in the pre-swap cycle/sequence section, one can assert `reset` for a couple of cycles. We also support specifying a general sequence to RTEM input pins, as shown in example (2) for `reset` and `global start`.

IV. CASE STUDIES

We applied the proposed tandem simulation methodology to seven case studies, including four accelerator and three processor implementations. We evaluated the following three aspects for all seven designs – (1) the performance (runtime) of each simulated component (ILEM, RTEM, etc.), (2) the simulation speedup with jump-starting, and (3) the improvement of bug detection with instruction-by-instruction AV-Check.

We conducted the experiments on a 3.4 GHz 24-core Intel Xeon server with 62 GB of RAM, running Ubuntu 16.04. We used Verilator v4.1 [18] and SystemC library 2.3.3 [19] for Verilog and SystemC simulation. The open-source ILAng [8] was used for ILA modeling, and we developed ILAtor and the tandem generator for applying the automatic tandem simulation. [2] We also used the base refinement map (described in Json format) from ILAng, and added extra fields ((iv) and (v), colored blue in Figure 3) to support tandem simulation.

The statistics of the ILA model, RTL implementation, refinement map and tandem simulation times are reported in Table I. As the ILA is a higher-level model than RTL, the ILA size (in lines of code, LoC) is smaller than RTL size in all designs. We view the LoC as a rough measure of design complexity or designer effort. Note also that the refinement map size is much smaller than the RTL size, indicating that the human effort in developing a refinement map is much smaller. GaussianBlur, Piccolo and Rocket core were originally generated from High-level Synthesis (HLS)/Hardware Generator – Halide [11], Bluespec [20] and Chisel [21], respectively. We also report the HLS/Generator code size for them.

A. Overview of Case Studies

1. Advanced Encryption Standard (AES): We consider two accelerator implementations for AES [10], implemented in Verilog and C respectively, which implement a block-based and a round-based algorithm, respectively. We use the same AES ILA model [9] (introduced in §II) for both implementations with individual refinement maps. This case demonstrates that different refinement maps enable different RTEMs to be tandem simulated with the same ILEM.

2. GaussianBlur: This case study is of a stencil image processing accelerator for GaussianBlur (GB) [11], synthesized from Halide description using HLS. It demonstrates that our methodology can handle HLS-synthesized implementations.

3. FlexNLP: FlexNLP [12] is designed for machine learning applications with RNN models with attention mechanisms. The design is implemented in 18k lines (excluding the Mentor library code) of synthesizable SystemC. It demonstrates our methodology's strength in handling practical scale designs.

4. Pico, Piccolo and Rocket Core: We have applied our methodology to three RISC-V processor implementations – Pico, a multi-cycle design, and Piccolo and Rocket Core, both pipelined designs. Similar to AES, this case also uses a single RISC-V ILA [9], and uses three different refinement maps for the three implementations.

B. Runtime Evaluation and Simulation Speedups

We applied the three tandem simulation scenarios in all seven case studies and evaluated the simulation speed. We have further broken down the simulation time for each tandem simulation component as presented in Table I . For designs with an available testbench, such as FlexNLP, we use the given testbench to drive the simulation. Other designs are driven by randomly generated test input sequences. Most designs successfully pass the tests except for AES-round, where we identified a bug. The bug happens in an inclusive loop boundary which should have been exclusive and causes encrypting an extra data block in some tests. Our method detects the bug right after the "start_encryption" instruction which causes the state deviation, in about 0.5s (after running about a third of the test sequence). For this design, we used the bug-fixed version in the other experiments measuring simulation time.

The simulation time for RTEM and ILEM is reported in the first two columns of Table I. It is averaged over the number

[2] Source code is available on https://github.com/yuex1994/ASPDAC-tandem

TABLE I
Case Studies – Statistics of ILA Models, RTL Designs, Refinement Maps and Simulation Time

