KAIROS: Incremental Verification in High-Level Synthesis through Latency-Insensitive Design

Luca Piccolboni, Giuseppe Di Guglielmo, and Luca P. Carloni
Department of Computer Science, Columbia University, New York, USA
Emails: {piccolboni, giuseppe, luca}@cs.columbia.edu

***Abstract*—High-level synthesis (HLS) improves design productivity by replacing cycle-accurate specifications with untimed or transaction-based specifications. Obtaining high-quality RTL implementations requires significant manual effort from designers, who must manipulate the code and evaluate different HLS-knob settings. These modifications can introduce bugs in the RTL implementations. We present KAIROS, a methodology for incremental formal verification in HLS. KAIROS verifies the equivalence of the RTL implementations the designer subsequently derives from the same specification by applying code manipulations and knobs.**

I. Introduction

The increasing complexity of hardware design is pushing the adoption of high-level synthesis (HLS) in academia [1] and industry [2]: designers are starting to abandon cycle-accurate specifications, e.g., *Verilog*, in favor of untimed or transaction-based specifications, e.g., *C, SystemC*. This allows designers to reduce simulation times and synthesize many RTL implementations, thereby improving design-space exploration (DSE) [3].

A high-level specification is usually organized hierarchically. A module contains processes and processes are divided into regions[1]. Regions are defined by partitioning the code into blocks with HLS macros. Modules, processes, and regions are often designed to expose latency-insensitive interfaces [1], and they can be connected (synchronized) with latency-insensitive channels, through HLS libraries such as *MatchLib* [5]. Latency-insensitive design (LID) allows modules (the same also applies to processes and regions) to tolerate any timing variation in the computation within themselves as well as in their communication with other modules [6], [7]. This is obtained by adding valid and ready signals to the interfaces of the modules. The valid signal indicates that the value of the signal is valid in the current clock cycle, while the ready signal is used to flag backpressure [8].

Unfortunately, HLS still requires considerable manual efforts to synthesize optimized RTL solutions [9]. HLS is supported by automatic tools and libraries [10], but manipulations of the high-level specifications are necessary. For instance, designers may need to modify the code to break the dependencies that limit parallelism. Further, designers need to set the HLS knobs to explore different architectural solutions. For example, by means of "loop unrolling", designers can generate RTL implementations with more hardware resources, thereby increasing performance in exchange for higher area/power. By applying code manipulations and knobs, designers obtain many RTL implementations $\mathcal{I}_1$, ..., $\mathcal{I}_N$, each offering potentially a unique trade-off point in terms of cost and performance. $\mathcal{I}_1$, ..., $\mathcal{I}_N$ are not necessarily equivalent clock cycle by clock cycle, but they are expected to be latency-insensitive equivalent [6], i.e., they should produce the same sequence of outputs, possibly with different timing.

[1]We use the terminology of *SystemC* [4] and *Cadence Stratus HLS* in this paper, but our methodology can be adapted to other languages and HLS tools.

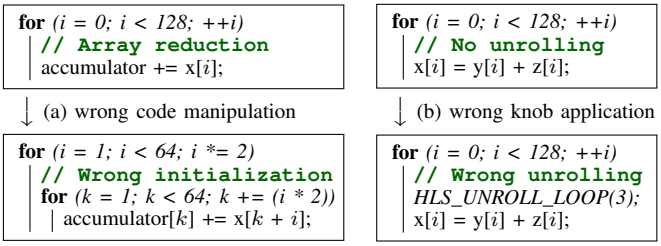


Fig. 1. Bugs that can be introduced during HLS-based DSE.

Code manipulations and knob applications are prone to bugs. For example, in Fig. 1 (a), the code of an array reduction has been modified to make the parallelism explicit. The index k of the inner loop is, however, initialized wrongly (the correct value is 0). In Fig. 1 (b), on the other hand, a loop is unrolled three times by means of the "loop unrolling" knob. If the loop is partially unrolled without adding the necessary exit checks, i.e., aggressively, then out-of-bound memory accesses occur. In fact, since the number of loop iterations (128) is not a multiple of the unrolling factor (3), the last loop iteration is incorrect. These two examples produce incorrect RTL implementations. Note that, while the first bug can be caught by simulating the high-level specification, the second bug requires to verify the RTL code synthesized with HLS. It is thus necessary to check the correctness of all RTL implementations obtained with HLS.

