Portland State University

PDXScholar

Computer Science Faculty Publications and Presentations

Computer Science

2023

An Equivalence Checking Framework for Agile Hardware Design

Yanzhao Wang
Portland State University, wyanzhao@pdx.edu

Fei Xie
Portland State University, xie@pdx.edu

Zhenkun Yang
Intel

Pascuale Cocchini
Intel

Jin Yang
Intel

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Citation Details

Wang, Y., Xie, F., Yang, Z., Cocchini, P., & Yang, J. (2023, January). An Equivalence Checking Framework for Agile Hardware Design. In Proceedings of the 28th Asia and South Pacific Design Automation Conference (pp. 26-32).

This Conference Proceeding is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

An Equivalence Checking Framework for Agile Hardware Design

Yanzhao Wang and Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97229, USA

Zhenkun Yang, Pasquale Cocchini, and Jin Yang
Strategic CAD Labs
Intel Corporation
Hillsboro, OR 97124, USA

Abstract

Agile hardware design enables designers to produce new design iterations efficiently. Equivalence checking is critical in ensuring that a new design iteration conforms to its specification. In this paper, we introduce an equivalence checking framework for hardware designs represented in HalideIR. HalideIR is a popular intermediate representation in software domains such as deep learning and image processing, and it is increasingly utilized in agile hardware design. We have developed a fully automatic equivalence checking workflow seamlessly integrated with HalideIR and several optimizations that leverage the incremental nature of agile hardware design to scale equivalence checking. Evaluations of two deep learning accelerator designs show our automatic equivalence checking framework scales to hardware designs of practical sizes and detects inconsistencies that manually crafted tests have missed.

CCS Concepts

• **Software and its engineering** → Compilers; Software verification; • **Hardware** → Equivalence checking.

Keywords

Equivalence Checking, Halide, Agile Hardware, Formal Verification

ACM Reference Format:
Yanzhao Wang and Fei Xie and Zhenkun Yang, Pasquale Cocchini, and Jin Yang. 2023. An Equivalence Checking Framework for Agile Hardware Design. In *28th Asia and South Pacific Design Automation Conference (ASPDAC '23), January 16–19, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3566097.3567843

1 Introduction

While advances in software engineering have enabled developers to create new iterations of applications in a matter of weeks or months, designing hardware still takes large teams years of effort. Such lengthy and costly design cycles substantially limit hardware innovation. Agile hardware design is a promising approach that allows designers to efficiently produce a new design without compromising quality [2, 14, 16]. It has two significant advantages:

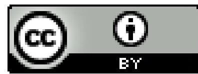
(1) Design agility: Designers can experiment at a high level of abstraction to explore design alternatives.
(2) Implementation agility: Various platform-specific implementations can be easily generated from high-level designs.

HalideIR [20] is a popular intermediate representation (IR) in software domains such as deep-learning [11] and image processing [7]. It is gaining traction in agile hardware design [14, 15, 19]. HalideIR enables agile design because it separates the specification of an algorithm from its execution schedule, i.e., code specifying loop nesting, loop unrolling, parallelization, vector operations, etc. Since changing the schedule does not require modification of the algorithm, designers can efficiently experiment with various optimizations for the algorithm. HalideIR also enables agile implementation through its well-designed transformation pipelines that help lower designs on to various implementation platforms such as C/C++/SystemC and RTL.

Central to the practical application of agile hardware designs is seamlessly integrated design verification that ensures uncompromised design quality. Equivalence checking is a key design verification method that checks if two designs exhibit the same behavior [12]. It plays a critical role in ensuring that new design iterations generated by agile hardware design are functionally equivalent to previous iterations and the original specification.

In this paper, we introduce an equivalence checking framework for agile hardware design, particularly targeting designs represented in HalideIR. Given two iterations of the same design in HalideIR, our equivalence checking framework can symbolically execute them on the same set of symbolic inputs and compare if they produce equivalent outputs. A naïve input-output symbolic comparison may lead to path explosion and does not scale to hardware designs of practical sizes. To enable scalable equivalence checking, we have developed an incremental checking algorithm that leverages the structural similarity of two design iterations. The algorithm analyzes the HalideIRs of the two design iterations to identify minimal design constructs that must be checked, and our equivalence checker is applied accordingly to discharge such localized checks. Since these design constructs may not be complete designs, test harnesses are required to wrap them into self-contained designs. We have developed an algorithm to automatically generate test harnesses, substantially reducing manual efforts. Practical hardware designs may contain many modules and complex computations, which can lead to path explosions. Hence, we have implemented an optimization using automatic uninterpreted function [9].