Design	Design Statistics				Simulation Time Breakdown						
	ILA Size (LoC)	# of Arch. Variable bits	RTL Size (LoC)	Ref-map Size (LoC)	RTEM (μs/instr)	ILEM (μs/instr)	S1 (μs/instr)	S2-type1 (μs/instr)	S2-type2 (μs/instr)	S2-type3 (μs/instr)	S3 (μs)
AES (block)	236	298	1078	73	387	7.3	0.25	0.033	0.033	0.033	74.1
AES (round)	236	298	321	62	7.49	7.1	0.64	0.2	0.22	0.22	80.9
GaussionBlur	285	621	11375 (1325†)	147	3.3	1.2	0.19	0.066	0.063	0.068	14
FlexNLP	5807	5008	18338	459	2999	262	17.6	0.083	0.071	0.21	16694
Pico	584	1056	2014	208	0.97	0.29	0.084	0.024	0.019	0.02	0.4
Piccolo	584	1056	6063 (4122†)	223	4.5	0.3	0.26	0.022	0.019	0.019	789
Rocket Core	584	1056	13468 (3856†)	213	101	0.29	0.85	0.029	0.025	0.026	652

† Lines of Code for HLS/Hardware Generator.

of instructions for each test, thus making it a *per-instruction simulation time*. As a higher-level model, the ILEM generally runs much faster than the RTEM, with the speedup ranging from 3X (e.g., GB) to 300X (e.g., Rocket). One exception is for AES-round – its C implementation is already very abstract, thus leaving little room for ILEM speedup.

The third column (S1) demonstrates the runtime overhead of Scenario 1, which includes the per-instruction time for monitoring the RTEM for the instruction boundary and checking the RTAVs against ILAVs after every instruction. For practical designs (e.g., FlexNLP, Rocket Core), this time is within 1% of the RTEM simulation time.

The fourth to sixth columns (S2-type1, S2-type2, S2-type3) show the per-instruction time for monitoring each type of checkpoint, respectively. This can be regarded as the simulation overhead for Scenario 2. As shown in Table I, these take much less time (less than 10% of that for S1), demonstrating speedups in comparison at check-points only, rather than after every instruction.

The last column (S3) lists the AV-Swapping time from ILEM to RTEM, which is the runtime overhead in Scenario 3. It presents the one-time overhead of applying AV-Swapping, not per-instruction. It varies significantly across the designs and is determined roughly by the number of AVs and the "cold start" length. Among all case studies, the swapping time is within the time required for executing several to several hundred instructions on the RTEM. Thus, this overhead is negligible in practical tests that have millions of instructions, as long as AV-Swapping is not invoked very frequently.

C. Simulation Speedup with Jump-Starting

We conducted experiments to evaluate the effectiveness of jump-starting in long input sequences. We divided each test into two parts – a "warm-up" phase and an important phase. We considered different fractions of the test inputs in the warm-up and important phases. For example, we considered the first 5%, 15%, ... as warm-up, and the rest as important phase. Figure 6 presents the simulation speedup for different fractions, in comparison to no jump-start.

Note that many designs have a significant speedup – more than 2X with 80% jump-started instructions – and the speedup increases as a higher fraction of instructions are jump-started. The dashed line in the figure plots a theoretical maximum speedup for a given fraction, which is computed by assuming the ILEM simulation takes no time (and RTEM simulates different test inputs with a constant speed). For example, when 95% of the test inputs are jump-started, the simulation time is at least simulating the remaining 5% on RTEM. Therefore, the upper bound of the speedup is $\frac{T_{RTEM}}{0.05*T_{RTEM}} = 20$. As seen in Figure 6, the speedup of AES-block, FlexNLP, and Rocket

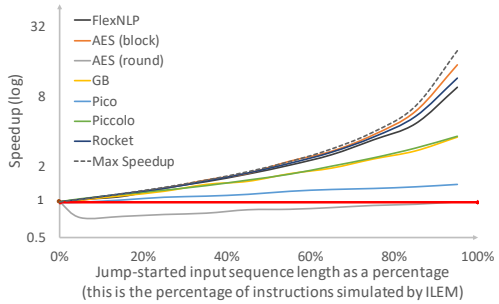


Fig. 6. Simulation Speedup (logarithm y-axis) for jump-starting x% of the input instruction sequence on ILEM

