THESIS

FORMAL VERIFICATION OF SOURCE-TO-SOURCE TRANSFORMATIONS FOR HIGH-LEVEL SYNTHESIS

Submitted by

Emily Tucker

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2025

Master's Committee:

Advisor: Louis-Noël Pouchet

Vinayak Prabhu
Francisco Ortega
James Wilson

ABSTRACT

FORMAL VERIFICATION OF SOURCE-TO-SOURCE TRANSFORMATIONS
FOR HIGH-LEVEL SYNTHESIS

Hardware processors are designed using a complex optimization flow, starting from a high-level description of the functionalities to be implemented. This description is then progressively lowered to concrete hardware: Register-Transfer Level (RTL) functional behavior, timing between operations, and eventually actual logic gates are produced. High-level synthesis (HLS) can greatly facilitate the description of complex hardware implementations, by raising the level of abstraction up to a classical imperative language such as C/C++, usually augmented with vendor-specific pragmas and APIs. HLS automatically compiles a large class of C/C++ programs to highly optimized RTL. Despite productivity improvements, attaining high performance for the final design remains a challenge, and higher-level tools like source-to-source compilers have been developed to generate programs targeting HLS toolchains. These tools may generate highly complex HLS-ready C/C++ code, reducing the programming effort and enabling critical optimizations. However, whether these HLS-friendly programs are produced by a human or a tool, validating their correctness or exposing bugs otherwise remains a fundamental challenge.

In this work we target the problem of efficiently checking the semantic equivalence between two programs written in C/C++ as a means to ensuring the correctness of the description provided to the HLS toolchain, by proving an optimized code version fully preserves the semantics of the unoptimized one. We introduce a novel formal verification approach that combines concrete and abstract interpretation with a hybrid symbolic analysis. Notably, our approach is mostly agnostic to how control-flow, data storage, and dataflow are implemented in the two programs. It can prove equivalence under complex bufferization and loop/syntax transformations, for a rich class of programs with statically interpretable control-flow. We present our techniques and their complete end-to-end implementation, demonstrating how our system can verify the correctness of highly complex programs generated by source-to-source compilers for HLS, and detect bugs that may elude co-simulation.

ACKNOWLEDGEMENTS

First and foremost, I want to thank my advisor, Louis-Noël, who is the only reason I started (and finished!) this degree in the first place. His encouragement, support, and guidance, both in my research and in general, have been invaluable, and I'll be happy if I can become even half of the scientist, mentor, and human that he is.

I also want to thank my parents, John and Laura, and my brother, Ethan, for their support and for always encouraging me to go for it, even when I don't think I can.

Finally, thanks to my friends Janay, Carlie, Sophie, Ryland, and Jay for keeping me sane throughout my degree, and always listening to my ideas and rants about my research (despite them not making any sense most of the time): I couldn't have done this without them. And, of course, thanks to Greg and my shrimp for keeping me entertained.

TABLE OF CONTENTS

Chapter 1
Introduction

Whether to improve performance over a general-purpose CPU or to decrease energy consumption, the use of specialized hardware accelerators is becoming increasingly common [62]. These accelerators range from programmable accelerators, like GPUs and TPUs, that execute instructions, to application-specific accelerators whose behavior is controlled by their hardware implementation like FPGAs and ASICs. Building specialized hardware accelerators requires designing a description of the logic gates and their connections that implements the desired behavior of the accelerator, commonly specified in a hardware description language (HDL). To simplify the process of designing these accelerators, *high-level synthesis* (HLS) was introduced to compile a specification in a high-level language, usually C or C++, to an HDL description [57].

HLS has seen increasing adoption since its introduction as a means to decrease the development time required to implement hardware accelerators [25, 26, 51]. However, just as with producing high-performance executables for CPUs, producing high-performance hardware designs requires performing optimizing transformations on the input code. Some optimizations are performed by the HLS compiler, but to achieve maximum performance, designers apply transformations on the C/C++ source code before using HLS [58]. Source-level optimizations can be performed by source-to-source compilers like AutoSA [84], HeteroCL [49] or Merlin [90], performed manually by the designer, or even a combination of the two.

These transformations can either be the relatively simple addition of HLS-specific directives that specify how a construct in the source code (e.g. a loop or function) should be translated to hardware, or an extremely complex restructuring of the code that changes data type and layout, control flow and function structure, and I/O handling [49]. To maximize the performance of the design, these transformations also need to expose and exploit concurrency in the input code. Typically, designers start from a relatively simple input C/C++ specification and apply a sequence of transformations to produce one or many optimized versions of the code. This complex sequence of transformations can result in incorrect code, which leads to bugs that can only manifest after HLS, or even after the entire compilation process in the hardware itself, making these bugs difficult and time-consuming to find and fix.

For example, in many domains like machine learning, a commonly-used high-performance architecture is a systolic array [84], which is composed of many processing elements that perform computations in parallel and pass data between each other, as opposed to each element having random access to memory. To transform typical sequential array-based C/C++ code into C/C++ for HLS that uses a systolic array architecture, developers or compilers have to rewrite the input code to extract the operations that can be performed in parallel, as processing elements, and implement the data connections between elements. Incorrectly implementing these rewrites can result in incorrectly changing the results produced for certain inputs if the behavior of any processing element is incorrect, as well as concurrency bugs like deadlock or data races if the communication scheme is incorrect. Verifying the correctness of these extensive rewrites is crucial to avoid bugs in the resulting accelerator, but existing solutions like simulation [43] lack the ability to ensure the correctness of these transformations: they either only cover a small set of inputs or only support a small set of programs.

The aim of this work is to *formally verify* the correctness of source-to-source transformations performed on code for HLS by proving the semantic equivalence between the input source code and the output source code after optimizing transformations. We aim to verify that the two programs will, for the same inputs, produce the same outputs. This approach is independent of the specific HLS toolchain used, since it checks the equivalence of programs at the C/C++ level before HLS. We support complex transformations used for HLS, including loop transformations, local buffer insertion, and coarse-grained dataflow parallelism using FIFOs for communication and synchronization.

To accomplish this, we restrict the class of programs we consider to those which have *statically-interpretable control flow*: programs whose control flow can be computed at compile time, i.e. programs where every branch condition is independent of the input data. This restriction is common to impose on programs used to specify hardware accelerators: spatial architectures like systolic arrays have fixed sizes, and many other applications commonly targeted to accelerators such as video processing, convolutional neural networks, and large language models typically have static control flow. With this restriction, our system can *concretely* interpret the control flow of each program to build a *symbolic* representation of the expression that computes each output variable in terms of the input variables. The contributions of this work are as follows [65]:

- We present an end-to-end, fully implemented system to prove the equivalence between a pair of programs for HLS in the C/C++ language, under meaningful and practical restrictions.

- We combine partial concrete evaluation of specific program parts with a symbolic analysis to make the system robust to a rich set of code transformations, including key optimizations for HLS, such as loop transformations, bufferization, data layout changes, blocking/non-blocking FIFO communications, etc.
- We extend our hybrid concrete/symbolic interpreter to support the concurrency model commonly used in the design of hardware accelerators — coarse-grained dataflow with communication and synchronization handled by blocking fifos — and have the ability to check programs for possible race conditions and deadlock.
- We support, through user-specified rewrite rules and normalization for associativity and commutativity, the verification of transformations that change the order and number of operations performed in the program.
- We provide an extensive experimental evaluation that demonstrates the ability of our system to verify the correctness of large designs that involve major code restructuring, such as systolic arrays produced by the AutoSA compiler [84], an accelerator for the inference of a full BERT layer, and numerous numerical kernels optimized for HLS with HeteroCL [49].

This thesis covers the design, implementation, and evaluation of our hybrid concrete/symbolic interpretation and verification system as discussed above, and is organized as follows:

In Chapter 2, we discuss the typical hardware design flow and how HLS is used. We also survey several existing approaches for verifying the correctness of source-to-source transformations, and why their limitations prevent them from supporting the programs we target.

Chapter 3 contains a high-level overview of how our system works: a brief description of the programs we target, as well as a small example of how our system builds — during interpretation — the symbolic data structure that we use to check program equivalence, the computation directed acyclic graph (CDAG).

In Chapter 4, we provide the details of how the interpretation system works for sequential programs, including how the abstract syntax tree of the input program is traversed. This chapter contains a definition of the class of programs supported by our system: statically-interpretable control flow (SICF) programs.

In Chapter 5, we provide a formal definition of the CDAG, the symbolic representation of computations we use to check that output variables are equivalent for any value of the inputs to the program. We also discuss the complexity of this representation and how its worst-case complexity can be avoided.

In Chapter 6, we explain how CDAGs for different programs are compared after interpretation to either prove equivalence or conclude that the equivalence result is unknown. This also includes a post-interpretation normalization of CDAGs to support properties like associativity and commutativity of some operators and user-supplied rewrite rules.

In Chapter 7, we describe how our system supports interpretation of concurrent programs, the concurrency API we expose to users, and the details of how our system can detect data races and deadlocks to ensure the correctness of parallelizing transformations.

Chapter 8 provides an experimental evaluation of our approach: we evaluate our system's coverage and performance on a set of realistic benchmarks for hardware accelerators, including systolic arrays [84], inference of a layer of BERT, a large language model [28], and all PolyBench [64] kernels optimized using the HeteroCL compiler [49].

We conclude in Chapter 9 with a summary of the thesis and directions for future improvements to coverage and extensions to different languages and models.

Chapter 2
Background

We now present background on hardware design, including with HLS, and how to verify designs.

2.1 Hardware Design Flow

The flow to design hardware involves many steps. Figure 1 outlines the key steps of this flow. The input to this flow is a Register Transfer Level (RTL) specification. RTL models the design as registers and synchronous operations on those registers. RTL describes data movement in terms of signals and registers, as opposed to C which describes the logical operations to be executed.

The next step, logic synthesis, then takes RTL and produces logic gates implementing the operations. Existing tools for logic synthesis include commercial tools like AMD's Vivado [5], Synopsys Design Compiler [79], Cadence Genus [17], and open-source tools like Yosys [88]. After synthesis, the place and route stage the logic gate description and maps it to a physical location on a board and connects operations with wires. Synthesis and place and route can take hours to days for large designs, and possibly not even complete in some cases; for example, if the target board is too small to fit all the logic.

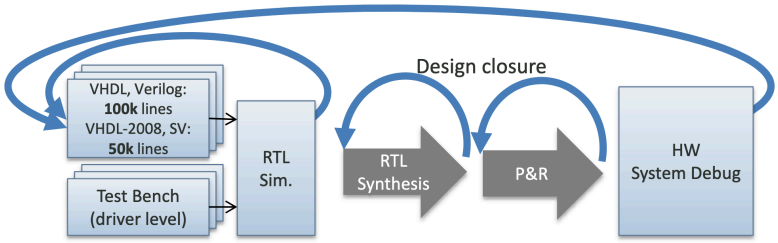

Figure 1: Major steps in the hardware design flow, adapted from [91].

An advantage of writing designs in RTL is that experienced designers have more fine-grained control over each and every operation implemented, and their schedule. A downside is the difficulty to implement transformations of the operations performed and their schedule. For example, even for codes that implement the same functionality, in C it takes 1 line and in RTL 20, as seen in Listing 1 that implements `c = a + b` extracted from [69].

```
module add (
    input [31:0] a,
    input [31:0] b,
    output [31:0] c,
    output c_vld,
    input ap_ce,
    input ap_rst,
    input ap_clk
);
    reg [31:0] c_d;
    reg c_vld_d;
    assign c = c_d;
    assign c_vld = c_vld_d;
    always @(posedge ap_clk) begin
        if (ap_rst == 1'b1) begin
        c_d <= 32'b0;
        c_vld_d <= 1'b0;
    end else begin
        c_d <= (a + b) & {32{ap_ce}};
        c_vld_d <= ap_ce;
```

Listing 1: A Verilog module that adds two 32-bit integers, `a` and `b`, and stores the output in `c`.