Our framework can support any HalideIR-based agile hardware design framework such as HeteroCL [14], HeteroHalide [15], and Halide-HLS [19]. In this paper, we use HeteroCL as an example. Our equivalence checking algorithms and optimizations are built on HalideIR and not tightly coupled with HeteroCL; therefore, the resulting automatic equivalence checking workflow can be seamlessly integrated with other HalideIR-based agile hardware design frameworks. We have applied our equivalence checking



This work is licensed under a Creative Commons Attribution International 4.0 License.
ASPDAC '23, January 16–19, 2023, Tokyo, Japan
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9783-4/23/01.
https://doi.org/10.1145/3566097.3567843

framework to two HeteroCL implementations [4] of an open-source deep-learning accelerator, Versatile Tensor Accelerator (VTA) [18]. VTA represents hardware designs of modest but practical sizes. These two HeteroCL designs represent varying design iterations of VTA. Evaluations have shown that our approach can automatically check the equivalence of these two designs with the original VTA (also modeled in HeteroCL) with modest time and memory usage. Our framework has detected design inconsistencies that manually crafted equivalence test suites fail to identify.

2 Background and Related Work

2.1 Agile Hardware Design with HeteroCL

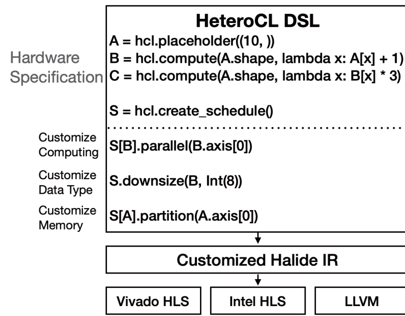


Figure 1: Agile hardware design with HeteroCL

Figure 1 depicts an agile hardware design workflow using HeteroCL. HeteroCL features a Python-based domain-specific language (DSL) for specifying hardware designs on a high abstraction level. It utilizes HalideIR as its intermediate representation in the compilation from this DSL to a lower-level design language such as C/C++/SystemC. In this example, the first part of the specification defines the computations: addition and multiplication. The second part creates an execution schedule for the computations and dictates parallelism, data types, and memory layout. For compilation, this specification is first transformed into HalideIR. Then, HeteroCL generates different target implementations (e.g., Intel HLS) from the HalideIR using different backends.

2.2 Related Work

There are commercial solutions for equivalence checking, such as Mentor's SLEC [5] and Synopsys's Hector [3], primarily targeting general-purpose RTL-RTL and RTL-C equivalence checking.

There has been studies on equivalence checking between high-level designs. Yang, et al. proposed a scalable equivalence checking framework to validate hardware designs in high-level description by comparing system states of designs through symbolic execution [23]. However, it does not scale well to HalideIR designs that often implement deep-learning and image processing algorithms which feature many loop structures. Previous research [8, 13, 21] also attempted to exploit structural similarities between designs to reduce complexities in equivalence checking. However, their approaches only work on RTL and do not apply to higher-level design representation, such as HalideIR. CoSA [17] implemented a formal verification framework for an agile hardware design framework, AHA [2]. CoSA carries out verification on CoreIR [1] and requires significant manual efforts.

Our work targets HalideIR, a higher-level design representation, and leverages the incremental nature of agile hardware design to implement an automatic and scalable equivalence checking workflow. To the best of our knowledge, there is no other equivalence checking framework for hardware designs represented in HalideIR.

3 Equivalence Checking Framework

3.1 Overview

Designers can efficiently produce new design iterations by agile hardware design. To ensure design correctness, equivalence checking must be conducted between these design iterations. We adopt the definition of functional equivalence between designs as in [22]. In short, two designs are considered functionally equivalent if they generate equivalent outputs given the same symbolic inputs.

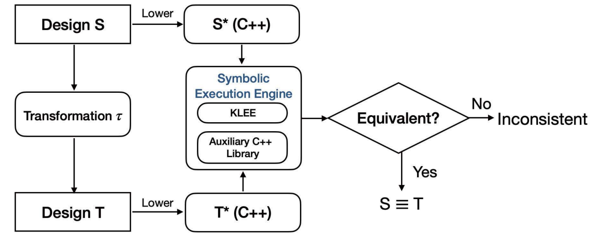


Figure 2: Equivalence checking framework

Figure 2 depicts the workflow for our equivalence checking framework, where a symbolic execution engine checks whether the synthesizable C++ generated from the HalideIRs of two designs (referred to as S^* and T^*) produces the same outputs, given the same symbolic inputs. If the check succeeds, designs S and T are equivalent; otherwise, we attempt to find the reasons for the divergence in behavior. Our symbolic execution engine consists of KLEE [10] and an auxiliary C++ library supporting arbitrary data types, FIFO, etc. in synthesizable C++, which the standard C++ library does not support. Figure 3 shows an end-to-end example of equivalence checking for two designs. First, we lower the HalideIRs of two designs to synthesizable C++, then we use KLEE to verify whether given the same symbolic inputs, the A arrays, the outputs of the two designs, the $Multiply$ arrays, are equivalent. If the check succeeds, the two designs are equivalent; otherwise, our framework flags an inconsistency.