The common practice to verify the RTL implementations generated with HLS is to use simulation-based approaches, i.e., the RTL implementations are co-simulated with a testbench written in a high-level language [11]. Most commercial tools adopt this technique, e.g., *Xilinx Vivado HLS*, *Cadence Stratus HLS*, etc. This allows designers to detect bugs introduced by the HLS tools or code manipulations, but it does not guarantee the absence of bugs. In this paper, we focus on the problem of performing formal incremental verification of the code manipulations and knob applications made to the high-level specification of a HLS design. Formally, we focus on the following problem:

Problem 1. Given a reference RTL implementation $\mathcal{I}_{ref}$ and a set of RTL implementations $\mathcal{I}_1$, ..., $\mathcal{I}_N$, obtained by applying (i) code manipulations or (ii) the HLS knobs to the same HLS-ready specification used for synthesizing $\mathcal{I}_{ref}$, formally prove that $\mathcal{I}_1$, ..., $\mathcal{I}_N$ are latency-insensitive equivalent to $\mathcal{I}_{ref}$.

For Problem 1, we assume that modules, processes and regions

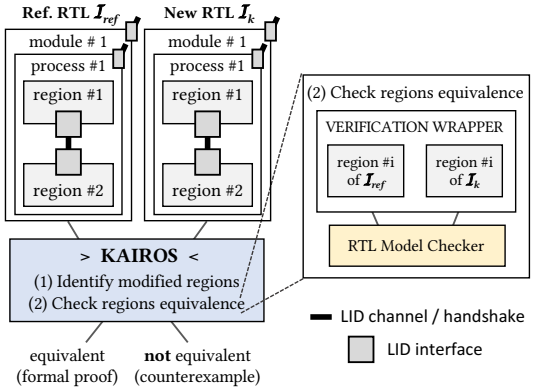


Fig. 2. An overview of the KAIROS verification methodology.

used in $\mathcal{I}_{ref}, \mathcal{I}_1$, ..., $\mathcal{I}_N$ are all developed by exploiting LID.

Contributions. We propose KAIROS, a methodology for incremental formal verification in HLS (Fig. 2). First, the designer synthesizes a reference RTL implementation $\mathcal{I}_{ref}$ from the initial high-level specification by exploiting LID. KAIROS assumes that $\mathcal{I}_{ref}$ has been validated by the designer. KAIROS supports the verification of the successive modifications that the designer may want to apply ($\mathcal{I}_1$, ..., $\mathcal{I}_N$). After the designer has modified the specification and synthesized a new RTL implementation $\mathcal{I}_k$ with $k \in [1, N]$, KAIROS (1) identifies which regions of the code have been modified in the new RTL implementation $\mathcal{I}_k$ and (2) tries to formally prove that such regions are equivalent to the corresponding ones in the reference implementation $\mathcal{I}_{ref}$. KAIROS uses latency-insensitive equivalence because regions can have different latencies as result of the designer's modifications (e.g., code refactoring, loop unrolling, etc.). KAIROS does not need to prove the correctness of the composition of regions, processes and modules since it is guaranteed by LID. We evaluate KAIROS by checking the equivalence of multiple RTL implementations of a hardware module and a RISC-V processor designed with HLS: KAIROS can quickly detect bugs caused by wrong code manipulations and knob applications.

II. The KAIROS Methodology

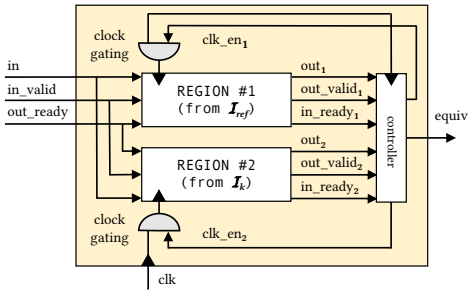
STEP #1: Identify Modified Regions. After $\mathcal{I}_k$ has been synthesized, KAIROS identifies which regions of $\mathcal{I}_k$ have been affected by code manipulations or knob applications. Note that only the affected regions have to be verified. The composition of such regions in processes and modules is guaranteed to be correct by construction thanks to the results proven for LID [7]. To identify these regions, KAIROS analyzes the RTL code synthesized by the HLS tool. A file is generated for each region, and thus KAIROS can determine which files present modifications with respect to $\mathcal{I}_{ref}$. Some HLS tools support also Engineering Change Order [12], which simplifies the identification of the modified regions by exploiting the information in the control-data flow graphs (CDFGs) created during the synthesis [13]. If some regions are merged or divided, KAIROS can expand the verification from regions to processes or modules. Once an implementation $\mathcal{I}_j$ has been verified, it can be used (in place of $\mathcal{I}_{ref}$) to identify the regions affected by the designer's modifications for another implementation $\mathcal{I}_k$ with $k \neq j$. Incrementally replacing the reference implementation with the new verified one can speed up the verification process.

