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 regions1. 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.

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...
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 $I_k$ has been synthesized, KAIROS identifies which regions of $I_k$ 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 $I_{ref}$. 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.

STEP #2: Check Regions Equivalence. After identifying a pair of regions in $I_k$ and $I_{ref}$, 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$_1$ ≠ 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 $b^i_1$ and $b^i_2$ are the output values produced by $\text{REGION#1}$ and $\text{REGION#2}$ corresponding to input $a^i$, respectively. The wrapper synchronizes the regions, so that the model checker can verify if $b^i_1 = b^i_2 \forall i$. 

<table>
<thead>
<tr>
<th>cycles</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>$a^0$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>out$_1$</td>
<td>-</td>
<td>$b^0_1$</td>
<td>$b^1_0$</td>
<td>$b^0_1$</td>
<td>-</td>
<td>$b^1_1$</td>
<td>$b^1_2$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>out$_2$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$b^1_2$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>equiv</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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).
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_valid1) and the slower region (out_valid2) 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]. 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 \(Z_{ref}\) 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...
We synthesized and validated a REF UDIV4 8 V [18]. The design is organized in three modules (with a single simulation-based approaches because they are activated (worst case). By minimizing the difference in latency between the cases.

**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 $I_{rej}$ and $I_k$, 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 divide 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 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