There are two major limitations in this straightforward approach to equivalence checking:

(1) **Checking the entire designs directly is not scalable:** The approach of directly comparing the entire states of two designs does not scale up to complex designs, because it can easily lead to path explosions for any non-trivial designs. Another problem with this approach is that it is difficult to locate the points of divergence between the designs being compared, making debugging very challenging.
(2) **Creating test harnesses requires major manual efforts:** Checking synthesizable C++ code with our symbolic execution engine requires time-consuming and error-prone work

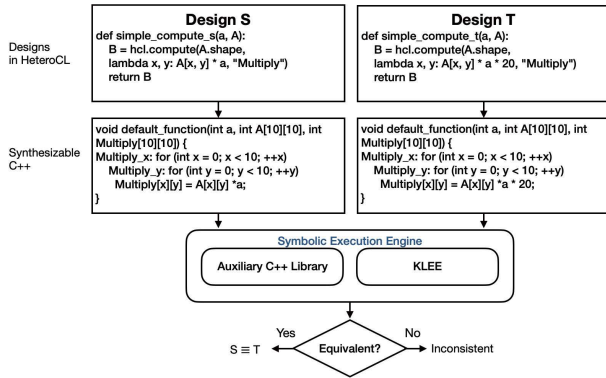


Figure 3: An end-to-end equivalence checking example

in creating test harnesses that include symbolic inputs, outputs, and wrapper code. This work needs to be repeated whenever there is a new design iteration to verify.

We have made three optimizations to address these limitations:

(1) **Identification of minimal check units**: To avoid path explosions, it is necessary to decompose the problem into smaller, more manageable, and independent *check units*. In the agile hardware design workflow, adjacent iterations of designs typically have high structural similarities, we utilize this characteristic to find those check units. For this purpose, we have identified the HalideIR **stage**, a group of operations that usually correspond to limited lines of code, as an appropriate minimal check unit. Thus, the process of checking the equivalence of an entire design can be reduced to checking individual minimal check units. And the use of a *stage* as the check unit also facilitates locating the specific operations that cause divergences in design behavior.

(2) **Automatic uninterpreted function optimization**: A HeteroCL design may contain thousands of *stages* and complex computations, all these complexities contribute to path explosions. To certify a high-level *stage*, e.g., the Top *stage* in Figure 4, its minimal check unit may contain thousands of sub-stages in practical designs. Thus, we employ the automatic uninterpreted function optimization, i.e., if we have certified *sub-stages* of a *stage*, we can replace those *sub-stages* with equivalent uninterpreted functions.

(3) **Automatic test harness generation**: We implement an automatic test harness generator to reduce manual work. To automatically generate a test harness, we need to identify input and output variables to minimal check units, because each check unit may use variables defined outside of the unit or produce variables that will be used by other units.

In the next sections, we explain how these optimizations enable automatic and scalable equivalence checking.

3.2 Identification of Minimal Check Units

A **stage** in HalideIR represents a group of operations while a design contains many stages. As shown in Figure 4, design S has two operations: vector addition B and vector subtraction C, for which

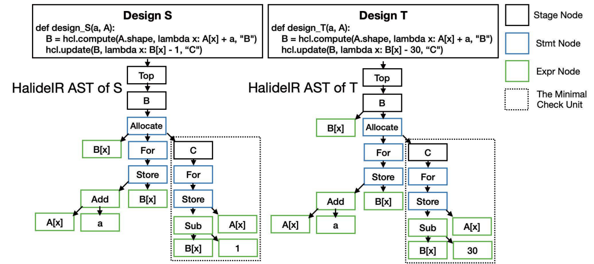


Figure 4: Example of HalideIR stages and IR checker

HeteroCL creates IR stages B and C respectively. Each stage node has two children: an operation node and an optional stage node.

There are two main reasons for choosing a stage as the minimal check unit for equivalence checking: First, a stage is a fundamental computational unit in HalideIR. A design in HalideIR can be split into multiple stages. Lowering the checking from the design level to the stage level can reduce a complex design checking problem into multiple stage checking problems that are much easier to solve while guaranteeing the completeness of equivalence checking for the entire design. Second, since each stage represents a specific computation in HeteroCL, we can isolate and map detected inconsistencies back to specific operations as part of a stage. This makes it easier for designers to pinpoint the causes of inconsistencies.