STEP #2: Check Regions Equivalence. After identifying a pair of regions in $\mathcal{I}_k$ and $\mathcal{I}_{ref}$, KAIROS verifies their equivalence. KAIROS generates a *verification wrapper* (in RTL Verilog) that encloses the two regions (Fig. 2). The block diagram of the wrapper for single-input and single-output regions is shown on the left of Fig. 3. The wrapper exploits clock gating [14] to make the regions stallable, so that they can be "stopped" when it is needed (this is required to apply LID). KAIROS creates the wrapper by using only the interface of the RTL code of the two regions synthesized by the HLS tool. The wrapper has a controller to enable and disable the clock-gating logic used to manage the encapsulated regions. The wrapper performs 4 steps:

(1) it waits for one region to complete its execution, i.e., it waits either for out_valid_1 or out_valid_2 to be equal to 1;
(2) it disables the clock of the region that completed (clk_en_1 or clk_en_2), so that it can then wait for the other region;
(3) it waits for the other region to complete its computation;
(4) it sets *equiv* to 0 if $out_1 \neq out_2$ and to 1 otherwise.

The verification wrapper synchronizes the two regions. To check the equivalence, KAIROS applies model checking [15] to prove that *equiv* is always equal to 1. While it is common practice to check cycle-by-cycle equivalence of two RTL designs, here we use model checking to prove latency-insensitive equivalence between two RTL designs synthesized with HLS.

Example II.1. The trace of Fig. 3 (right) shows how to check equivalence. The colors indicate the steps the wrapper performs. a^i is a generic input value, while b_1^i and b_2^i are the output values produced by `REGION#1` and `REGION#2` corresponding to input a^i, respectively. The wrapper synchronizes the regions, so that the model checker can verify if $b_1^i = b_2^i \; \forall \; i$. □



steps: (1) (2) (3) (4)

cycles	0	1	2	3	4	5	6	7	8	9
in	a^0	-	-	-	a^1	-	-	-	a^2	-
out_1	-	-	b_1^0	b_1^0	b_1^0	-	b_1^1	b_1^1	b_1^1	-
out_2	-	-	-	-	b_2^0	-	-	-	b_2^1	-
clk_en_1	1	1	0	0	1	1	0	0	1	1
clk_en_2	1	1	1	1	1	1	1	1	1	1
out_valid_1	0	0	1	1	0	0	1	1	0	0
out_valid_2	0	0	0	0	1	0	0	0	1	0
equiv	1	1	1	1	1	1	1	1	1	1

Fig. 3. The block diagram of the verification wrapper (left) and an example of latency-insensitive equivalence checking (right). Here, we assume that (i) the regions are not pipelined, and (ii) we do not have backpressure (KAIROS can handle these cases). The regions have a single input (*in*) and a single output (*out*).

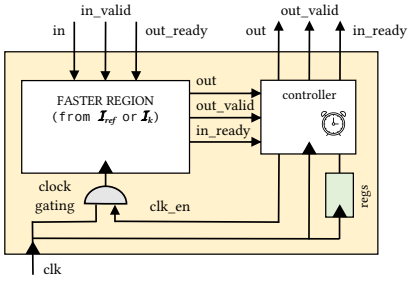


Fig. 4. Verification wrapper for data-independent regions.