To produce high-performance designs, transformations that modify the storage and schedule (e.g. concurrency) of operations need to be performed. These transformations can involve global changes to the RTL program structure; for example, when implementing operation schedule transformations like loop tiling. Due to these transformations' complexity, their implementation can be error-prone. These errors would ideally be easier to track if they were implemented in a higher-level language, where they would require fewer lines changed in the program.

2.2 High-Level Synthesis

Instead of writing complex, signal-based RTL code as in Listing 1, High-Level Synthesis (HLS) enables users to use a high-level language such as C or C++ as input. HLS became mature enough to generate high-performance designs in the last decade, and is becoming adopted to generate high-performance accelerators especially on FPGAs. The success of AutoESL [26, 94], which was acquired by Xilinx to become the Vivado/Vitis high-level synthesis tool, helped democratize HLS in the hardware design flow. The HLS compiler converts C/C++ into optimized RTL, becoming a front-end for the hardware design flow and alleviating the need for the developer to manually write RTL code. Many HLS compilers exist,

including commercial solutions like Vitis HLS [6], Catapult [75], and the Intel HLS Compiler [40] as well as open-source compilers like LegUp [18] that all take C or C++ as input. Listing 2 is an example of a C function that multiplies a matrix and a vector that an HLS compiler could take as input.

```
void matvec_128(float* restrict A, float* restrict x, float* restrict y) {
    for (int i = 0; i < 128; ++i) {
        y[i] = 0;
        for (int j = 0; j < 128; ++j)
            y[i] += A[i * 128 + j] * x[j];
    }
}
```

Listing 2: A function that multiplies a matrix `A` and vector `x` written in C.

In practice, the input to the HLS compiler is a restricted subset of C/C++. HLS tools provide guidelines for how specific syntactic constructs will be synthesized, and directives (pragmas) to let the user indicate how constructs should be implemented in hardware. These pragmas can specify key parameters of the final hardware design (replication of units, concurrency, etc.). However, just specifying pragmas is usually not enough for the HLS compiler to produce a high-performance design; changes to the program besides pragmas are also necessary. For example, loop transformations, such as loop tiling or fusion, are necessary to change the schedule of operations and expose concurrency, but these transformations are typically not applied by the HLS compiler. Transforming a program in C or C++ is faster and easier than performing those same transformations on RTL code, but the HLS compiler can still take several hours or even days to compile a single design into RTL [66, 76].

When creating a hardware design for an algorithm description in C, the HLS tool has a lot of freedom to decide the data layout and operation schedule for the program. This design space is immense, and the tool may lack the ability to pick a good design [67, 70], resorting to design-space exploration instead [66, 76]. Performance goals also vary by design and user; a design optimizing for latency may look different from a design optimizing for resource usage (area), which may also look different from a design optimizing for power consumption. The tradeoff between these different aspects is something most users want control over, to generate the best possible hardware for their specific use case.

Listing 3 shows a simple transformation of Listing 2, a C specification of a matrix-vector multiplication function, but now modified for Vitis HLS. It illustrates some basic program transformations that

need to be implemented before HLS. The modified version now uses blocking FIFOs to communicate data, instead of memory buffers. HLS-specific directives (pragmas) to specify the hardware architecture are also inserted. This specific example uses the Vitis HLS directives and API, but other HLS tools have similar constructs.

One transformation applied in Listing 3 is hardware loop unrolling, a transformation where all iterations of the loop are replicated in hardware such that they execute in parallel, reducing latency but consuming more resources. The `#pragma HLS unroll` on line 16 of Listing 3 specifies that the enclosing `j` loop should be unrolled. Note that the `j` loop is a reduction loop: Vitis HLS can parallelize reductions [7]. `pragma HLS pipeline` on line 13 means the containing loop should be pipelined: each iteration of the loop starts immediately after the previous iteration, even if said iteration has not completed yet. Hardware resources and dependence between iterations actually control when iterations start executing. Finally, in the body of the top-level function `matvec`, the `#pragma HLS dataflow` on line 29 creates a dataflow region. The FIFO arguments to each function call under the pragma indicate the producer-consumer relationship between tasks: in this example, the call to `data_in` produces data sent through `FIFO_A` and `FIFO_x` that is then consumed by `matvec_core`. More information on the various other pragmas can be found in [7].

Often program transformations performed in practice result in even more complex code than in Listing 3. These transformations can either be done manually by designers or by source-to-source compilers designed for creating input to HLS. This makes bugs difficult to find; a bug introduced in a transformation of the source code might only be caught once the design has been compiled all the way to hardware.

2.3 Verification of Correctness of Transformations

As shown in Section 2.2, creating a hardware design involves many transformations to achieve high performance, even at the C level. The breadth and complexity of these transformations can make it easier for bugs to go unnoticed during development. These bugs can be costly, especially when designing hardware: if bugs are found after the hardware has been manufactured, producing fixed hardware can cost hundreds of thousands to millions of dollars [35]. To reduce the likelihood of bugs, designers spend a significant amount of development time and cost on verifying a design is correct according to its specification [35].

```
typedef hls::stream<float> FIFO_t;

void data_in(FIFO_t& FIFO_A, FIFO_t& FIFO_x, float* A, float* x) {
    for (int i = 0; i < 128; ++i)
        FIFO_x.write(x[i]);
    for (int ij = 0; ij < 128 * 128; ++ij)
        FIFO_A.write(A[ij]);
}

void matvec_core(FIFO_t& FIFO_A, FIFO_t& FIFO_x, float* x_buff, float* y) {
    for (int i = 0; i < 128; ++i)
        x_buff[i] = FIFO_x.read();
    for (int i = 0; i < 128; ++i) {
    #pragma HLS pipeline
        float y_temp = 0;
        for (int j = 0; j < 128; ++j) {
        #pragma HLS unroll
            y_temp += FIFO_A.read() * x_buff[j];
        }
        y[i] = y_temp;
    }
}

void matvec_128(float* restrict A, float* restrict x, float* restrict y) {
   FIFO_t FIFO_A, FIFO_x;
   float x_buff[128];
#pragma HLS dataflow
   data_in(FIFO_A, FIFO_x, A, x);
   matvec_core(FIFO_A, FIFO_x, x_buff, y);
}
```

Listing 3: A function that performs multiplication of matrix `A` and vector `x`, written in C++ using Vitis HLS directives and FIFOs.

2.3.1 Possible Approaches to Verification

Many approaches for software and hardware verification are used to ensure that designs are correct. These techniques vary in terms of coverage, time to complete, and applicability.

Testing and Simulation. The easiest and most common way to verify a design's functionality is through testing and simulation: verifying the correct output is produced for a single set of inputs. Testing increases confidence in a design, but can miss important corner cases that expose bugs. For example, Listing 2 is passing this test against a program doing no operations if one tests with `x` and `A` filled with zeroes.

In the context of hardware and HLS, *co-simulation* is often used to verify the correctness of designs; this involves executing the C specification and translated RTL in tandem and verifying that they produce the same output [43]. We further illustrate the limits of this approach in Section 8.5, where we display subtle bugs that can elude co-simulation.

Test generation techniques used in software testing can also be used in co-simulation [43]. However, all testing-based approaches lack the ability to guarantee the correctness of the design under test — even advanced test generation techniques can miss corner cases.

Symbolic Execution. To alleviate the issue of producing valid inputs that exercise the full code structure, another approach is to use a *symbolic* execution engine to interpret a program. It can explore all possible control-flow paths, irrespective of any input provided. Once a path to some error state is found, possibly by solving constraints on possible paths using a Satisfiability Modulo Theories (SMT) solver [14], the execution engine can also find inputs that satisfy path constraints and generate a new test [11]. KLEE [2, 16] is a popular symbolic execution engine that uses heuristics to explore paths, potentially exposing bugs. Instead of using heuristics, tools such as CUTE [72] and DART [36] use a concrete execution path from a single test to guide path exploration. In either case, since the number of control flow paths in a program can be intractably large or even infinite, these methods may fail to conclude.

UC-KLEE [68] is an extension of KLEE [2, 16]. It also uses a form of interpretation, to build a set of feasible execution paths for a program, discovering invariants and proving equivalence of complex programs, including pointer-based data structures. Complex techniques have been built to discover equivalence between programs including to verify processors, e.g., [10, 47]. Preliminary extensions for floating point support have also been developed [55], however they do not scale to the problem sizes nor program complexity we target.

For example, using UC-KLEE for proving a simple 3-line matrix-multiply kernel (`mm`) equivalent to itself takes for N=16 (matrices of 16x16) 0.6s, 7s for N=32, 1min20s for N=64, 3m15s for N=80 and fails for N=96 with 0 completed path. We then inserted a trivial bug in one of the `mm` functions: write 0 to the last cell of the output matrix at the end of the transformed function. For buggy cases, if we let it attempt to explore all paths (default mode), despite a single path being found eventually, it does not finish after 60 minutes when we for any problem sizes > 4, taking already 22s for N=4. Stopping at the first error leads to time more comparable to the equivalent case eventually. When using symbolic floats, which requires testing with KLEE-Float [55], it takes about 5s for N=2 to prove equivalence, 4min for N=4, and was killed

without an answer after 60min for all larger cases. Note that making the problem size also symbolic with KLEE does not work (timeout 60min) on our tests as soon as the test involves multiple dependent calls to gemm, or the data type is float, whichever version of KLEE is used.

In contrast, as shown in Chapter 8, the system we build handles both int and float (and double, etc.) symbolic variables, with and without bugs, N=4 in 0.02s, N=32 in 1s, N=64 in 8s, N=128 in 70s, etc. To illustrate further the scaling of our approach, we note the complete verification of an optimized BERT accelerator written for HLS with 12 attention heads, an input feature dimension of 768, and a hidden dimension of 3072 in the feed-forward network takes less than 30 minutes as shown in Section 8.2.

Verifying Parallelizing Transformations. The detection of concurrency bugs and equivalence between two implementations are often split into two different problems. Detecting bugs in OpenMP programs has been effectively implemented [38], and various static analyses have also been proposed, e.g. [19, 92]. However none verifies at the same time the compliance of the parallelized program with the semantics of the original unoptimized program.

The problem of determining the absence of races or deadlocks in parallel programs has been studied from multiple angles ranging from static analyses e.g. [15, 19, 78, 92], dynamic analyses [8, 20] including intercepting the OpenMP runtime [38], symbolic analyses [65, 71, 74], as well as using Coq-formalized proofs [23].

Program Equivalence. Another way to verify the correctness of programs is by *proving* two programs necessarily have the same semantics. Program equivalence is not decidable in general [27], but is decidable for important classes of programs. For example, PolyCheck is a system to prove equivalence of an affine program and its transformed variant via dynamic analysis [13] and supports "arbitrary" iteration reordering transformations. It is however fundamentally limited by the need to find a mapping between statements in both programs, preventing it from supporting statement transformations, as well as data/ storage transformations.

ISA is a tool to prove equivalence between a pair of affine programs [1, 83]. It supports parametric loop bounds and proves equivalence between a pair of restricted affine programs. However, it remains highly limited in transformation coverage [13], preventing its deployment for complex HLS optimizations.

In general, numerous approaches to prove the equivalence of expressions, in restricted contexts, have been developed such as [45, 73]. However, they are typically limited in applicability; that is, the space of program transformations supported. Equality saturation [86, 87] has also been deployed to prove

equivalence. It has reached a level of performance allowing impactful transformations to be modeled [80, 86, 87, 93]; however, these techniques are far from covering the rewrites necessary to model the class of transformations we target. In addition, the computational complexity of such approaches prevents manipulating complex programs with a rich transformation coverage.

Finally, deep learning methods have also been proposed to handle equivalence under a rewrite rule system, but the approach only handles programs of a few hundred nodes, without loops [48].