To identify minimal check units, we implement an IR checker which traverses the abstract syntax trees (AST) of the HalideIRs for two designs under check to find stages that may have structural and behavioral divergences. Figure 4 illustrates how the IR checker works. First we lower a HeteroCL design into the HalideIR. Then, starting from the root node (stage Top) in Figure 4, the checker traverses the AST in post-order algorithm while examining the attributes for each node, for instance, the *Sub* node in our example.

```
1  bool IRChecker::visit(const Sub *nodeS, const Sub *nodeT) {
2    if (!compare_expr(nodeS->left, nodeT->left))
3      return false;
4    if (!compare_expr(nodeS->right, nodeT->right))
5      return false;
6    return true;
7  }
```

The code above illustrates how the IR checker operates during the traversal, using the *Sub* node as an example. Upon visiting the *Sub* node, its left and right child nodes are checked for structural equivalence. If they are consistent in both designs, the Sub nodes are considered consistent; otherwise, the checker reports them as structural inconsistent nodes and flags their stages. In the example shown in Figure 4, while checking the *Sub* nodes, we detect a structural inconsistency: the right child nodes of the *Sub* nodes do not match. Accordingly, the checker would flag stage C with all its children as the minimal check unit.

There are also cases where the structures of the IR ASTs differ significantly, and certain stages have no matching ones in other designs. For instance, as shown in Figure 5, the IR checker cannot find a matching stage in design S for stage Mul2 in the target design T. In such cases, the IR checker marks the parent stage of the unmatched node as structurally inconsistent. For instance, in Figure 5, the IR checker marks stage Mul1 as the divergence point with the minimal check unit consisting of stage Mul1 and all its sub-stages.

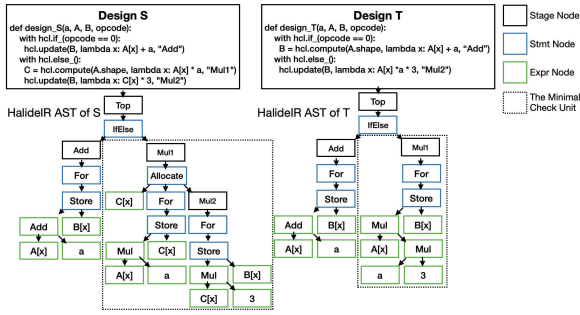


Figure 5: Example of designs with the same behavior but different IR structures

At the HalideIR level, even with structural inconsistencies, we cannot assert that the designs are inequivalent since the corresponding minimal check units in two designs could realize the same functionality even though they are structurally different, as in the case of Figure 5. Therefore, to certify the equivalence of the minimal check units, we will apply our symbolic execution engine as described in Figure 2 to them.

However, to apply our symbolic execution engine, the minimal check units must be self-contained designs. Therefore, we need to provide test harnesses to wrap these minimal check units automatically into self-contained designs, which we will discuss below.

3.3 Automatic Test Harness Generation

Figure 6 illustrates how a test harness wraps the synthesizable C++ generated from a pair of corresponding minimal check units in designs S and T into a self-contained design that can be checked by our symbolic execution engine. The test harness consists of three

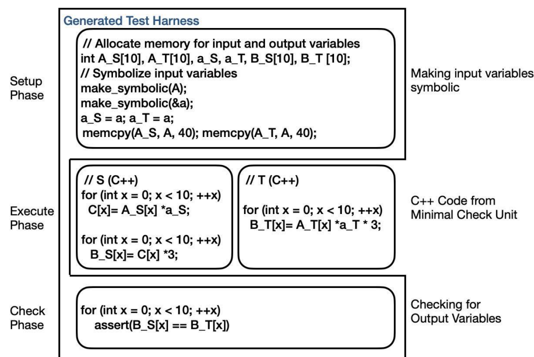


Figure 6: Generated test harness wrapping around synthesized C++ code of minimal check units of designs S and T

phases: setup, execute, and check. In the setup phrase, the input variables to the pair of corresponding minimal check units in S and T are assigned the same symbolic values accordingly. In the execute phase, the synthesizable C++ of S and T's minimal check units are invoked with the symbolic inputs, respectively. In the check phrase, the corresponding output variables of the minimal check units are checked for equivalence.

In generating the test harness, a key challenge is to identify the input and output variables of the minimal check unit. For design-level checking, identifying such variables are trivial since they can easily be extracted from the interface definitions of the design. However, we cannot directly derive those variables from a minimal check unit, because it is not a complete design but simply a code segment. We must consider the context of a minimal check unit in the entire design to derive its input and output variables.