III. Improving Scalability

RTL equivalence checkers [16] are usually faster than model checkers for checking the equivalence of two RTL descriptions. Theoretically, model checkers can be used for equivalence checking. From a practical perspective, however, equivalence checkers are faster because they can exploit the structural similarities of the two RTL descriptions. In fact, many formal tools, e.g., *Cadence JasperGold*, have distinct "Apps" for the two tasks. Unfortunately, equivalence checkers cannot be directly used in our context because they do not allow one of the two descriptions to access the signals of the other one (we would need to access the signal *out_valid* to synchronize the regions). We can observe, however, that accessing the *out_valid* signal is not needed in case of data-independent regions: i.e., regions that have a fixed latency and whose latency does not depend on the particular values of the inputs. For these regions, KAIROS creates a specialized version of the verification wrapper that enables the use of an equivalence checker. The wrapper encapsulates only the faster of the two regions (Fig. 4). Then, KAIROS uses an equivalence checker to verify the equivalence between the "wrapped" faster region and the slower region. The wrapper uses a counter that accounts for the difference in latency of the two regions that are being verified. KAIROS extracts the initialization value of this counter from the CDFG created by the HLS tool [13]. In fact, the HLS tool can easily estimate the number of clock cycles required to execute a data-independent region. In this way, KAIROS can slow down the faster region and synchronize its outputs with the outputs of the slower region. To verify the equivalence, identical inputs are given to the "wrapped" faster region and the slower region at every cycle and the equivalence checker verifies that the corresponding outputs of the regions match only when the *valid* signals of the wrapper (out_valid_1) and the slower region (out_valid_2) are both 1. This wrapper can be used only in the case of data-independent regions. If the regions are data-dependent, the more general solution reported in Fig. 3 should be adopted.

IV. Experimental Evaluation

Experimental Setup. We designed a hardware module in *SystemC*, called GRAY, that converts a $N \times N$ RGB image into a grayscale image. The architecture of the module is organized in three processes: the load process reads the input data and stores it in a private array of size N; the compute process performs the computation; the store process produces the output data. The three processes are pipelined with ping-pong buffering [3].

GRAY 32×32

	Entire Module			Compute Process			
Name	**FFs**	**Gates**	**Lines**	**FFs**	**Gates**	**Lines**	**Outputs**
REF	7716	∼33K	∼60K	3420	∼14K	∼7K	64
UROL32	16374	∼410K	∼61K	12438	∼390K	∼8K	64

Name	**Area**	**Latency**	**Bounded**	**Exhaustive**
REF	51831 *um²*	497500 *ns*	–	–
UROL2	55927 *um²*	481500 *ns*	*proven* - 1 *min*	*proven* - 22 *min*
UROL4	63895 *um²*	457500 *ns*	*proven* - 1 *min*	*proven* - 25 *min*
UROL8	79955 *um²*	409500 *ns*	*proven* - 1 *min*	*proven* - 28 *min*
UROL16	111632 *um²*	313500 *ns*	*proven* - 1 *min*	*proven* - 33 *min*
UROL32	169949 *um²*	177250 *ns*	*proven* - 1 *min*	*proven* - 33 *min*

Name	**Area**	**Latency**	**Equivalence**	**Bug Description**
UROL#2	55927 *um²*	481500 *ns*	–	–
BUG#1	51189 *um²*	497500 *ns*	*cex* - <1 *min*	+ instead of *
BUG#2	99600 *um²*	581500 *ns*	*cex* - <1 *min*	aggressive loop unrol.
BUG#3	51831 *um²*	493250 *ns*	*cex* - <1 *min*	wrong index initial.
BUG#4	51851 *um²*	497500 *ns*	*cex* - <1 *min*	stuck-at-0 (16 bits)
BUG#5	49880 *um²*	497500 *ns*	*cex* - <1 *min*	stuck-at-1 (16 bits)

GRAY 64×64

	Entire Module			Compute Process			
Name	**FFs**	**Gates**	**Lines**	**FFs**	**Gates**	**Lines**	**Outputs**
REF	14489	∼52K	∼116K	6478	∼15K	∼12K	128
UROL64	32867	∼820K	∼120K	24856	∼787K	∼15K	128

Name	**Area**	**Latency**	**Bounded Proof**
REF	95869 *um²*	1914500 *ns*	–
BUG#1	95226 *um²*	1914500 *ns*	*cex* - 2 *min*
UROL64	336192 *um²*	665250 *ns*	*proven* - 3 *min*

GRAY 128×128

	Entire Module			Compute Process			
Name	**FFs**	**Gates**	**Lines**	**FFs**	**Gates**	**Lines**	**Outputs**
REF	28034	∼79K	∼186K	12588	∼64K	∼22K	256
UROL128	65136	∼1600K	∼193K	49690	∼1560K	∼28K	256

Name	**Area**	**Latency**	**Bounded Proof**
REF	179457 *um²*	7532500 *ns*	–
BUG#1	179244 *um²*	7532500 *ns*	*cex* - 2 *min*
UROL128	664151 *um²*	2601250 *ns*	*proven* - 10 *min*

Fig. 5. Results for GRAY 32×32, 64×64 and 128×128.