Translation Validation. Other approaches to check the correctness of a program transformation include translation validation [59]. Alive2 provides bounded translation for LLVM [56], and excels at catching a wide class of intricate bugs related e.g. to incorrect instructions emitted. However, it cannot catch all bugs: simply removing the exit condition of a loop without altering the rest of the program can fail to be captured as a bug. Other approaches for translation validation tend to restrict the scope of programs and optimizations supported, yet deliver significant bug detection capability, including for MLIR, e.g. [12, 85]. TV has also been deployed for specialized languages like Halide [22], and for HLS [21, 44].

Certified Compilers. Another approach to guarantee the correctness of transformations is to prove that the compiler itself is always correct. The first verified C compiler is CompCert [54]. It was extended for some forms of concurrent C programs [39, 95]. Vericert [39] is the first formally verified HLS framework based on CompCert. These methods are limited both in the types of transformations they support and the fact that only transformations performed by the compiler can be verified.

2.3.2 Specialization for Improved Scalability

The downsides of existing verification methods, as summarized above, prevent them from verifying the correctness of the complex HLS-based hardware designs we target in this work. Instead, to achieve relevant coverage for practical high-performance designs, we aim to be mostly agnostic to how statements and storage are implemented (unlike PolyCheck and ISA) while also verifying realistically-sized accelerators (unlike symbolic execution methods). We accomplish this by restricting the class of programs we support to programs which have Statically-Interpretable Control Flow (SICF), explored further in Section 4.5. We observe, for example, that all affine programs with constant loop bounds (e.g. N=128) fit in the SICF class. This includes dense linear algebra computations, as typically occurring in inference of large language models, an arguably critical class to support nowadays. There is a need for fast and

accurate *verification* of correctness of the optimized programs produced for inference of large models, which is exacerbated by the plethora of techniques and research tools developed to generate high-performance inference implementations. Based on these observations, we conclude the merits in exploring a specialized approach that would complement techniques such as UC-KLEE: a technique restricted to a class of programs but offering order(s) of magnitude acceleration for the verification process. Intuitively, we target programs which have a single execution path, alleviating the need for any SMT-based approach or mitigation strategies for paths explosion. We define our coverage more formally in Section 4.5.

SICF programs have a single path through their control flow, so the verifier does not need to maintain many different states nor their complex path constraints as in symbolic execution techniques, while still proving equivalence for all values of variables not involved in control flow. In particular, KLEE implements a different symbolic interpretation approach; ours is specialized for equivalence of programs with a single concretely interpretable CFG path and concretely interpretable array subscripts, in order to trade-off generality for speed. We limit coverage to fixed problem sizes (which are highly relevant in HLS-based designs), but can operate at order(s) of magnitude faster speed than KLEE due to our linear complexity for CDAGs construction and equivalence checking. This significantly widens the class of programs supported for equivalence checking in feasible time.

We aim to develop a framework that is (a) mostly independent from how the program is implemented (schedule, storage, syntax); (b) scalable to realistic problem sizes (scales linearly with the number of operations executed in the program); (c) independent from how the program is produced (compiler, research tool, human-written). This enables the deployment of highly optimized programs produced by experimental tools in a safer fashion, proving their correctness at the source level.

Chapter 3
Verification by Hybrid Concrete-Symbolic Interpretation

We now outline our approach to proving the equivalence between two programs, which is designed with the following considerations.

We target optimized loop-based functions, with the objective of being mostly independent from the *syntax* used to implement these functions, and reasoning instead on the *semantics* of the program computations. The class of program we support, which includes for example affine programs [53], encompasses a broad range of applications such as linear algebra, image processing, data mining, machine learning, physics simulation, and more [64], as well as modern deep learning inference computations [49, 61]. Our coverage extends far beyond programs that are *syntactically analyzable* as polyhedral programs: in Chapter 4, we define for the first time a novel class called *Statically Interpretable Control-Flow (SICF) programs*, which we handle in time and space linear with respect to the number of operations executed in the programs.

As our approach is mostly agnostic to the syntax used, that is, how the program has been written, we cover a wide variety of program transformations that are typically implemented for high-performance HLS designs — arbitrary loop transformations, arbitrary statement transformations, and arbitrary storage and data transfer approaches: for example, scalarization, local and multi-buffering, and data transfer using FIFOs. To the best of our knowledge, this is the richest set of code transformations supported in a single automated program equivalence tool.

Finally we target a *practical* system capable of formally proving equivalence, subject to a meaningful set of restrictions, while maintaining high throughput for proof computation — our verification system offers roughly the performance of code simulation, processing at approximately 0.5 million statements per second, utilizing just a single CPU core.

We immediately set a number of restrictions on the class of programs we support for our system to be able to prove their equivalence. First, we do not prove equivalence between arbitrary pairs of programs: we specifically reason only on a pair of programs P_A, P_B *such that* P_B *is meant to replace* P_A *in a larger program.* That is, these two programs are necessarily called with the exact same *environment* [27], for every possible

execution. Second, as we require the control-flow to be statically *interpretable*, a looser condition than for static analyses where the control-flow must be statically *analyzable* (e.g., using polyhedral structures [31–34]), we typically require the problem sizes to be known at compile-time. We do not support parametric loop nest analysis, which is a requirement that also arises when performing co-simulation or testing, and which can be partially alleviated by proving equivalence once for each element in a set of problem sizes.

3.1 Overview of the Approach

With these objectives and restrictions in mind, we now introduce the key principles of our approach, each developed later in this manuscript. We illustrate the concepts to prove equivalent the pair of simple programs shown in Listing 4. For clarity we use explicitly the AMD Vitis HLS semantics for FIFOs and dataflow region declaration, but our approach is not specific to any HLS toolchain in particular.

As shown later, we cover a complex range of code and data transformations; however, this example already illustrates changing I/O, storage, statements, and loops in the program. We are not aware of any tool that can currently prove the equivalence between these programs within a single framework. For example, both ISA [83], based on static analysis, and PolyCheck, based on a more general dynamic analysis [13] would fail to prove equivalence: statements cannot be matched between the two programs as they differ in storage. To prove that P_A is equivalent to P_B for the calling context considered (which provides the problem sizes here), our approach operates as follows:

- We aim to build a *symbolic canonical representation* of the computation that is performed to produce the value of *each memory cell that is live-in/live-out* for the program. In the case of well-defined functions without side effects, the class we support, this set of cells is captured in the function arguments. This representation shall be independent of which statement(s) were used to produce the computation, as well as any temporary storage implemented. This is presented in Chapter 5.
- We prove equivalence by computing whether this canonical representation is *identical* between both programs, for *every live-in/live-out memory cell*. This is presented in Chapter 6.
- To be robust to "any" implementation of the program and its control-flow, we rely on a partial *concrete interpretation for the program*, which will concretely evaluate control-flow expressions and simplify them when possible. When an expression cannot be concretely evaluated, it is automatically promoted to symbolic representation, during interpretation. If the interpreter reaches the end of the program

control flow, then and only then we can prove the programs equivalent if their per-cell computation representations are fully identical. This is presented in Chapter 4.

```
// Program PA, the original program:
void matvec(float* restrict A, float* restrict x, float* restrict y, int N) {
  for (int i = 0; i < N; ++i) {
    y[i] = 0;
    for (int j = 0; j < N; ++j)
      y[i] += A[i * N + j] * x[j]; } }

// Program PB, a replacement for program PA:
typedef hls::stream<float> fifo_t;
void data_in(fifo_t& fifo_A, fifo_t& fifo_x,
             float* A, float* x, int N) {
  for (int i = 0; i < N; ++i)
    fifo_x.write(x[i]);
  for (int ij = 0; ij < N * N; ++ij)
    fifo_A.write(A[ij]); }
void matvec_core(fifo_t& fifo_A, fifo_t& fifo_x, float* x_buff, float* y, int N)
{
  for (int i = 0; i < N; ++i) x_buff[i] = fifo_x.read();
  for (int i = 0; i < N; ++i) {
    float y_temp = 0; int j;
    for (j = 0; j + 2 < N; j += 2) {
      y_temp += fifo_A.read() * x_buff[j];
      y_temp += fifo_A.read() * x_buff[j+1];  }
    for (; j < N; j++)
      y_temp += fifo_A.read() * x_buff[j];
    y[i] = y_temp; } }

void matvec(float* restrict A, float* restrict x, float* restrict y, int N) {
#pragma HLS dataflow // Ignored for sequential verif.
   fifo_t fifo_A, fifo_x;
   float x_buff[100];
   data_in(fifo_A, fifo_x, A, x, N);
   matvec_core(fifo_A, fifo_x, x_buff, y, N);  }

// Caller:
int main() {
   float *x, *y, *A; // data not needed for verification
   matvec(A, x, y, 100); // calling context
}

```

Listing 4: Illustrating example: dense matrix-vector product.

- We prove equivalence when using FIFOs of a given depth, non-blocking and blocking, both sequentially and with coarse-grain dataflow-style concurrent execution of functions that write/read the FIFOs, as in programs generated by AutoSA [84]. This is presented in Chapter 7.

3.2 CDAG Representation

We build a representation of the computation producing a value stored in memory cell (e.g., `y[0]`) in the form of a graph, specifically a computation directed acyclic graph (CDAG) [30, 60]. We formally define CDAGs in Section 5.1. Figure 2 shows an excerpt of CDAGs built for program P_B. Every variable in the program which can be *concretely* evaluated is, such as `(i * N) + j`, are *replaced by their result* during interpretation, giving values $0, 1, \ldots$ which are used to identify the memory cells being addressed. When an expression cannot be computed, for example because it uses a live-in, unknown value such as `A[0]` or `x[0]`, the expression is automatically promoted to a symbolic representation: its CDAG. At every assignment the current CDAG is stored, so that it can be used as replacement for the next use of the variable. The process is repeated for every iteration of `j`, and one CDAG per `y[i]` is eventually created.

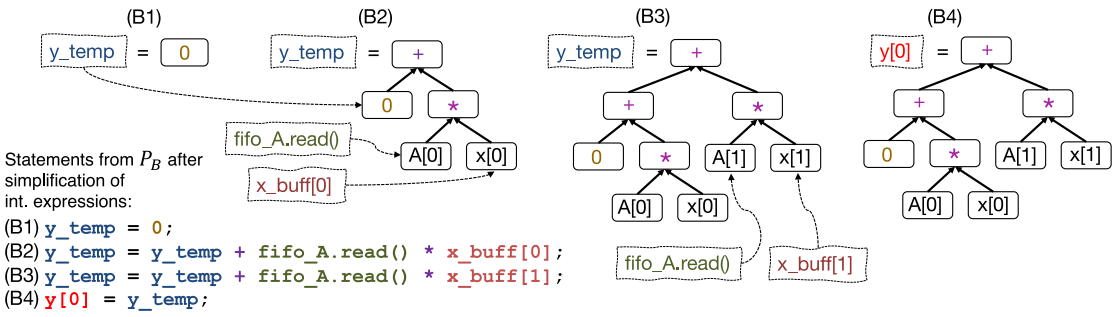

Figure 2: When interpreting a statement, first, every variable referenced is replaced by its content from the interpreter memory, if any. Integer sub-expressions are then simplified with concrete evaluation, and the result is stored in memory: B1 assigns 0 to `y_temp`. Then the first `j` loop is interpreted, storing 0 for `j`, testing `2 < 2`, transferring control to the next `j` loop. It tests `0 < 2` and moves to interpreting its body. For B2, in the RHS, `y_temp` is replaced by its current value, 0, `fifo_A.read()` is replaced by its first written element, the symbolic (live-in) value `A[0]`, as well as `x[0]` for `x_buff[0]`. As these values are symbolic, the entire expression is symbolic, stored as a CDAG for `y_temp`. When B3 is interpreted, we replace `y_temp` with its CDAG from memory, creating a new CDAG, now stored for `y_temp`. Once loop `j` terminates, B4 assigns the CDAG of `y_temp` to the live-out `y[0]`.