We first define the input and output variables of a minimal check unit. HalideIR uses *Allocate* nodes to allocate variables and *Store* nodes to represent an operation that stores data into memory, which changes design states. Input variables are variables within the minimal check unit that is neither allocated nor written by the minimal check unit's internal *Allocate* or internal *Store* nodes. Take the minimal check unit of stage Mul1 in design S in Figure 7 as an example. A in the minimal check unit is neither allocated nor written by any *Allocate* or *Store* nodes within the unit. Therefore, it is an input variable. In contrast, output variables are variables within the minimal check unit that are written by the unit's internal *Store* nodes, but not allocated by the unit's internal *Allocate* nodes. For instance, B in the minimal check unit of stage Mul1 in design S from Figure 7 is an output variable.

We identify input and output variables of a minimal check unit by analyzing the variable dependency in HalideIR. First, we analyze the HalideIR of the design to find all variable nodes within the minimal check unit. For example, for stage Mul1 in design S from Figure 7, we have a variable set: A, B, and C. Then, we identify all variables that *Allocate* nodes allocate within the minimal check unit, in this case, C. Subsequently, we subtract the locally allocated variables from the variable set. Now we have identified all variables not allocated by *Allocate* nodes within the minimal check unit: A, and B. Since B is written by a *Store* node, it is marked as an output variable for the minimal check unit in design S, while the remaining variable, A, is marked as input. We repeat this process for design T. Next, we check if stage Mul1 in S and T have the same input and output variables. If they mismatch, we record the inconsistencies as warnings.

After we identify the input and output variables of minimal check units, then the test harness generator produces the synthesizable C++ code for the minimal check units of S and T. Lastly, it generates the checking code for output variables. Now, we have a complete executable program that contains the C++ code generated from minimal check units of S and T and the wrapping test harness. This program is readily consumable by the symbolic engine.

3.4 Automatic Uninterpreted Function Optimization

Figure 7 shows an example of replacing certified sub-stages with uninterpreted functions. Design S, and T have two inconsistent stages, Mul1 and Mul2. Since Mul1 includes Mul2, if we have certified stage Mul2 equivalent in both designs, when certifying Mul1, we can replace Mul2 in both designs with the same uninterpreted functions to reduce verification complexities. Since KLEE does not readily support uninterpreted functions, we provide a workaround: For each pair of certified minimal check units, we use KLEE to check their input variables' equivalence, remove all nodes in the minimal check units, and replace the units with equivalent symbols in both designs. This process can be executed recursively. We can make the

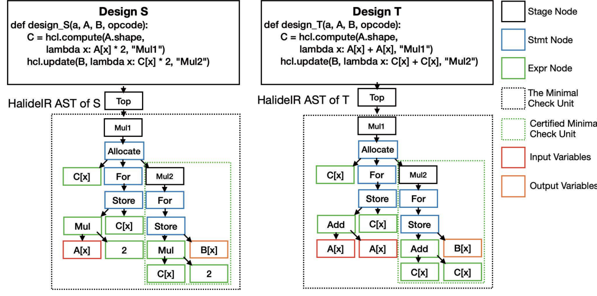


Figure 7: An example of replacing certified sub-stages with uninterpreted functions

equivalence checking combinational with automatic uninterpreted function optimization, helping handle complicated designs.

3.5 Optimized Equivalence Checking Framework

Figure 8 illustrates our optimized equivalence checking framework. Design Y is a newer design iteration produced by transformation

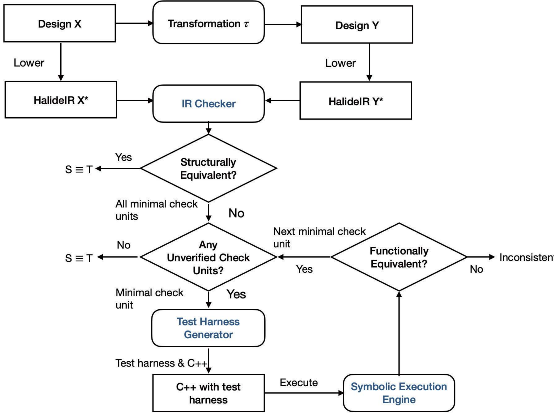


Figure 8: Optimized equivalence checking of designs X and Y, the input and output of design transformation τ respectively

τ from design X, where τ can be a manual or automated transformation, such as updating the hardware structures and algorithms or bringing new optimizations, such as adding new pipelines and unrolling loops. Instead of directly comparing the entire design Y against design X symbolically, we first use the IR checker to determine if the IRs (referred to as X^* and Y^*) of two designs are structurally equivalent. If yes, we can already report that X and Y are equivalent. Otherwise, we send the pairs of minimal check units in the two IRs, as identified by the IR checker, to the test harness generator. For each corresponding pair of minimal check units, the test harness generator produces a test harness that wraps the synthesized C++ code of this pair into a complete C++ program that the symbolic execution engine can consume. If all minimal check unit pairs are equivalent, the equivalence checking framework reports the designs as equivalent; otherwise, it reports the detected inconsistencies.