We focused the DSE and verification on the compute process because the load and store processes contain only regions that have a fixed latency. We synthesized three versions with private arrays of size $N = 32$, 64 and 128. Larger values of N increase the verification complexity. For all these experiments, we use the verification wrapper shown in Fig. 4 (the wrapper in Fig. 3 is used for the results in Section V). We used *Stratus HLS* for synthesis and *JasperGold* for equivalence and model checking.

Experimental Results. Fig. 5 shows the experimental results. The first three tables report the results for the case $N = 32$. The first table reports the characteristics of two representative RTL implementations (the largest and the fastest) in terms of number of flip-flops, gates and lines of synthesized RTL Verilog code, by considering the entire module or only the compute process. For the compute process we also indicate the number of outputs, i.e., the number of properties that need to be proven for equivalence. By considering the compute process only, KAIROS significantly reduces the amount of code to check (the lines of code are reduced by up to 88.3%). The second table reports the results of the DSE in terms of area and effective latency (as reported by *Stratus HLS*) and average verification time per property (as reported by *JasperGold*) for bounded and exhaustive proofs. For the bounded proofs [16], we set the bound equal to the number of clock cycles necessary to compute one iteration of the compute process. First, we synthesized and validated a reference implementation $\mathcal{I}_{ref}$ called *REF*: we verified its correctness by using a combination of formal and semi-formal methods. We synthesized five correct implementations in addition to REF. These implementations unroll the loops for different numbers of iterations (we checked

them against REF). There is a difference of ∼100 clock cycles between the slowest and the fastest implementation for the execution of a single iteration of the compute process. The results are reported in the second table of Fig. 5. KAIROS finds an exhaustive proof for all the cases. We also synthesized some bugged RTL implementations (checked against UROL#2) by manipulating the code (BUG#1, BUG#3), by injecting faults (BUG#4, BUG#5) or by applying wrong HLS knobs (BUG#2). The bugs in BUG#4 and BUG#5 are very unlikely to be detected with simulation-based approaches because they are activated only when specific values are observed in input. We introduced these bugs to represent (1) possible errors that the designer could introduce with manual manipulations or applications of knobs, and (2) faults that could be introduced by the HLS tool. In all cases, KAIROS finds a counterexample in less than a minute. The last four tables of Fig. 5 report the results for the cases $N = 64$ and $N = 128$. In both cases, we considered one correct implementation and one bugged implementation in addition to REF. Again, KAIROS finds bounded proofs in few minutes and detects the bugs in less than two minutes.

Remarks. The results reported in Fig. 5 use the same reference implementation to check the equivalence of all implementations (worst case). By minimizing the difference in latency between $\mathcal{I}_{ref}$ and $\mathcal{I}_k$, it is possible to reduce the verification time. In the case of GRAY, the verification of UROL#16 and UROL#32 against REF takes 33 minutes each (see Fig. 5), while verifying UROL#16 against UROL#32 takes only 20 minutes. We observed similar results for other combinations of implementations.

V. The RISC-V Case Study

We evaluated KAIROS also on a RISC-V processor designed in *SystemC* with HLS [17]. While processor design is not a typical target of HLS, it offers insights on how KAIROS works on control-dominated designs. We designed a pipelined 5-stage in-order processor that implements the RV32IM subset of RISC-V [18]. The design is organized in three modules (with a single process each): *fedec* implements the fetch and decode stages, *execute* implements the execution stage, and finally *memwb* implements the memory and writeback stages. The use of LID allows the processor to tolerate any latency variation in its computation. We focused the DSE and the verification on the execute module, and in particular on the division operations, which in our case work on 8-bit integer values. We used the same setup discussed in Section IV. We used the wrapper that handles data dependencies (Fig. 3) since each instruction can have a different latency.