When interpreting P_B *sequentially*, we emulate the FIFO API by implementing `fifo.read()` and `fifo.write()` via a simple array `fifo[]` and its start/end positions in C, which is processed by the interpreter. We discuss in Chapter 7 the details of verifying FIFOs, sequentially or in dataflow-style mode.

Note this construction process is agnostic to how storage and computations are implemented: creating scalars and different loop nests simply leads here to building the same CDAG that is eventually stored in `y[i]` for both P_A and P_B.

If and only if the control-flow interpreter has reached the end of the program control, we have built CDAGs for every live-in/live-out memory cell touched by the program. We can then compute their equivalence by checking, cell by cell, whether the CDAGs are isomorphic. If so, we have proven the programs are equivalent. We can catch errors in the loop nests, handling of FIFOs (e.g. incomplete data, deadlocks, etc.), in how statements are scheduled with relation to dependencies, etc. Here we prove P_A and P_B equivalent for $N = 100$ (and any `A`, `x`) in 0.4s.

Usually, transformations should not alter the number and relative order of dependent operations for the isomorphism check to be successful. However as discussed in Chapter 6, our post-normalization of the CDAG enables the support of transformations exploiting associativity/commutativity of operations, as well as some changes in the set of operations computed, if a set of semantics-preserving rewrite rules is provided.

Chapter 4
AST-based Hybrid Interpreter

We now present our concrete interpreter to automatically build CDAGs for programs. We have implemented support for a large subset of the C/C++ language, within the PAST [3, 63] library. PAST is a generic, language-independent Abstract Syntax Tree library, equipped with a parser from C to PAST, built using flex/bison ANSI C grammars by Lee and Degener [4].

4.1 Architecture of the Interpreter

The interpreter operates on a PAST tree representing an input, compilable program. Contrary to a full C interpreter, it does not require a complete program to be provided: it supports any code region, and functions with their definitions provided. *Any value which is not computable during interpretation will be considered symbolic*, allowing the interpreter to proceed even without the concrete data the program operates on. The interpreter comprises the following components:

- A parser from a partial C/C++ code fragment to the PAST AST. During parsing, no analysis is done: it is a simple encoding of the program as an AST.
- An AST traversal mechanism that implements all control-flow operations of the program, including function calls, variable declarations, etc. This is presented in Section 4.2.
- A memory storage system, to store concrete and symbolic values for every variable accessed during interpretation, including temporary/local variables. We implement a dynamically allocated sparse tensor approach to this end, which is presented in Section 4.3.
- A concrete expression evaluator, used to evaluate and simplify expressions typically used for control-flow and array subscript expressions. This evaluator must implement the same exact *concrete* semantics as the target hardware for which programs are proved equivalent: identical overflow behavior, support of different bitwidths, etc. This is presented in Section 4.4.
- A CDAG building system, which manipulates symbolic expressions building them by partial evaluation, presented in Chapter 5.

4.2 AST Traversal for Interpretation

Our interpreter traverses the program AST following C/C++ execution conventions. It terminates when there are no more instructions to interpret; that is, it has reached the end of the control-flow of the region analyzed. We support most classical C constructs: `for`, `while`, `do-while`, `if`, `switch`, `return`, and `break`, as well as function definitions and function calls. When a function is called, its definition must be available in the region analyzed, and control is simply transferred to this function. We support pass-by-value and pass-by-reference for function calls. We require that all functions' arguments are `restrict` to ensure no aliasing, as we do not perform any aliasing analysis and assume different named arguments point to different memory regions.

We currently support variable declarations, and a very limited form of pointer arithmetic and type casting. However, this is only a limitation of our current implementation, since supporting those in full does not pose any particular technical challenges.

4.3 Interpreter Memory

The interpreter maintains a memory state for the program. Every variable (or memory cell for arrays) accessed during the program has associated storage, where we store (a) whether the variable is (currently) symbolic or concrete; (b) the concrete value or the symbolic CDAG representing the expression to compute that variable; (c) whether the variable is live-in or not (that is, it is being read before being written); and (d) statistics useful for subsequent program optimizations, such as the number of reads/writes to the variable. To control memory storage size, especially in the case of manipulating arrays, we implement a dynamically allocated sparse tensor for the memory. As we support C99 multidimensional arrays, whose size is not necessarily known at compile-time, when an array is sparsely accessed (e.g., the only element touched during execution is `A[42][51]`), storing its dense representation 0-42 × 0-51 would consume unnecessary memory.

4.4 Concrete Expression Evaluation

Equipped with a traversal mechanism and memory for the interpreter, we can now evaluate a program. *A fundamental aspect of our system to be able to prove equivalence is the availability of a concrete expression evaluator that implements exactly the same concrete semantics as the target hardware.* Indeed,

in a nutshell, our interpreter will replace expressions like `j = 0; j++; print(j);` with `print(1);`. This simplification is what makes our system robust to any implementation of the control-flow, and is fundamental for its execution time, but it requires a verified implementation of the concrete evaluator. This expression evaluator shall carefully implement the exact same behavior as the target architecture, for example as described in documentation of the HLS framework being used. Issues of overflow, handling of various bitwidths, type promotion, etc. must be exactly implemented as they would behave on the target hardware for our system to conclude a proof of equivalence. Otherwise, different behavior may be observed when running on the concrete target hardware.

In this work, we implemented the concrete semantics of the C language for integer expressions, supporting all available unary and binary operators, including bitfield manipulation. Our PAST-based integer expression evaluator is implemented using about 300 lines of C, making its manual verification accessible. We are, however, dependent on the compiler and test machine to correctly execute this program; using certified compilers such as CompCert [54] may be desired for increased confidence.

4.5 Overview Algorithm for SICF Interpretation

Algorithm 1 below outlines our approach to hybrid concrete/symbolic interpretation. Our implementation behaves identically, but is optimized for speed and minimizes the work done for each statement by reusing prior computations whenever possible and caching per-statement analysis.

We remark on the complementary nature of approaches such as KLEE [68], which computes a set of possible execution paths to support "symbolic" control-flow by interpreting LLVM bytecode, but is mostly limited to integer symbolic variables; or Alive2 [56] which can expose fine bugs in LLVM programs. We target coverage of the "converse" case: supporting equivalence of symbolic expressions, especially for floating-point operations or any well-defined type for symbolic variables, but requiring that the control-flow can be completely computed by concrete interpretation at compile-time. Combining both approaches is the subject of future work. This leads to the following definition of the class of program we support:

Definition 1 (*Statically-Interpretable Control-Flow program*): Given a compilable (set of) function(s), possibly without some of the input data it operates on: this program is SICF if there is enough input data provided such that all branches can be exactly evaluated and taken at compile-time by concrete

interpretation (control-flow can be computed), and every distinct memory cell accessed can be uniquely mapped to a finite-size array (dataflow can be computed, and the program terminates)

Algorithm 1: `interpret_program`

Input: Program AST P, using C semantics

Output: Set of CDAGs for each memory cell after interpreting P

```
process_statement(S, stmt):
    stmt′ = evaluate_and_simplify_concrete_expressions(stmt)
    if stmt′ contains a symbolic expression used in branch or array subscript
        error
    if stmt′ is an assignment to write location w
        if s′ contains symbolic expressions
            let CDAG = build_CDAG_from_rhs(stmt′)
            S = store_symbolic_cdag(S, w, CDAG)
        else
            S = store_concrete_value(S, w, CDAG)
    return S
interpret_program(P):
    let S = ∅ // program state
    let stmt = get_first_statement_in_CFG(P)
    while stmt do
        S = process_statement(S, stmt)
        stmt = get_next_statement_in_CFG(A)
```

Chapter 5
Building CDAGs by Symbolic Analysis

We now present our approach to CDAG construction, by means of program interpretation. CDAGs are constructed for every expression result assigned to any variable in the program which contains one or more symbolic elements, including local variables. However, as detailed in Chapter 6, we limit the set of variables considered for equivalence checking to the live-in/live-out values of the program. This restriction serves as a sufficient condition for our purposes.

5.1 Formal Definition

A CDAG can capture an expression that computes a single value [30].

Definition 2 (*CDAG*): A CDAG is a directed acyclic graph such that every leaf node vl_i is a value (symbolic or concrete) and every other vertex vc_i represents an n-ary computation producing a single value, a function of the children of vc_i. An edge $v_i \rightarrow vc_j$ exists iff the value produced by any vertex v_i is used in the computation vc_j.

CDAGs are a well-known representation of computations, which have been used e.g. in proving I/O lower bounds for programs [30]. Intuitively, they can be built from a set of ordered instructions, where the operands are named. That is, it is possible to first build an execution trace for the program, and manipulate it to rebuild the same CDAGs as we implement in this work. However, by finely integrating their construction with program interpretation, we can easily track operands and their value, and reason distinctively between concrete and symbolic values. CDAGs have a fundamental property: they are agnostic to storage, and only represent a computation, not how it is implemented, as illustrated in Figure 2.

5.2 Procedure for CDAG Construction

We aim to build a CDAG for a variable incrementally, by interpreting expressions in the order they would be executed by the program. We first clone the AST of the original expression in the program, e.g. `(y[i] + A[i*N+j] * x[j])`, and progressively rewrite it.

- We traverse the AST of the expression in postfix, and, for every operation which has only concrete value(s) as operands, we invoke the concrete expression evaluator and replace the associated subtree

with this concrete value. When a concrete variable is referenced in an expression, it is first replaced by its current concrete value from the interpreter memory.

- We then re-traverse this modified AST in postfix, now with integer computations simplified, e.g. the tree is now `(y[0] + A[0] * x[0])`. For every variable left referenced in the AST, we replace it with its known current CDAG, if any. Otherwise, we create a symbolic node modeling this value, e.g. `A[0]` and `x[0]`.

During this process, *every symbolic variable being referenced has been replaced with the expression which computes its value*. By design, the resulting tree can only refer to symbolic values (typically live-in data) and numerical constants. More specifically, we have:

Theorem 1 (*CDAGs after termination*): Given a function f with non-aliasing input arguments, without any side effects and using only local variables which are dead after the function exits. If the interpreter succeeds in reaching the end of the function control-flow, then necessarily the CDAGs computed for its arguments will contain as leaves only constants and symbols of the arguments themselves, or be compositions of the CDAGs already computed for the arguments prior to executing the function.

The proof relies on the function being side-effect free, as all local variables have a liveness limited to the function scope and therefore cannot generate new symbols used in CDAGs after the function exit. CDAGs can only contain symbols reachable via the function arguments prior to its execution.

5.3 Complexity Considerations

CDAGs, built as described above by repetitive replacement of variables referenced by their known CDAG, so far have the fundamental property of being agnostic to how the computation is implemented, including if using local storage. However, this comes at an important complexity cost: for a program that executes N *operations*, its CDAG will have at least $O(N)$ nodes. Taking for example matrix multiplication, it has $O(N^3)$ FMA operations, so the CDAG will therefore have at least $O(N^3)$ nodes to represent these operations.

There exists also a degenerate case that can make CDAGs grow exponentially, for example when a variable is used at multiple places in the same expression, itself being in a reduction-style computation: `s += s + s + s` being repeated under a loop, leads to a final CDAG of $O(3^N)$ size in theory.

In our implementation we address this problem by ensuring the space complexity remains roughly $O(N)$ even in this case, using pointers and caching (shallow copies) to represent identical subtrees [81].

Finally, the complexity of our implementation for CDAG construction, both in time and space, is typically $O(N)$ for a program executing N operations for sequential verification. This aspect is fundamental to the performance of verification, as shown in Chapter 8.