3.6 Integration with HalideIR-based Agile Hardware Design Frameworks

Figure 9 shows the integration between our equivalence checking framework and HalideIR-based agile hardware design frameworks, such as HeteroHalide [15], Halide-HLS [19], and HeteroCL [14]. Instead of applying to a specific agile hardware design framework, our equivalence checking framework broadly applies to any HalideIR-based agile hardware design framework. We first lower the designs into HalideIR and send HalideIR ASTs to our IR checker. The IR checker will identify inconsistent stages as minimal check units and send them to the test harness generator. The test harness generator will generate the test harness and combines it with C++ code from the minimal check units generated by HalideIR's backend. The symbolic execution engine then runs on the combined code to discharge the checks. Our approach can also be adapted to other agile hardware design frameworks, where we can identify structural similarities in their IRs.

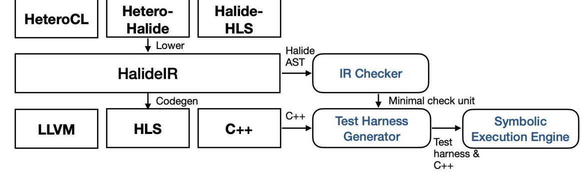


Figure 9: Integration of our equivalence checking framework with HalideIR-based agile hardware design frameworks

4 Evaluations

We have evaluated the effectiveness and efficiency of our equivalence checking framework using two HeteroCL implementations [4] of an open-source deep-learning accelerator: VTA [18]. VTA is composed of four modules: `fetch`, `load`, `compute`, and `store`. The `fetch` module fetches instructions from the DRAM, the `load` module loads data from DRAM to internal data buffers, the `compute` module includes an ALU (arithmetic logic unit) and a GEMM (general matrix multiplication) and carries out the computations, and the `store` module moves the computed results back to DRAM.

Researchers from Intel have implemented two versions of the VTA architecture using HeteroCL's Python-like DSL [4]: one is a sequential model of the VTA that processes one instruction at a time, and the other breaks down each of the 128-bit instructions into smaller micro-ops for potential parallelization. We refer to them as sVTA and uVTA in this paper. For comparison purposes, we also implemented a HeteroCL version of the VTA architecture strictly following its original structure [18], referred to as hVTA. To make sure hVTA is equivalent to the original VTA in C++, we symbolically executed hVTA and VTA function by function to verify their equivalence manually. Thus hVTA can serve as the specification for verification of sVTA and uVTA. We conducted this evaluation by checking the equivalence between sVTA and hVTA (sVTA-hVTA) as well as between uVTA and hVTA (uVTA-hVTA). The evaluation is done on a workstation with a 12-core AMD Ryzen 5900x CPU, 128 GB RAM, and Ubuntu 18.04 operating system.

Table 1 shows our evaluation results. We first run equivalence checking on the entire synthesizable C++ code generated from

Table 1: Evaluation of our framework without/with minimal check unit & uninterpreted function optimizations

Design	LoC Python	LoC C++	Minimal check unit & uninterpreted function optimizations	Time (s)	Memory (MB)	# of stages	# of structural inconsistencies	# of behavioral inconsistencies
sVTA-hVTA	296	560	No	timeout	6781.73	no data	no data	no data
			Yes	65.39	128.37	211	8	2
uVTA-hVTA	195	1224	No	timeout	7384.34	no data	no data	no data
			Yes	1238.38	2384.98	301	84	3

sVTA-hVTA and uVTA-hVTA, respectively. The symbolic execution engine timed out after 24 hours with more than 6 GB of memory consumption. With our optimization that breaks the checking problem into much smaller chunks at the *stage* level, the equivalence checking is completed in 65 seconds for sVTA-hVTA with 128.37 MB of memory, and 1238.38 seconds for uVTA-hVTA with 2384 MB of memory. We broke sVTA into 211 stages and detected 8 structural inconsistencies between sVTA and hVTA. After sending them to the symbolic execution engine, two behavioral inconsistencies are detected, which are flagged as bugs. For uVTA, we broke it into 301 stages and detected 84 structural inconsistencies between uVTA and hVTA; three of them caused behavioral inconsistencies. The results demonstrated that by checking the design at the *stage* level and lowering the complexity, we could tackle the equivalence checking problem for entire deep-learning accelerator designs, that are modest-size, and structurally distinct but functionally equivalent. The execution time of uVTA-hVTA is much longer than sVTA-hVTA because uVTA has significant structural differences from hVTA though they are functionally equivalent. uVTA still reuses many low-level functions in the original VTA as specified in hVTA, such as memory copy and computations, whose checking can be optimized with uninterpreted functions.