Fig. 6 shows the results of the verification of the RISC-V processor with the same format of the tables reported in Fig. 5. We synthesized and validated a REF implementation. Then, we synthesized other four correct implementations (checked against REF). ARDIV relaxes the constraint on the latency for implementing the division loop, resulting in a longer execution time; PDIV1 and PDIV2 are pipelined implementations of the division with initiation interval of one and two, respectively; UDIV4 unrolls the division loop four times. In all cases, KAIROS finds an exhaustive proof showing that, independently from the latency

	Original Design			Execute Stage			
Name	**FFs**	**Gates**	**Lines**	**FFs**	**Gates**	**Lines**	**Outputs**
REF	3161	∼4700	∼16K	588	∼1000	∼4K	11
UDIV4	3160	∼4800	∼16K	587	∼1100	∼4K	11

Name	**Area**	**Latency**	**Exhaustive**
REF	27138 *um²*	324780 *ns*	–
ARDIV	27141 *um²*	485020 *ns*	*proven* - 33 *min*
PDIV1	27108 *um²*	363840 *ns*	*proven* - 1 *min*
PDIV2	27122 *um²*	384870 *ns*	*proven* - 1 *min*
UDIV4	27499 *um²*	244660 *ns*	*proven* - 1 *min*

Name	**Area**	**Latency**	**Equivalence**	**Bug Description**
REF	27138 *um²*	324780 *ns*	–	–
BUG#1	27044 *um²*	324780 *ns*	*cex* - < 1 *min*	swap remainder/division
BUG#2	27139 *um²*	324780 *ns*	*cex* - < 1 *min*	wrong loop comparison
BUG#3	27119 *um²*	324780 *ns*	*cex* - < 1 *min*	wrong bit shifting (1 bit)
BUG#4	27149 *um²*	304750 *ns*	*cex* - < 1 *min*	wrong loop condition
BUG#5	27158 *um²*	324780 *ns*	*cex* - < 1 *min*	stuck-at on numerator

Fig. 6. Experimental results of the RISC-V processor core.

required for the division, the different implementations are latency-insensitive equivalent. We designed also some bugged implementations (checked against REF). Among them, BUG#5 contains a bug that is unlikely to be found with simulation. KAIROS detects all the bugs in less than one minute per bug.

VI. Related Work

KAIROS is inspired by the idea of incremental HLS [12], which is about applying "incrementality" to reduce synthesis times. In contrast, KAIROS combines it with LID compositionality to verify the synthesis results. Several verification methods can be adopted in HLS. Some methods can be used to check the high-level specification *before synthesis*. For example, there exist techniques for bounded model checking of *C* and *SystemC* programs [19], [20], [21], [22] and for checking the equivalence of *C* programs [23], [24]. All these techniques are complementary to KAIROS. They can identify bugs in the high-level specification, but they cannot guarantee the correctness of the RTL code. Other methods can verify the correctness of the *synthesis* step [25], [26], [27], [28], for example, by using translation validation [9], [29], [30], [31] or intermediate models [32], [33], [34]. There are also techniques to verify the correctness of some specific HLS optimizations [35], [36]. All these methods can be integrated in KAIROS. In fact, they can be used to verify the correctness of the reference implementation obtained from the high-level specification, while the successive modifications to such implementation can be checked more efficiently with KAIROS, which exploits LID compositionality. Finally, there are methods for *post-synthesis* validation [16] to check that the RTL implementations satisfy formal properties or to check the equivalence of two implementations. KAIROS leverages these methods by using a commercial equivalence checker for RTL-to-RTL equivalence.

VII. Concluding Remarks

We described KAIROS, a formal methodology for automatic incremental verification in HLS. We showed that KAIROS can quickly detect bugs in a hardware module designed with HLS. We also discussed a case study where we verified multiple RTL implementations of a RISC-V processor core.

ACKNOWLEDGMENTS

We would like to thank Georgios Charitos for some preliminary analysis, Michael Theobald, Paolo Mantovani and Davide Giri for the helpful discussions and feedback, and Robert Margelli for providing the RISC-V case study. This work was supported in part by the NSF (A#: 1764000) and DARPA (C#: HR0011-18-C-0122).

REFERENCES

[1] L. P. Carloni. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proceedings of the IEEE*, 2015.

[2] B. Khailany, E. Krimer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao, S. Srinath, C. Torng, S. Xi, Y. Zhang, and B. Zimmer. A Modular Digital VLSI Flow for High-Productivity SoC Design. In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2018.

[3] L. Piccolboni, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 2017.

[4] D. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up (Second Edition)*. Springer, 2009.