Chapter 6
Equivalence Checking

We now describe equivalence checking for sequential programs, and report possible bugs to the user and their location otherwise. Concurrency checking is presented in later Chapter 7. Intuitively the process amounts to checking full isomorphism of the CDAGs produced by P_A and P_B for the same memory cell, for all memory locations that are live-in and live-out for the programs — that is, memory locations reachable outside the function(s) checked for equivalence. We also support a variety of rewrite-rule based normalizations to increase equivalence coverage, including support of transformations altering the order and count of operations.

6.1 Theoretical Foundation

By construction, our system can prove the equivalence of a pair of programs, under the hypothesis that all global arrays and function arguments do not alias. We require the ability to match data that is live-in (that is, read first before being written) and live-out (that is, alive after the program region terminates) between P_A and P_B: this is easily achieved by encapsulating the regions to analyze in a single entry function with well-defined arguments.

Theorem 2 (*Equivalence of Programs*): Suppose two programs are such that the interpreter terminates by reaching the end of their control-flow without error. They are semantically equivalent if, for every memory cell that is live-in/live-out to the programs, the CDAGs produced by each program for that cell are semantically equivalent.

The proof requires that P_B replaces P_A in a larger program, ensuring the same execution context necessarily for both. It requires that the integer expression evaluator implements exactly the concrete semantics of the target hardware, making code replacement via partial concrete evaluation in the program necessarily equivalent to the code before evaluation. As the interpreter can only terminate if exclusively concrete values are used in the control-flow and array subscripts, no other execution path than the one interpreted can exist for the given input programs. CDAGs are built by successive equivalent replacements; if the process terminates, they correspond exactly to the full computation to be performed on every memory cell. This can only be a function of live-in data, per Theorem 1, and is therefore independent of

any temporary data. If CDAGs are identical for the same memory cell for both programs, then they must produce the same output value for this cell; two programs are equivalent if this is true for every live-in/out memory cell.

However, our framework does not prove non-equivalence in general. For example the absence of a rewrite rule in the system to show that `pow(x,2)` is equivalent to `(x * x)` would prevent proving these two expressions equivalent. While our proof of equivalence holds for any superset of the semantics considered, exposing differences in CDAGs after normalization only indicates the programs may not be equivalent under a subset of the semantics.

Corollary 3 (*Non-Equivalence of Programs*): Given two programs such that the interpreter either fails to terminate, or such that their CDAGs are not shown to be semantically equivalent, then these two programs may be semantically equivalent.

To compute the isomorphism of CDAGs, we simply perform, for each CDAG, a merged prefix and postfix collection of its nodes to form a vector of size $2n$ for a CDAG of n nodes, and check the strict structural equality of the two vectors obtained.

6.2 CDAG Normalization

Our system can natively detect equivalence when the count and order of operations performed to produce a CDAG is identical for both P_A and P_B. For example, `(A[i*N+j] * x[j])` is not equivalent to `(x[j] * A[i*N+j])`: the CDAGs are not identical. To handle a wider class of equivalences, we augment the system with the capability to normalize CDAGs, which is applied if checking their isomorphism originally fails.

Specifically, the *user* can declare a set of semantics-preserving rewrite rules to be applied to the CDAGs. For example, to support detecting equivalence between the CDAGs `(a + b + 0)` and `(a + b)`, the user may specify a rewrite rule $+(0, a) \rightarrow a$. Our normalization system simply applies these rules greedily to modify the CDAGs until no more change can be achieved. This ensures completeness if and only if the user-provided rewrite rule set is *terminating*, where every possible sequence of rule applications is finite, and *confluent*, where every sequence of rule applications eventually computes the same result [9].

In contrast, techniques like equality saturation [80, 87] may be able to saturate the representation and be complete even for non-terminating and non-confluent sets of rewrite rules, but at a very high

computational cost — especially when associativity and commutativity are needed — that is impractical for trees of thousands or millions of nodes as we manipulate.

Associativity and commutativity of operations are handled separately: if the user specifies that an operation like addition or multiplication is associative and commutative (AC), we simply flatten AC operators (i.e. convert binary operators into n-ary operators: $+(a, +(b, c))$ is rewritten into $+(a, b, c)$) and lexicographically sort their children [29]. This leads to a canonical CDAG representation under associativity and commutativity. This representation can also be rewritten *modulo* AC using user-specified rewrite rules, but only under restrictions on rules involving AC operators as in [29].

Chapter 7
Verification of Concurrent Programs

While our approach to sequential verification can ensure that two sequential programs are correct, concurrency is important to support as well, since many hardware accelerators exploit concurrency to maximize performance. A common strategy for exposing concurrency using HLS is to partition the program into processing elements that execute independently, and that use blocking FIFOs for communication and synchronization. In the Vitis programs our implementation supports, this corresponds to `#pragma HLS dataflow` regions which contain function calls that execute concurrently and have a producer/consumer relationship using FIFOs. The transformations used to implement this concurrency can introduce bugs like data races and deadlock. We extend the sequential interpreter to support concurrency and reason about programs' concurrent behavior to catch these bugs.

7.1 Interpreting Concurrent Programs

To verify the correctness of concurrent programs using our system, we first need to be able to interpret them. We equip our sequential interpreter with two important features. First, support for concurrent tasks, which can be scheduled for interpretation akin to a multiprogramming approach in operating systems, switching between concurrent regions ready to execute. Second, the ability to interrupt and resume a particular task, at any point. This is necessary, for example, if the task the interpreter is currently interpreting tries to read an element from an empty FIFO.

To represent concurrency in the C programs we interpret, we augment the interpreter with a concurrent API. This API enables spawning concurrent tasks, and uses operations on semaphores to synchronize these tasks. The functions in the concurrent API are listed below:

- `spawn(block)`: spawns a new concurrent task that executes the instructions in `block`.
- `semaphore_init(semaphore_id, val)`: initializes a semaphore with id `semaphore_id` to the value `val`.
- `semaphore_set(semaphore_id, val)`: sets semaphore with id `semaphore_id` to value `val`.
- `semaphore_wait(semaphore_id, val)`: blocking wait on semaphore with id `semaphore_id`; if the semaphore does not equal `val`, the current task is interrupted and cannot resume interpretation until `semaphore_set` is used to set the semaphore to `val`.

7.2 Verifying Programs That Use FIFOs

We now present our approach to verifying the insertion of FIFOs to communicate data between functions, and the associated program restructuring, is correct. We distinguish two cases: a *sequential* verification approach, where we assume FIFOs are of infinite depth; and a *concurrent, dataflow-style* verification approach, where we assume FIFOs are of a finite depth, and functions manipulating FIFOs appear in a dataflow-style region such that they execute concurrently, being activated until waiting for data based on FIFO readiness and use.

7.2.1 Modeling FIFOs Using Semaphores

To translate a program that uses blocking FIFOs to a program that uses the interpreter's semaphore API, we use a translation that amounts to the `#defines` in Listing 5. Our translation adds two global arrays: `fifo_semaphores`, an array of semaphores initialized to `READY_TO_WRITE`, and `fifo_buffers`, which holds the contents of each FIFO. We model *blocking* FIFOs that have a maximum depth of 1 by guarding reads and writes to the FIFO buffer using blocking waits on the unique semaphore associated with the FIFO.

```
1  #define fifo_write(fifoid, val):
2    semaphore_wait(fifo_semaphores[fifoid], READY_TO_WRITE)
3    fifo_buffers[fifoid] = val
4    semaphore_set(fifo_semaphores[fifoid], READY_TO_READ)
5
6  #define fifo_read(fifo, val):
7    semaphore_wait(fifo_semaphores[fifoid], READY_TO_READ)
8    val = fifo_buffers[fifoid]
9    semaphore_set(fifo_semaphores[fifoid], READY_TO_WRITE)
```

Listing 5: Implementation of a blocking FIFO of maximum depth 1 using semaphores

7.2.2 Sequential Verification of FIFO-based Programs

To verify the correctness of program restructuring for concurrency without incurring the overhead of concurrent interpretation, we can sequentially interpret a program that uses FIFOs. We rely on the assumption that functions producing data appear prior to functions consuming that data in sequential execution order, a realistic assumption e.g. in the AMD Vitis toolchain for `hls::stream` FIFOs. We assume here FIFOs with infinite size, hence non-blocking writes.

We substitute the API calls in the program for reading/writing FIFOs with our own emulating read/write implementation, replacing them using (a) a self-growing array of same type as the FIFO, and a start/end position pointer; and (b) for every write we write the element to this array at the first available position `end`, then increasing it — for reads, we do the converse with `start`. This simple approach is not able to catch concurrency bugs such as deadlocks or data races, in contrast to our concurrent dataflow-style verification below. However, it still allows our system to catch bugs in program restructuring and how the FIFO is used, while maintaining our target complexity of $O(N)$ time and space for N statements interpreted in the program.

7.3 Concurrent Verification Approach

As in the sequential verification approach, we aim to interpret two programs and compare the CDAGs produced for each output memory location. However, concurrent programs have the issue of operation scheduling: for any concurrent program, there can be multiple schedules for operations that can execute in parallel that may even compute different results. Rather than interpreting every possible interleaving of statements in the program which can be intractable, we want to reason about all schedules while interpreting only *one*. Our system also needs to be agnostic to the specific implementation of operations during HLS. Rather than reasoning about the latency of operations, which can vary based on the HLS approach used, we assume that every pair of statements can happen in parallel unless they are ordered by explicit point-to-point synchronization using blocking FIFOs.

Maintaining, either explicitly or implicitly, a model of which operations and variable accesses can occur at the same time is necessary to correctly interpret these concurrent programs. To model which variable accesses can happen concurrently, we use the well-studied happens-before relation [52]. We define this relation as a partial order on dynamic statement *instances* executed by the interpreter (i.e. a statement in the body of a loop that executes N times will have N instances rather than 1), as below:

Definition 3 (*Happens-before relation*): For statement instances $stmt_1$ and $stmt_2$ that are interpreted in a program, $stmt_1$ happens before $stmt_2$ — denoted $stmt_1 \succ stmt_2$ — if and only if $stmt_1$ must complete before $stmt_2$ starts for every possible valid schedule of statements in the program.

We use the following criteria to dynamically build the happens-before relation during interpretation:

- Within a task, statements execute sequentially: for statement instances $stmt_1$ and $stmt_2$ that belong to the same task, if $stmt_1$ is interpreted before $stmt_2$, then $stmt_1 \succ stmt_2$.
- The spawn of a task must happen before the task begins: for a statement instance $stmt_s$ that spawns a task t, and $stmt_t$, the first statement of t, $stmt_s \succ stmt_t$.
- A semaphore `wait` on a value v has to happen after the semaphore is `set` to v: for a statement instance $stmt_w$ that contains a `wait` on semaphore S and value v, and $stmt_s$, the most recent `set` of S to v if one exists (discussed in more detail in Section 7.3.1), $stmt_w \succ stmt_s$.
- The happens-before relation is transitive: if $stmt_a \succ stmt_b$ and $stmt_b \succ stmt_c$, then $stmt_a \succ stmt_c$.

To ensure that only interpreting a single schedule is sufficient to reason about equivalence over all schedules, the interpreter needs to check that the program is deterministic [42]. That is, whichever the actual valid concurrent schedule interpreted, all shared memory variables will necessarily hold the same value at program exit.

Therefore, to accomplish this in our model, it is sufficient to ensure the absence of data races; that is, the ordering of accesses to shared variables is the same for any valid schedule and no two tasks can write at the same time (without synchronization) to the same shared location. Without this possibility, the final value of shared variables cannot be different as a function of the specific schedule interpreted.

We define these possible parallel accesses as read/write conflicts below:

Definition 4 (*Read/write conflict*): Given two statement instances $stmt_1$ and $stmt_2$ that access the same shared memory location, if $stmt_1 \not\succ stmt_2$, $stmt_2 \not\succ stmt_1$, and one of the accesses is a write, then there exists a read/write conflict between $stmt_1$ and $stmt_2$.