In our experiments, we detected one inconsistency in sVTA's `load` module and another one in `ALU` module. For uVTA, we detected two inconsistencies in `load` module and one in `ALU` module. Additionally, we found one bug in the HeteroCL framework. We have reported all detected inconsistencies and the bug to the developers. HeteroCL has marked the bug as high-priority [1].

The inconsistency in `ALU` module of uVTA and the one in `ALU` module of sVTA share the same cause, so do one of inconsistencies in uVTA's `load` module and the one in sVTA's `load` module. Therefore, in the following subsections, we will elaborate on inconsistencies in the `ALU` module of sVTA and `load` module of uVTA.

4.1 sVTA Inconsistency with hVTA in ALU

Our framework found one inconsistency in the `ALU` module of sVTA. The code fragment below is excerpted from this module.

```
1   ALU_OPCODE = hcl.scalar(instr[111:108], name="ALU_OPCODE")
2   # extend OPCODE to 3 bits
3   USE_IMM = hcl.scalar(instr[112:111], name="USE_IMM",
4            dtype=hcl.UInt(1))
5   IMM = hcl.scalar(instr[128:112], name="IMM")
6   src = hcl.select(USE_IMM.v == 1, hcl.cast(hcl.Int(16), IMM),
7         hcl.cast(hcl.Int(32), src_tensor[x][y]))
8   dst = hcl.cast(hcl.Int(32), dst_tensor[x][y])
9   with hcl.if_(ALU_OPCODE.v == VTA_ALU_OPCODE_MIN):
10       dst_tensor[x][y] = hcl.select(dst <= src, dst_tensor[x][y], src)
```

[1] https://github.com/cornell-zhang/heterocl/issues/258

At line 1, the `ALU_CODE` variable has 3 bits, while the corresponding variable in hVTA only has 2 bits [6]. The left-hand side in Figure 10

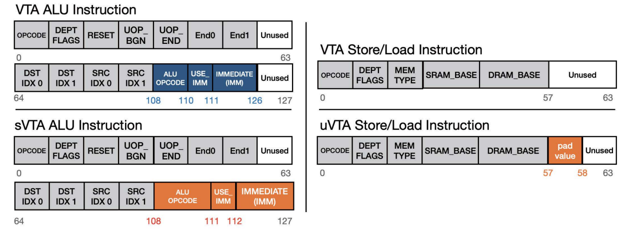


Figure 10: sVTA ALU instruction inconsistency / uVTA Load instruction inconsistency

shows the ALU instruction of hVTA and sVTA, and we can see that extending the `ALU_OPCODE` bits to 3 bits in sVTA causes `USE_IMM` and `IMM` fields to shift 1 bit, which is inconsistent with the original VTA instruction.

4.2 First uVTA-hVTA Inconsistency in load

Our framework found two inconsistencies in the `load` module of uVTA. Hereby we elaborate on the first one. As shown in the right-hand side of Figure 10, uVTA uses an extra bit `pad_value` that is not present in the original VTA instruction.

```
1   is_min_pad_value = hcl.scalar(instr[58:57])
2   pad_val = hcl.select(is_min_pad_value.v == 1,
3                        hcl.cast(hcl.Int(16),
4                        1 << (sram_bits - 1)), 0)
5   sram_idx = sram_base + x_tot * y + x
6   def clear(row, col):
7       sram[sram_idx][row][col] = pad_val
8   hcl.mutate((nrows, ncols), clear, name='pad_clear')
```

The code above is excerpted from the `load` module of uVTA. At line 1, the `load` module reads the extra bit from instructions. At line 2, the `pad_value` variable is assigned different values, according to the extra bit. And at line 6, the `load` module resets the `sram` with `pad_value`. However, according to the code below excerpted from the `load` module of hVTA, at line 3, `sram` is always reset with zeros.

```
1   for (int i = 0; i < range; i++)
2     for (int j = 0; j < MAT_AXI_RATIO; j++)
3       mem[sram_idx++][j] = 0;
```

Manually written unit tests failed to detect this inconsistency because those unit tests use VTA runtime to generate instructions. VTA runtime resets the instruction memory to zero before writing instructions. As a result, the `pad_value` bit in the uVTA instruction is always set to zero, which hides this inconsistency. This underlines the importance of applying formal equivalence checking.