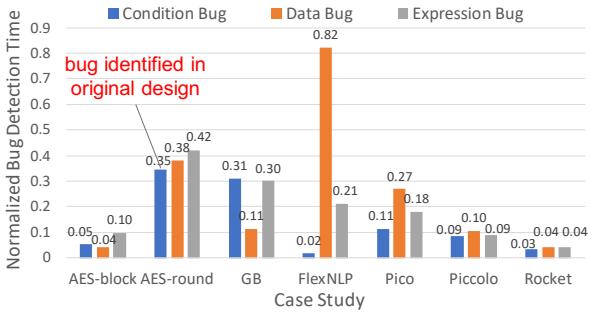


Fig. 7. Bug detection time (normalized to the simulation-to-the-end time) for various design cases

is very close to the upper bound. They all achieve more than $10X$ speedup at the 95% fraction point. However, due to the C implementation of the AES-round being very abstract, jump-starting it achieves no speedup (speedup is less than one).

D. Improvement in Bug Detection

We also studied the improvement in bug detection time by measuring the elapsed time to detect bugs using tandem simulation. As mentioned in §IV-B, most available designs are bug-free. So, we set up the experiment by inserting a bug in each design. Specifically, we consider three types of bugs: a "condition bug" changes a value/condition in a conditional (an if-then-else or case) statement; a "data bug" changes a value in a computation; an "expression bug" changes a logic operator (e.g., from AND/OR to XOR). We inserted a bug of each type separately, leading to three buggy variations per design. Further, there are tens/hundreds of candidate locations for bug insertion – we randomly picked one for our experiments. For AES-round, the bug identified in §IV-B belongs to the "conditional bug" category and is also used here. The test inputs are randomly generated (as in §IV-B), and are long enough to detect the bugs.

We evaluated two debug strategies: 1) traditional conformance testing – which runs the test to the end and then compares the ILEM and RTEM results, and 2) tandem simulation

– which runs the test instruction-by-instruction and applies the AV-Check at the end of each instruction. We applied these two strategies to each bug variant of the designs and measured their bug detection time. We normalized the bug detection time of the second strategy by that of the first strategy and plotted it in Figure 7. (The absolute simulation time for the first strategy ranges from 1-15 seconds for different design variants.) The normalized numbers here demonstrate that tandem simulation often detects the bug earlier than finishing the test in conformance testing. In many cases, it finds the bugs in less than 10% of the full test time, and in most cases in less than 40%. An outlier is a data bug in FlexNLP, where the buggy data is used only in a very late stage of the test program.

E. Summary

In summary, our experimental results demonstrate that:

- Tandem simulation for all three proposed scenarios can be effectively automated using the ILA model and refinement map from the ILA verification methodology for processors and accelerators.
- The overhead of the extra components introduced by our automatic tandem simulation methodology (i.e., AV-Comparator, Instruction Monitor, and AV-Converter) is negligible compared to RTEM simulation time.
- There is a significant simulation speedup by jump-starting unimportant/"warm-up" phases.
- The instruction-by-instruction checking detects bugs earlier than run-to-the-end methods.

V. Related Work

The idea of tandem simulation was proposed in BlueSpec's toolchain [4] for various RISC-V processor implementations [14]. Similarly, the BlackParrot project integrated the Dromajo RISC-V ISA co-simulator [22] with their RTL simulation, effectively providing tandem simulation capability [5]. Earlier designs, such as the IBM Power processors, were also validated using instruction-by-instruction checking [23]. These methods are limited to processor designs, and are manually done with no systematic methodology. In contrast, our proposed method leverages the ILA model for extending tandem simulation to accelerators and leverages the refinement map for automation.

Past work has also explored the idea of co-simulating cross-level models, especially between the transaction-level model (TLM) and RTL [24, 25]. These techniques utilize a transactor (either manually or automatically generated) to refine the existing TLM test inputs or functional assertions into RTL simulation, where the RTL can be verified by checking the test output or triggered assertions. Unlike these works, our tandem simulation approach is based on the RTL model being a refinement of the ILA model, and thus focuses on the architectural state variables and checks them at the granularity of instructions. This provides the key benefit of bug detection and early termination, while the TLM/RTL co-simulation generally requires finishing the whole test before checking. It also enables jump-starting through AV-Swapping, which is a harder task for TLM-to-RTL cross-level simulation.