To make sure concurrent programs we interpret do not contain read/write conflicts (and thus possibly nondeterminism), we keep track of which statement instances access which variable. If two statement instances that access the same variable are not ordered by the happens-before relation, the interpreter reports an error and aborts interpretation.

Semaphores are a special case: they are shared variables whose reads and writes are used to *build* the happens-before relation. Often, reads and writes (`waits` and `sets`) to the same semaphore do not have explicit synchronization between them, since these accesses are used to add synchronization to the program. We need extra machinery to reason about their behavior, as discussed in Section 7.3.1.

Another issue that can occur in concurrent programs is deadlock, which is detected using a simple approach: if the interpreter has one or more currently executing tasks (i.e., not completed by reaching

the end of their control-flow) that cannot make further progress because of a blocking operation, then we report a deadlock:

Definition 5 (*Deadlock*): Suppose that $f_1, ..., f_n$ is a set of concurrently executing tasks. If there exists f_i in a blocking state which depends on semaphore S and no other f_j can update S, either because the interpreter completed its execution or because it does not modify S, then the program deadlocks under a possible concurrent schedule.

7.3.1 Building Happens-Before with Semaphores

While interpreting a concurrent program, we only need to store one value for each non-semaphore variable. When the interpreter reaches a write to these variables, we have the guarantee that either 1) this write happens after all other writes and thus should set the value of the variable for subsequent accesses, or 2) this write happens concurrently with at least one other write, and the interpreter will catch a read/write conflict and abort. Interpretation will only compute one value for this variable regardless of the order in which statements in the program are interpreted.

We lack this guarantee for semaphores: semaphore `waits` and `sets` can occur in parallel, so if semaphores were handled like other variables, it would be possible for the interpreter to compute different values for a semaphore depending on the schedule of operations. This could also result in a different schedule and final result altogether, since semaphores are used to *create* the schedule for operations via the happens-before relation.

To know which value a semaphore has during interpretation, we first require that all `semaphore_set` statements that `set` the same semaphore are ordered by the happens-before relation, and that all arguments to `semaphore_set` are concrete. Given a `semaphore_wait` statement $stmt_w$ that waits on semaphore S to be value v, the *most recent* `semaphore_set` is defined as the `semaphore_set` statement $stmt_s$ that sets S to v such that $stmt_w \not\succ stmt_s$ and there does not exist another `semaphore_set` $stmt'_s$ where $stmt_s \succ stmt'_s$ and $stmt_w \not\succ stmt'_s$. When interpreting a `semaphore_wait` $stmt_w$, we find the most recent `semaphore_set` statement $stmt_s$ and add $stmt_s \succ stmt_w$ to the happens-before relation. If there does not exist a most recent `semaphore_set`, this is a blocking wait: the interpreter interrupts the task and switches to another until an appropriate `semaphore_set` is interpreted.

To avoid explicitly traversing the happens-before relation as a graph to retrieve the most recent `sempahore_set`, we can simply maintain an array, for each semaphore, that stores a tuple of each

`semaphore_set`'s statement instance and value. Since the `sets` for any given semaphore are totally ordered, the most recent `set` can be found by iterating over the array from the beginning until a statement $stmt_s$ such that $stmt_w \not\succ stmt_s$ is found. In our implementation, the function `restore_semaphore_values` finds and assigns the appropriate value to all semaphores that have been initialized: this function is used in Algorithm 3.

7.3.2 Concurrent Interpretation Algorithm

To perform checks for read/write conflicts and handle semaphore `waits` during interpretation, we need to represent the happens-before relation. However, this relation is transitive and defined over statement *instances*, so explicitly representing it as a graph would be inefficient in both time and space. We allow approximation for performance and represent, at worst, a subset of the happens-before relation. This approach may result in false read/write conflicts, but will never miss read/write conflicts: any two statements that are unordered in the happens-before relation will also be unordered in any subset of the relation.

We first need to add several features to the interpreter to keep track of variable accesses. Each statement instance is assigned an ID, which is an integer value that monotonically increases in the order statement instances are executed by the interpreter. In addition, two arrays are associated with each memory location in the interpreter: `read_data` and `write_data`. These arrays have an element for each concurrent task, and every time the variable is read or written, the associated array at the index for the current task is set to the current statement instance ID.

The main data structure used to check for read/write conflicts is the happens-before matrix H, a 2D array whose size in both dimensions is the maximum number of concurrent tasks in the program. $H[t_1][t_2]$ represents the greatest statement ID for statements executed by task t_2 such that subsequent statements executed by task t_1 happen *after* all statements with an ID less than this value. For statements $stmt_1$ and $stmt_2$ that belong to tasks t_1 and t_2 and have IDs ID_1 and ID_2 respectively, the edge $stmt_2 \succ stmt_1$ can be added by setting $H[t_1][t_2] = ID_2$. Read/write conflicts can then be checked using Algorithm 2.

Now that we have a way to check for read/write conflicts, we can interpret concurrent programs. Algorithm 3 shows an overview of our concurrent interpretation algorithm, which includes our concurrent scheduler, adding and checking happens-before information as discussed above, and catching deadlocks.

Algorithm 2: `conflict_exists`

Input: Happens-before matrix H, set of new variable accesses A'

Output: `true` if a possible data race exists between old and new accesses, `false` otherwise

```
1  conflict_exists(H, A')
2      for each access a' ∈ A':
3          let t' = index of task that performed a'
4          let v = variable accessed by a'
5          let C = {get_write_data(v)}
6          if a' is a write:
7              C = C ∪ {get_read_data(v)}
8          for each array d ∈ C:
9              for each task index t ∈ d:
10                 if t = t':
11                     continue
12                 if H[t'][t] = null_stmt_id or d[t] > H[t'][t]:
13                     return true
14     return false
```

The outer loop starting on line 7 represents the concurrent scheduler: while there is a task available to interpret, the scheduler picks one with `get_next_task`, executes a statement, and continues. In practice, the interpreter only needs to switch tasks if the current task cannot continue, so `get_next_task` will return the same task until it either finishes or blocks. Our scheduler is fair; that is, if a task is currently blocked on a `semaphore_wait`, the scheduler will move to other tasks to avoid an infinite loop. A task that is currently blocked will still need to be checked, however, and can be a result of the `get_next_task` call on line 8.

Lines 14–26 handle semaphore `wait`s and task spawns, both of which add happens-before information we need to store in H. On interpreting a `semaphore_wait` on a value, the interpreter either needs to interrupt the current task if the semaphore has not been set to the value or find the most recent `semaphore_set` to that value as defined in Section 7.3.1.

After handling task scheduling and semaphore operations, the statement is processed in the same way as for sequential interpretation: `process_statement` in Algorithm 3 is the same as defined in Algorithm 1. In our implementation, the happens-before edge on line 29 is implicitly captured by the interpreter's monotonically increasing statement instance IDs.

Finally, if the loop on line 7 has completed and there remains any tasks that have not completed — that is, reached the end of their control flow — then it is impossible for any task to make progress,

Algorithm 3: `interpret_concurrent_program`

Input: Program AST P

Output: Set of CDAGs for each memory cell after interpreting P

```
1  interpret_concurrent_program(P):
2      let S = ∅   // program state
3      let H = ∅   // happens-before relation
4      let A = ∅   // set of variable accesses
5      t = new_task_at_first_statement(P)
6      let T = {t}
7      while there exists a task ∈ T in ready state:
8          let t = get_next_task(T)
9          let stmt = get_next_statement(P, t)
10         let A' = get_variable_accesses(stmt)
11         if conflict_exists(H, A'):
12             error: data race
13         A = A ∪ A'
14         S = restore_semaphore_values(t)
15         if stmt = semaphore_wait(sem, val)
16             if semaphore sem = val:
17                 let stmt' = get_most_recent_set(sem)
18                 H = H ∪ (stmt' ≻ stmt)
19             else
20                 set_blocking_state(t)
21                 continue
22         else if stmt = spawn(block):
23             let t' = new_task_at_block(block)
24             T = T ∪ t'
25             let stmt' = peek_next_statement(P, t')
26             H = H ∪ (stmt ≻ stmt')
27         S = process_statement(S, stmt)
28         let stmt' = peek_next_statement(P, t)
29         H = H ∪ (stmt ≻ stmt')
30     if there exists a task not at the end of its control flow
31         error: deadlock
32     return S
```

and the interpreter reports a deadlock on line 32. If this is not the case, then all tasks have completed interpretation and the final result (set of CDAGs) computed by the program is stored in S, which can then be checked for equivalence with another program, either sequential or concurrent.

7.4 Example and Discussion

Our approach provides a conservative analysis of read-write conflict and deadlocks: if there exists a possible schedule under which these occur, we report so. We illustrate with a simple example in Listing 6.

```
1  // shared
2  float x;
3  fifo_t f;
4
5  void foo(int a) {
6    a += 1;
7    x += 1;
8    f.write(42);
9  }
10
11
12
13 void bar(int a) {
14   a *= 3;
15   f.read();
16   x *= 3;
17 }
18
19
```

```
1  // shared
2  float x;
3  fifo_t f;
4
5  void foo(int a) {
6    a += 1;
7    x += 1;
8    semaphore_wait(f, READY_WRITE);
9    buffer_f = 42;
10   semaphore_set(f, READY_READ);
11 }
12
13 void bar(int a) {
14   a *= 3;
15   semaphore_wait(f, READY_READ);
16   buffer_f;
17   semaphore_set(f, READY_WRITE);
18   x *= 3;
19 }
```

Listing 6: Concurrent interpretation example: `foo` and `bar` execute concurrently. The program on the left uses FIFO operations, and the program on the right is a translation of that program to our semaphore API.

Suppose we execute `foo` and `bar` concurrently. To ensure that reads and writes to the shared variable `x` do not execute at the same time, under any possible schedule, inserting a blocking FIFO write/read is sufficient to synchronize them. The `semaphore_set` on line 10 happens before `semaphore_wait` on line 15, so transitively the write to `x` on line 7 happens before the write to `x` on line 18: no read/write conflicts are detected during interpretation. The final CDAG computed for `x` during interpretation of this program would be `((x + 1) * 3)`.

But suppose the `f.write` and `f.read` are absent. We assume zero latency for operations: neither access to `x` happens before the other, so the interpreter would report a possible data race on `x`. However when synthesizing this program with a particular HLS toolchain, operations have non-zero latencies, and accesses to `x` may happen at different cycles even without the blocking FIFO. It is enough to have a latency of 1 cycle for `+` and 10 for `*` (assuming data accesses are achieved in 1 cycle) for this program to execute `foo` before accesses to `x` in `bar` are executed, leading to a deterministic execution.

Now suppose only `f.write` is absent. Interpreting `foo` runs to completion, however `bar` is actively waiting (blocking read) on `f`. As it cannot further change state, we report a deadlock as per Definition 5.

Our approach is a *conservative* analysis for concurrency correctness, which can incur false negatives: we may report conflicts that could be addressed by actual timing in the final design. We never generate false positives: if we report the absence of deadlocks and read/write conflicts, then under any possible valid concurrent schedule or HLS approach, the programs cannot have conflicts.

Chapter 8
Experimental Results

All experiments are performed on Intel Alder Lake Core i9 12900K, with 128GB of RAM, 30MB cache and running at 5.2GHz single-core frequency. All verification experiments use a single CPU core. We use AMD Vitis HLS v2022.1 to simulate and synthesize designs. Numerous HLS files for realistic accelerators were provided by Prof. Zhiru Zhang's team at Cornell for our evaluation: we report verification results on these files.