[5] MatchLib, NVIDIA Research. https://github.com/NVlabs/matchlib.

[6] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency Insensitive Protocols. In *Proc. of the International Conference on Computer-Aided Verification (CAV)*, 1999.

[7] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of Latency-insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2001.

[8] L. P. Carloni. The Role of Back-Pressure in Implementing Latency-Insensitive Design. In *Proc. of the International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Architectures (FMGALS)*, 2006.

[9] S. Kundu, S. Lerner, and R. K. Gupta. Translation Validation of High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2010.

[10] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[11] A. Takach. High-Level Synthesis: Status, Trends, and Future Directions. *IEEE Design & Test*, 2016.

[12] L. Lavagno, A. Kondratyev, Y. Watanabe, Q. Zhu, M. Fujii, M. Tatesawa, and N. Nakayama. Incremental High-level Synthesis. In *Proc. of the ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[13] P. Coussy and A. Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer Science & Business Media, 2008.

[14] V. G. Oklobdzija, V. M. Stojanovic, D. M. Markovic, and N. M. Nedovic. *Digital System Clocking: High-Performance and Low-Power Aspects*. IEEE Press, 2003.

[15] E. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2003.

[16] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1999.

[17] R. Margelli. System-level Design of a Latency-insensitive RISC-V Microprocessor and Optimization via High-level Synthesis. *Master's Thesis, Politecnico di Torino*, 2017.

[18] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, 2014.

[19] H. Rocha, H. Ismail, L. Cordeiro, and R. Barreto. Model Checking Embedded C Software Using k-Induction and Invariants. *Embedded Software Verification and Debugging*, 2017.

[20] D. Kroening and M. Tautschnig. CBMC - C Bounded Model Checker. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014.

[21] C-N. Chou, Y-S. Ho, C. Hsieh, and C-Y. Huang. Symbolic Model Checking on SystemC Designs. In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2012.

[22] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: A Software Model Checking Approach. In *Proc. of the ACM/IEEE International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2010.

[23] H. Yoshida and M. Fujita. Rule-based Equivalence Checking of System-level Design Descriptions. In *Proc. of the IEEE International Conference on Communications, Circuits and Systems (ICCCAS)*, 2009.

[24] N. Thole, H. Riener, and G. Fey. Equivalence Checking on ESL Utilizing a Priori Knowledge. In *Proc. of the IEEE Forum on Specification and Design Languages (FDL)*, 2016.

[25] D. Kroening and E. Clarke. Checking Consistency of C and Verilog Using Predicate Abstraction and Induction. In *Proc. of the ACM/IEEE International Conference on Computer-aided Design (ICCAD)*, 2004.

[26] S. Vasudevan, V. Viswanath, J. A. Abraham, and J. Tu. Sequential Equivalence Checking Between System Level and RTL descriptions. *Design Automation for Embedded Systems*, 2008.

[27] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma. Non-cycle-accurate Sequential Equivalence Checking. In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2009.

[28] R. Mukherjee, D. Kroening, T. Melham, and M. Srivas. Equivalence Checking Using Trace Partitioning. In *Proc. of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2015.

[29] T. Li, Y. Guo, W. Liu, and C. Ma. Efficient Translation Validation of High-Level Synthesis. In *Proc. of the IEEE International Symposium on Quality Electronic Design (ISQED)*, 2013.

[30] T. Li, Y. Guo, S. Li, and Q. Tan. Equivalence Checking of Scheduling in High-Level Synthesis. In *Proc. of the IEEE International Symposium on Quality Electronic Design (ISQED)*, 2015.

[31] A. Leung, D. Bounov, and S. Lerner. C-to-Verilog Translation Validation. In *Proc. of the ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015.

[32] C. Karfa, C. Mandal, and D. Sarkar. Formal Verification of Code Motion Techniques Using Data-flow-driven Equivalence Checking. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2012.

[33] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar. An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2008.

[34] S. Kundu, S. Lerner, and R. Gupta. Automated Refinement Checking of Concurrent Systems. In *Proc. of the ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2007.

[35] K. Hao, S. Ray, and F. Xie. Equivalence Checking for Behaviorally Synthesized Pipelines. In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2012.

[36] Z. Yang, S. Ray, K. Hao, and F. Xie. Handling Design and Implementation Optimizations in Equivalence Checking for Behavioral Synthesis. In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2013.