VI. Conclusions

In this paper, we generalize the notion of instruction-level and RTL tandem simulation to include accelerators in addition to processors. We propose an automatic flow for tandem simulation by leveraging the refinement map, which is used by designers for formal verification. We discussed the challenges in this generalization and proposed a methodology that uses the ILA model for automatically generating an instruction-level execution model, and adapts its associated refinement map for automating the comparison checks and jump-starting in tandem simulation. We applied this methodology to seven design case studies, including several processor and accelerator designs. The evaluation results demonstrate the effectiveness of the proposed tandem simulation methodology in improving simulation speed-up and earlier bug detection.

References

[1] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemCTM*. Springer Science & Business Media, 2007.

[2] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," in *1st CODES+ISSS*, 2003, pp. 19–24.

[3] P. Herber, M. Pockrandt, and S. Glesner, "Automated conformance evaluation of SystemC designs using timed automata," in *15th IEEE European Test Symposium*, 2010, pp. 188–193.

[4] R. Nikhil and D. Rad, "RISC-V at Bluespec," 2015, [Online]. Available:https://riscv.org/wp-content/uploads/2015/01/riscv-bluespec-workshop-jan2015.pdf, accessed on: 2020-11, (slides from 1st RISC-V Workshop).

[5] D. Petrisko, F. Gilani, M. Wyse, T. Jung, S. Davidson, P. Gao *et al.*, "BlackParrot: An Agile Open Source RISC-V Multicore for Accelerator SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020.

[6] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *CAV*, 1994, pp. 68–80.

[7] P. Manolios and S. K. Srinivasan, "Automatic verification of safety and liveness for pipelined machines using WEB refinement," *TODAES*, vol. 13, no. 3, pp. 1–19, 2008.

[8] B.-Y. Huang, H. Zhang, A. Gupta, and S. Malik, "Ilang: a modeling and verification platform for SoCs using instruction-level abstractions," in *TACAS*, 2019, pp. 351–357.

[9] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," *TODAES*, vol. 24, no. 1, pp. 1–24, 2018.

[10] H. Hsing, "OpenCores.org: Tiny AES," 2014, [Online]. Available:https://opencores.org/project/tiny_aes, accessed on: 2020-04.

[11] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming Heterogeneous Systems from an Image Processing DSL," *TACO*, vol. 14, no. 3, pp. 1–25, 2017.

[12] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush *et al.*, "Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference," in *DAC*, 2020, pp. 1–6.

[13] C. Wolf, "PicoRV32," 2020, [Online]. Available:https://github.com/cliffordwolf/picorv32, accessed on: 2020-11.

[14] Bluespec, Inc., "BlueSpec RISC-V designs," 2020, [Online]. Available:https://github.com/bluespec, accessed on: 2020-11.

[15] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[16] S. Kraemer, R. Leupers, D. Petras, and T. Philipp, "A checkpoint/restore framework for SystemC-based virtual platforms," in *International Symposium on SoC*, 2009, pp. 161–167.

[17] Y. Xing, H. Lu, A. Gupta, and S. Malik, "Leveraging processor modeling and verification for general hardware modules," in *DATE*, 2021, pp. 1130–1135.

[18] W. Snyder, D. Galbi, and P. Wasson, "Verilator," 2009, [Online]. Available:https://www.veripool.org/projects/verilator, accessed on: 2020-11.

[19] Mentor, "Accellera Systems Initiative." 2020, [Online]. Available:https://www.accellera.org/downloads/standards/systemc, accessed on: 2020-11.

[20] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *MEMOCODE*, 2004, pp. 69–70.

[21] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis *et al.*, "Chisel: constructing hardware in a scala embedded language," in *DAC*, 2012, pp. 1212–1221.

[22] Esperanto Technology, "Dromajo," [Online]. Available: https://github.com/chipsalliance/dromajo/, accessed on: 2020-11.

[23] D. W. Victor, J. M. Ludden, R. D. Peterson, B. S. Nelson, W. K. Sharp, J. K. Hsu *et al.*, "Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems," *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 541–553, 2005.

[24] N. Bombieri, F. Fummi, and G. Pravadelli, "On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL," in *DATE*, 2006, pp. 1–6.

[25] M. Chen and P. Mishra, "Assertion-based functional consistency checking between TLM and RTL models," in *26th VLSID*, 2013, pp. 320–325.