4.3 Second uVTA-hVTA Inconsistency

Our experiments found another inconsistency in the `load` module of uVTA, which is caused by the following code fragment.

```
1   with hcl.if_(memtype == VTA_MEM_ID_WGT):
2     sram = wgt_mem
3   with hcl.else_():
4     sram = acc_mem
5   def fmutate_out(x,y,z):
6     sram[x][y][z] = 0
7   hcl.mutate(sram.shape, fmutate_out)
```

As the code above illustrates, the `sram` variable is assigned a different cache type according to the `memtype` condition. We expect that `wgt_mem`, and `acc_mem` should be set to zero according to the different `memtype`. However, in the synthesizable C++ generated by HeteroCL, only `acc_mem` was set to zero (See the code below).

```
1 for (int x = 0; x < range; ++x)
2   for (int y = 0; y < 1; ++y)
3     for (int z = 0; z < 16; ++z)
4       acc_mem[(y+x)][0][z] = 0U;
```

After we reported this inconsistency to the developers, they confirmed that it was caused by misuse of HeteroCL API. At line 2 of the above code, the developers used the assignment from Python to set the `sram`; instead, `hcl.update` should have been used.

4.4 Bug Analysis

The above examples represent two typical kinds of bugs that designers will encounter when conducting agile hardware design using HeteroCL. The first one is a violation of specifications, such as the bugs reported in Section 4.1 and Section 4.2. Developers may change their design implementations, such as updating algorithms, or apply new optimizations, such as adding new pipelines. Those changes may violate specifications. All those transformations will change the ASTs of the HalideIRs. Our approach uses an IR checker to detect AST changes, generate corresponding test harnesses, and check the equivalence of minimal check units. The second kind of bugs is API misuse, as in Section 4.3. HeteroCL provides similar APIs as Python's, and there are no compiler warnings in the design process. Therefore, developers may easily misuse Python's APIs in HeteroCL designs, causing implementation inconsistencies.

4.5 Discussions

When two designs have no similar structures, our equivalence checking framework will resort back to the basic approach of directly comparing the states of the two designs. However, in practice, the design agility of existing agile hardware design frameworks necessarily makes the adjacent iterations of the designs highly similar, which permits our framework's optimizations to function. Our structural decomposition following the AST of HalideIR ensures that our optimizations do not result in false positives or negatives.

5 Conclusions

In this paper, we have presented a scalable equivalence checking framework for HalideIR. It can automatically decompose design-level equivalence checking into stage-level checking and precisely identify inconsistencies between two hardware design iterations. We have demonstrated its effectiveness by performing equivalence checking on two practical deep-learning accelerator designs, sVTA and uVTA. Results show that our framework can check equivalence for designs of practical sizes and detect inconsistencies that manually written unit tests failed to detect. In future work, we will extend our framework to more domain-specific languages based on HalideIR, e.g., TVM [11], a popular deep-learning framework.

6 Acknowledgment

This research is partially supported by Semiconductor Research Corporation Contract: 2932.001 and a gift from Intel Corporation.

References

[1] Coreir. URL github.com/rdaly525/coreir.
[2] Aha agile hardware project. URL aha.stanford.edu.
[3] Vc formal datapath validation. URL synopsys.com/verification/static-and-formal-verification/vc-formal.html.
[4] Vta implementations in heterocl. URL github.com/pasqoc/incubator-tvm/tree/bsim_fpga.
[5] Questa sequential logic equivalence check. URL eda.sw.siemens.com/en-US/ic/questa/formal-verification/slec/.
[6] Vta hardware specification. URL github.com/apache/tvm/blob/v0.6/vta/include/vta/hw_spec.h.
[7] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.
[8] D. Brand. Verification of large synthesized designs. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 534–537. IEEE, 1993.
[9] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic (TOCL)*, 2(1):93–134, 2001.
[10] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.
[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.
[12] J.-H. Jiang and R. K. Brayton. On the verification of sequential equivalence. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6): 686–697, 2003.
[13] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th annual Design Automation Conference*, pages 263–268, 1997.
[14] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019.
[15] J. Li, Y. Chi, and J. Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 51–57, 2020.
[16] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni. Agile soc development with open esp. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.
[17] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan. Cosa: Integrated verification for agile hardware design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE.
[18] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. Vta: an open hardware-software stack for deep learning. *arXiv preprint arXiv:1807.04188*, 2018.
[19] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.
[20] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.
[21] C. Van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):814–819, 2000.
[22] Z. Yang, K. Hao, K. Cong, S. Ray, and F. Xie. Equivalence checking for compiler transformations in behavioral synthesis. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 491–494. IEEE, 2013.
[23] Z. Yang, K. Hao, K. Cong, L. Lei, S. Ray, and F. Xie. Scalable certification framework for behavioral synthesis front-end. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.