8.1 Verifying A Systolic Array Compiler

Systolic array compilers generate high-performance systolic designs from high-level functional descriptions [24, 50, 77, 84]. However, formally verifying the generated designs remains a challenge due to the complex transformations during compilation and the dataflow parallel nature of systolic architectures. Our colleagues at Cornell generated systolic arrays of different sizes for matrix multiply kernels with AutoSA [84], and we verify the generated HLS program against the input high-level functional description in C, which is a 5-line matrix-multiply kernel. Table 1 lists the verification results. On the left, we present sequential-only verification, and when considering fixed-depth FIFOs using coarse-grain dataflow on the right. The number of Lines of Code (LoC) in the input program, number of statements interpreted (Stmts), number of nodes in the final CDAGs for the live-in/out variables checked for equivalence (Nodes), time to complete interpretation of this file, and maximal memory used during the process. We note the significant time and memory cost of performing deadlock/race detection for any valid concurrent schedule; the number of concurrent Tasks and number of FIFOs is displayed.

Note that table 1 reports the performance of verifying concurrent programs using a simpler and more limited support for concurrency than presented in Chapter 7: instead it follows the approached published [65]. Chapter 7 presents a generalization of concurrent program verification that supports hierarchical parallelism.

The time to interpret the original 5-line matrix-multiply kernel, and verify the equivalence of CDAGs, is negligible here. It amounts to 2.5s for 64×64, less than 1s for all others. We have also manually inserted

bugs in the code, to validate that our tool can successfully catch them. No bugs were found in the codes produced by AutoSA.

Array Size	LoCs	#Tasks	#FIFOs	#Stmts	#Nodes	Time	Mem.
2×2	1.1k	-	-	1.7k	44	0.01s	4MB
		22	31	3.3k	44	0.01s	5MB
4×4	1.6k	-	-	7.5k	304	0.01s	5MB
		56	91	17k	304	0.02s	7.5MB
8×8	3.5k	-	-	41k	2.2k	0.05s	11MB
		172	307	109k	2.2k	0.11s	21MB
16×16	10.5k	-	-	268k	17k	0.32s	46MB
		596	1k	787k	17k	1.05s	132MB
32×32	37.6k	-	-	1.9M	134k	2.76s	447MB
		2.2k	4k	6.2M	134k	27.9s	1.6GB
64×64	144.6k	-	-	14M	1.06M	24.1s	5.9GB
		8.5k	16k	54M	1.06M	16m	34GB

Table 1: AutoSA Systolic array verification results. For each array size, the top row is sequential-only verification, bottom row uses blocking FIFOs.

8.2 Verifying Customizations in HeteroCL

HeteroCL is a domain-specific compiler with decoupled customizations for hardware accelerator designs [37, 49, 89].

PolyBench/HCL: The team at Cornell implemented and customized the PolyBench polyhedral benchmark suite [64] with HeteroCL, and we verify the customized kernels. PolyBench consists of 30 kernels covering data mining, linear algebra kernels and solvers, and stencil kernels. We customize the kernels with optimizations listed in Table 2.

We choose the medium kernel sizes for verification to demonstrate real-world problem sizes. The number of statements of the medium-size benchmarks ranges from 239K (`jacobi_1d`) to 1.6 billion (`floyd_warshall`), the median number of statements is 22M (`heat_3d`). Verification time ranges from 1.1 second to 2.1 hours, with a median run time of 192s. The memory footprint ranges from 0.1 MB to 172 GB, with a median memory footprint of 3.5 GB.

BERT: Transformer on FPGA Accelerator: Transformers deliver state-of-the-art performance for various tasks in NLP and vision [82]. The building block of transformer models is matrix-multiplication,

Optimization	Description
reorder	Loop reordering
tile	Loop tiling
stream	Use FIFO streaming between two HLS kernels
line/window buffer	Insert reuse buffers to cache rows/columns of input matrix
write buffer	Insert write buffers to cache partial results
double buffer	Create ping-pong buffers and alternating read/write logic
unify	Unify multiple functions for resource sharing
layout	Transform memory layout

Table 2: HLS optimizations considered in evaluation.

which provides abundant opportunities for hardware acceleration [46]. Our colleagues at Cornell built an FPGA accelerator for the BERT-base model [28] with 12 attention heads, an input feature dimension of 768, and a hidden dimension of 3072 in the feed-forward network. The BERT accelerator is implemented with HeteroCL, and customized with optimizations listed in Table 2. We deploy the accelerator on an AMD U280 FPGA with three Super-Logic Regions (SLRs). To meet the timing requirement at the routing stage, our colleagues added an additional customization to establish new function boundaries to group kernels and assign them to each SLR. The BERT accelerator verification executes 1.37B statements and checks 693M CDAG nodes, taking 27 minutes, and has a memory footprint of 56.9 GB.

8.3 Verifying Intel HLS Examples

Since we verify transformations before HLS, our verification method is not limited to any specific HLS tools or vendors. We verify the example HLS designs from Intel HLS [41] against their C reference program in the testbench. The Intel HLS sample designs consist of five kernels: counter, image downsample, interpolation and decimation filters, Gram-Schmidt QR factorization, and YUV-to-RGB color space conversion. The customizations of the HLS kernels include loop reordering, unrolling, and customized storage implementation. For example, the interpolation and decimation filter kernels use a temporary partial delay line to break loop-carried dependency. We verify all five cases in under 2 minutes. The example with the largest problem size is QR factorization, which takes 63.6 seconds to verify, and has a memory footprint of 1.4GB.

8.4 Verifying Compositions of Optimizations

Some optimizations do not change the program semantics alone, but may cause bugs when composed with other optimizations. Some optimizations only preserve semantics when composed with others. Our colleagues at Cornell implemented many transformations, both buggy and correct, to evaluate our system's ability to verify the correctness of compositions of transformations.

Our colleagues selected two representative kernels to apply specifications of HLS optimizations: `two-conv` for two back-to-back 2D convolution kernels, `binary-conv` for a binary convolution kernel on 4D input tensors. The input size of the `two-conv` kernel is 8×8, and both convolution kernels are 3 ×3. The `binary-conv` input size is $(N, C, H, W) = (1, 3, 8, 8)$ and the convolution kernel dimension is $(OC, IC, K, K) = (2, 3, 3, 3)$. Table 3 shows the verification results on optimizations, including the number of CDAG nodes and running time. We discuss each specification with cases listed in the table.

Kernel	Case	Optimizations	Bug	Detect	#Node	Run time (s)
two-conv	T.1	line buffer	no bug	PASS	10.9K	0.03
	T.2	stream	access pattern violation	✓	10.9K	0.02
	T.3	line buffer + stream	no bug	PASS	10.9K	0.04
binary-conv	B.1	reorder (nchw → nhwc)	no bug	PASS	16K	0.2
	B.2	line buffer + window buffer	no bug	PASS	16K	0.2
	B.3	line buffer + window buffer + reorder	loop order dependency violation	✓	16K	0.2
	B.4	layout (nchw → nhwc)	difference in input memory layout	✓	16K	0.2

Table 3: Results of verifying HLS optimization specifications — PASS indicates the optimized HLS program is bug-free, and the optimized program is verified to be semantically equivalent to the original program. ✓ indicates the optimized HLS program is not semantically equivalent to the original program, and our tool correctly reports the semantic differences.

FIFO stream and line buffers. The original `two-conv` program has an array to store the intermediate result. The second convolution kernel reads the intermediate array in a sliding window. Case T.1 adds a line buffer to the second convolution kernel to increase data reuse and serialize the data access. Case T.2 simply replaces the intermediate array with a streaming FIFO. Without a line buffer in the second kernel

to serialize its data access, this optimization causes an access pattern violation. Case T.3 implements the correct composition of both optimizations.

Loop reorder and reuse buffers. Some HLS optimizations make certain assumptions about the program, and further optimizations that break the assumptions can cause bugs. For example, reuse buffer insertion assumes a certain loop order to load correct data. In case B.1, we verify that loop reordering from channel-first (NCHW) to channel-last (NHWC) does not change program semantics. In case B.2, our colleagues first inserted a reuse buffer at height (H) loop to create a line buffer, then inserted another reuse buffer at width (W) loop to create a window buffer, and we verify that inserting reuse buffers does not cause bugs either. However, when reordering was applied after reuse buffer insertion, the output channel (C) loop is moved inside the width (W) loop, causing line buffer load repeated input rows. As shown in Table 3 case B.3, our tool detects this issue caused by optimization dependency violation.

Layout transformations. Our colleagues transformed the memory layout of the input multi-dimensional array from channel-first (NCHW) to channel-last (NHWC) in case B.4. Memory layout transformation benefits access locality, but changes the program semantics. Our tool correctly reports the difference in case B.4.

8.5 Detecting Simulation Mismatches

C simulation can miss critical issues such as over-bound array access, which leads to hard-to-debug issues and may only be discovered during RTL co-simulation. This case study demonstrates how our tool efficiently finds memory partition bugs that C simulation does not uncover. Listing 7 shows such an example: applying array partitioning on a loop kernel with over-bound array access.

Since C/C++ stores arrays in contiguous memory, the over-bound array access `A[i][3]` overflows to the next row `A[i+1][0]`. Such an over-bound access does not happen in the synthesized RTL design. Our tool symbolically evaluates the array index expression. For array partitioning, it creates separate arrays for each subarray, and treats the original array as a live-in variable. Therefore, it captures the discrepancy in array indices before and after the array partitioning.

Our tool takes 0.05s to verify and raise the semantic difference, C simulation takes 21s, while RTL co-simulation takes 1 minute. it uncovers subtle bugs that C simulation does not raise, while offering faster debugging and shorter turnaround time.

```
int A[4][3], B[3][3], C[3][3];
#pragma array_partition var=A dim=1 complete
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        C[i][j] = 0;
        for (int k = 0; k < 3; k++) {
            C[i][j] += A[i][k+1] * B[k][j];
} } }
```

Listing 7: Partitioning array with over-bound access.

8.6 Verification Time and Scalability

We typically verify functions such as GEMM at a rate of about 0.5 million statements per second, amounting to about 0.8MFlop/s, for problem sizes of 500^3 and less. The process of proving 2 CDAGs equivalent is typically a negligible factor in the total time, and time is dominated by the number of instructions to interpret. Limits are dominated by memory usage: the CDAGs grow in space consumption linearly with the number of operations, with $O(500M)$ FLOPs in reductions using 50GB. Future work includes run-time compression of CDAGs for increased scalability.

Note however as illustrated in Section 8.1, for concurrent verification there are significantly more instructions to execute due to the check-semaphore/update-semaphore operations that need to be executed, and more importantly, significantly larger space being consumed due to the bookkeeping of memory snapshots when tasks change status. 64×64 shows limits in memory usage.

Chapter 9
Conclusion

Proving the equivalence between two different implementations of the same program provides a verification of correctness of the optimizations for HLS implemented by either humans or tools. Focusing on source-to-source transformations for HLS and imposing sensible restrictions on the programs supported, we have developed a framework that can prove that the result of applying numerous fundamental optimizations, such as data buffering, FIFO-based communications and arbitrary loop transformations, preserves the exact semantics of the original program [65].

Our framework can also verify the correctness of advanced transformations, such as those implemented by an automatic systolic array generator. However, this comes at the cost of restricting the class of programs supported to statically-interpretable control-flow programs, which typically requires known problem sizes at verification time.

While we have provided the foundation for efficient hybrid concrete-symbolic interpretation for equivalence verification, we are currently researching numerous extensions. First, we target further extending the class of programs supported: from just SICF programs to allowing more than one execution path in the program. Supporting even a limited class of conditionals in an input program could significantly increase the coverage of our approach. However, to avoid the problem of an exploding number of paths to explore common to SMT-based approaches, we must restrict the types of paths modeled.

Second, we aim to improve the space complexity of the CDAGs built by the interpreter, which is currently linear in the number of operations executed. This complexity limits scaling to around a billion operations using about 100GB of memory. To extend this limit, we will investigate compressing CDAGs by exploiting regular patterns that occur in loop-based code.

Third, we plan to investigate verifying transformations for applications beyond those targeting HLS. Support for some OpenMP pragmas has already been added [81], and we plan to additionally support OpenMP tasks and other representations such as the Multi-Level IR (MLIR) [53] developed in LLVM.

References

[1] 2022. ISA 0.13. Retrieved from http://repo.or.cz/w/isa.git

[2] 2023. The KLEE Symbolic Execution Engine. Retrieved from https://klee.github.io/

[3] 2023. PoCC, the Polyhedral Compiler Collection 1.6. Retrieved from https://pocc.sourceforge.net/

[4] 2023. Programming in C. Retrieved from https://www.quut.com/c/

[5] AMD. AMD Vivado Design Suite. Retrieved from https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html

[6] AMD. Vitis HLS. Retrieved from https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

[7] AMD. Vitis High-Level Synthesis User Guide. Retrieved from https://docs.amd.com/r/en-US/ug1399-vitis-hls

[8] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016. 53–62.

[9] F. Baader and T. Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press.

[10] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[11] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys* 51, 3 (May 2018), 1–39. https://doi.org/10.1145/3182657

[12] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. 2022. Smt-based translation validation for machine learning compiler. In *International Conference on Computer Aided Verification*, 2022. 386–407.

[13] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. PolyCheck: Dynamic Verification of Iteration Space Transformations on Affine Programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[14] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of Satisfiability*. IOS press.

[15] Utpal Bora, Santanu Das, Pankaj Kukreja, Saurabh Joshi, Ramakrishna Upadrasta, and Sanjay Rajopadhye. 2020. LLOV: A Fast Static Data-Race Checker for OpenMP Programs. *ACM Transactions on Architecture and Code Optimization* 17, 4 (December 2020), 1–26.

[16] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008. 209–224.

[17] Cadence. Genus Synthesis Solution. Retrieved from https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

[18] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, February 2011. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[19] Prasanth Chatarasi, Jun Shirako, Martin Kong, and Vivek Sarkar. 2017. An extended polyhedral model for SPMD programs and its use in static data race detection. In *Languages and Compilers for Parallel Computing: 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers 29*, 2017. 106–120.

[20] Guang-Ien Cheng, Mingdong Feng, Charles E Leiserson, Keith H Randall, and Andrew F Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, 1998. 298–309.

[21] Ramanuj Chouksey and Chandan Karfa. 2020. Verification of Scheduling of Conditional Behaviors in High-Level Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2020).

[22] Basile Clément and Albert Cohen. 2022. End-to-end translation validation for the halide language. In *OOPSLA 2022-Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2022.

[23] Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco T Vasconcelos, and Max Grossman. 2017. Deadlock avoidance in parallel programs with futures: why parallel tasks should not wait for strangers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

[24] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *IEEE/ACM International Conference on Computer-Aided Design*, 2018.

[25] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2022).

[26] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (April 2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[27] Patrick Cousot. 2012. Formal Verification by Abstract Interpretation. In *Proceedings of the 4th international conference on NASA Formal Methods*, 2012.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[29] Steven Eker. 2003. Associative-commutative rewriting on large terms. In *International Conference on Rewriting Techniques and Applications*, 2003. 14–29.

[30] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2015. On characterizing the data access complexity of programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[31] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming* (1992).

[32] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, (1991), 23–53.

[33] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming* (1992).

[34] Paul Feautrier. 2006. Scalable and structured scheduling. *International Journal of Parallel Programming* 34, (2006), 459–487.

[35] Harry D. Foster. 2015. Trends in Functional Verification: A 2014 Industry Study. In *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*, June 2015. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2744769.2744921

[36] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, June 2005. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[37] Cornell Zhang Group. 2023. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. Retrieved from https://github.com/cornell-zhang/heterocl/releases/tag/v0.5

[38] Yizi Gu and John Mellor-Crummey. 2018. Dynamic data race detection for OpenMP programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018. 767–778.

[39] Yann Herklotz, James D Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[40] Intel. High Level Synthesis Compiler. Retrieved from https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[41] Intel. 2023. High Level Synthesis (HLS) Design Examples and Tutorials. Retrieved from https://www.intel.com/content/www/us/en/docs/programmable/683053/19-1/high-level-synthesis-hls-design-examples.html

[42] Feiyang Jin, Lechen Yu, Tiago Cogumbreiro, Jun Shirako, and Vivek Sarkar. 2023. Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, 2023. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 13–11.

[43] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. 2021. Effective processor verification with logic fuzzer enhanced co-simulation. In *54th IEEE/ACM International Symposium on Microarchitecture*, 2021.

[44] C. Karfa, C. Mandal, D. Sarkar, S.R. Pentakota, and C. Reade. 2006. A formal verification method of scheduling in high-level synthesis. In *7th International Symposium on Quality Electronic Design (ISQED'06)*, 2006.

[45] Chandan Karfa, Kunal Banerjee, Dipankar Sarkar, and Chittaranjan Mandal. 2013. Verification of loop and arithmetic transformations of array-intensive behaviors. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* (2013).

[46] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W Mahoney, and others. 2023. Full stack optimization of transformer inference: a survey. *arXiv preprint arXiv:2302.14017* (2023).

[47] Lucas Klemmer and Daniel Große. 2021. EPEX: processor verification by equivalent program execution. In *Proceedings of the Great Lakes Symposium on VLSI*, 2021.

[48] Steve Kommrusch, Martin Monperrus, and Louis-Noël Pouchet. 2023. Self-Supervised Learning to Prove Equivalence Between Straight-Line Programs via Rewrite Rules. *IEEE Transactions on Software Engineering* (2023).

[49] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. 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*, 2019.

[50] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, and others. 2020. Susy: A programming model for productive construction of high-performance systolic arrays on fpgas. In *39th International Conference on Computer-Aided Design*, 2020.

[51] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.* 14, 4 (September 2021). https://doi.org/10.1145/3469660

[52] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[53] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2021.

[54] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM* 52, 7 (2009), 107–115.

[55] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F Donaldson, Rafael Zahl, and Klaus Wehrle. 2017. Floating-point symbolic execution: a case study in n-version programming. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017. 601–612.

[56] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.

[57] Grant Martin and Gary Smith. 2009. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers* 26, 4 (July 2009), 18–25. https://doi.org/10.1109/MDT.2009.83

[58] Anmol Mathur, Masahiro Fujita, Edmund Clarke, and Pascal Urard. 2009. Functional Equivalence Verification Tools in High-Level Synthesis Flows. *IEEE Design & Test of Computers* 26, 4 (2009), 88–95. https://doi.org/10.1109/MDT.2009.79

[59] George C Necula. 2000. Translation validation for an optimizing compiler. In *ACM SIGPLAN conference on Programming language design and implementation*, 2000.

[60] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, Ponnuswamy Sadayappan, and Fabrice Rastello. 2020. Automated derivation of parametric data movement lower bounds for affine programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[61] Debjit Pal, Yi-Hsiang Lai, Shaojie Xiang, Niansong Zhang, Hongzheng Chen, Jeremy Casas, Pasquale Cocchini, Zhenkun Yang, Jin Yang, Louis-Noël Pouchet, and others. 2022. Accelerator design with decoupled hardware customizations: benefits and challenges. In *59th ACM/IEEE Design Automation Conference*, 2022.

[62] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. 2022. A Survey on Hardware Accelerators: Taxonomy, Trends, Challenges, and Perspectives. *Journal of Systems Architecture* 129, (August 2022), 102561. https://doi.org/10.1016/j.sysarc.2022.102561

[63] Louis-Noël Pouchet and Emily Tucker. 2023. PAST, the PoCC AST Library, version 0.7.2. Retrieved from https://sourceforge.net/projects/pocc/files/1.6/testing/modules/past-0.7.2.tar.gz,https://doi.org/10.5281/zenodo.10449349

[64] Louis-Noël Pouchet and Tomofumi Yuki. 2023. PolyBench/C 4.2.1. Retrieved from https://polybench.sourceforge.net/

[65] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodr\iguez, and Zhiru Zhang. 2024. Formal Verification of Source-to-Source Transformations for HLS. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2024. 97–107.

[66] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. Automatic hardware pragma insertion in high-level synthesis: A non-linear programming approach. *ACM Transactions on Design Automation of Electronic Systems* 30, 2 (2025), 1–44.

[67] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. A unified framework for automated code transformation and pragma insertion. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2025. 187–198.

[68] David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[69] Bartosz Rycko. Vitis - RTL - BLACKBOX. Retrieved from https://www.hackster.io/bartosz-rycko/vitis-rtl-blackbox-aa359b

[70] Benjamin Carrion Schafer and Zi Wang. 2020. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (October 2020), 2628–2639. https://doi.org/10.1109/TCAD.2019.2943570

[71] Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. 2020. Symbolic partial-order execution for testing multi-threaded programs. In *International Conference on Computer Aided Verification*, 2020. 376–400.

[72] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Softw. Eng. Notes* 30, 5 (September 2005), 263–272. https://doi.org/10.1145/1095430.1081750

[73] KC Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. 2005. Verification of Source Code Transformations by Program Equivalence Checking. *CC* (2005).

[74] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2015. ACM, Austin Texas, 1–12.

[75] Siemens EDA. Catapult HLS. Retrieved from https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/

[76] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Trans. Des. Autom. Electron. Syst.* (2022).

[77] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, and others. 2019. T2S-Tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[78] Bradley Swain, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. 2020. OMPRacer: A Scalable and Precise Static Race Detector for OpenMP Programs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2020. 1–14.

[79] Synopsys. Design Compiler. Retrieved from https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[80] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*, January 2009. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[81] Emily Tucker and Louis-Noël Pouchet. 2024. Verification of Concurrent Programs Using Hybrid Concrete-Symbolic Interpretation. In *Springer LNCS, Vivek Sarkar Festschrift, ACM SPLASH'24*, October 2024.

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30, (2017).

[83] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. on Programming Languages and Systems (TOPLAS)* (2012).

[84] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

[85] Yanzhao Wang, Fei Xie, Zhenkun Yang, Pasquale Cocchini, and Jin Yang. 2024. A Systematic Translation Validation Framework for MLIR-Based Compilers. (2024).

[86] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932.

[87] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* (2021).

[88] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a Free Verilog Synthesis Suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[89] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22)*, 2022. Association for Computing Machinery, Virtual Event, USA, 78–88. https://doi.org/10.1145/3490422.3502369

[90] AMD Xilinx. 2022. Merlin. Retrieved from https://github.com/Xilinx/merlin-compiler

[91] Xilinx. 2015. Advanced Synthesis Techniques. Retrieved from https://www.xilinx.com/publications/prod_mktg/club_vivado/presentation-2015/paris/Xilinx-AdvancedSynthesis.pdf

[92] Fangke Ye, Markus Schordan, Chunhua Liao, Pei-Hung Lin, Ian Karlin, and Vivek Sarkar. 2018. Using polyhedral analysis to verify openmp applications are data race free. In *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2018. 42–50.

[93] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better together: Unifying datalog and equality saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 468–492.

[94] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. AutoPilot: A Platform-Based ESL Synthesis System. *High-Level Synthesis: From Algorithm to Digital Circuit* (2008), 99–112.

[95] Jaroslav Ševč\ik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)* 60, 3 (2013), 1–50.

Personal Bibliography

Peer-reviewed international conferences:

- Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, Zhiru Zhang. Formal Verification of Source-to-Source Transformations for HLS. 2024. In *32nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (FPGA'24). *Best Paper Award.*

Peer-reviewed international workshops:

- Emily Tucker and Louis-Noël Pouchet. Verification of Concurrent Programs Using Hybrid Concrete-Symbolic Interpretation. In *Springer LNCS, Vivek Sarkar Festschrift, ACM SPLASH'24, October 2024.*