Formal Verification of Source-to-Source Transformations for HLS

Louis-Noël Pouchet
Colorado State University

Emily Tucker
Colorado State University

Niansong Zhang
Cornell University

Hongzheng Chen
Cornell University

Debjit Pal
University of Illinois Chicago

Gabriel Rodríguez
CITIC, Universidade da Coruña

Zhiru Zhang
Cornell University

ABSTRACT

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. 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 semantics 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.

CCS CONCEPTS

• **Software and its engineering → Software verification**.

KEYWORDS

Program equivalence, formal verification, high-level synthesis

ACM Reference Format:
Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, 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 (FPGA '24), March 3–5, 2024, Monterey, CA, USA*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3626202.3637563

FPGA '24, March 3–5, 2024, Monterey, CA, USA

ACM ISBN 979-8-4007-0418-5/24/03.
https://doi.org/10.1145/3626202.3637563

1 INTRODUCTION

Over the past decade, high-level synthesis (HLS) has been increasingly adopted for specialized hardware design targeting FPGAs and ASICs [9, 10, 27]. To achieve good performance, users apply source-to-source transformations on HLS algorithm specifications, necessitating substantial program rewriting and intricate compositions of program transformations. Such customizations for HLS, whether applied manually by a designer, or by (domain-specific) compilers for HLS such as AutoSA [43], HeteroCL [25] or the AMD Merlin compiler [47], may introduce subtle bugs that cannot be easily detected during testing or simulation. It remains a fundamental challenge to efficiently verify the correctness of a program optimized for HLS under a rich set of hardware-oriented optimizations.

Modern HLS design process typically starts from an algorithmic description and undergoes a series of source-to-source transformations [32] before synthesis. The goal is to transform the algorithmic descriptions to hardware-friendly ones that can generate high-quality RTL. While some transformations can be expressed by adding vendor-specific pragmas, many crucial optimizations require substantial changes in control-flow structure, I/O approach, on-chip buffer management, function boundaries, and exposing concurrency. For example, to transform a matrix multiplication kernel into a high-throughput systolic array (SA), one needs to build a customized memory hierarchy, an array of vectorized processing elements (PEs), and an I/O network [43]. Finally, one needs to tune the design to maximize the performance while meeting the resource constraints, and enable parallel execution of the PEs. Every source-to-source transformation applied to the files supplied to the HLS toolchain for eventual synthesis involves extensive rewriting. This process is prone to errors and can be time-consuming to debug.

We target the problem of verifying the correctness of source-to-source transformations by proving the semantics equivalence between two programs. We leverage two properties that often manifest in high-performance HLS designs: fixed kernel sizes and static control flow. Many spatial architectures have a fixed size, e.g. SAs [43]. Many hardware accelerated applications such as video processing, convolutional neural networks, and large language models typically have a static control flow. Technically we support any control flow whose branches can be unambiguously evaluated through *concrete interpretation*. This set of programs is formally defined as *statically interpretable control-flow (SICF) programs* in Sec. 3.

In this work we propose a novel framework to prove the semantic equivalence between a pair of programs, where one program is a substitute for the other, built after applying source-to-source transformations for HLS. We overcome the difficulty to support a rich set of optimizations when proving their equivalence: *our equivalence system is agnostic to how the code is implemented*, be it in terms of storage or loop structure, communication scheme, etc. This

future-proof design enables support for emerging optimizations. In particular, we can prove equivalence under any loop transformation approach, code refactoring, insertion of local/reuse buffers, insertion of FIFOs for communication, including blocking FIFOs for communication/synchronization using coarse-grain dataflow-style concurrency, etc. However, we note that by design our work is independent from any specific HLS toolchain: *we verify the equivalence of two C/C++ source programs that are input to a HLS toolchain.* Assuming the HLS backend used is correct, then whichever the HLS approach implemented, necessarily the two programs would compute the same output if given the same inputs, if we have verified them equivalent. We make the following contributions:

- We present an end-to-end, fully implemented system to prove the equivalence between a pair of functions 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 provide extensive experimental evaluation, demonstrating the ability of our system to *quickly* verify the correctness of advanced optimizations, including AutoSA-generated matrix-multiply 64×64 systolic array designs over 145,000 lines of code, the inference of a full BERT layer optimized for FPGAs, or a variety of numerical kernels optimized with HeteroCL, in seconds to minutes while using a single CPU core.

2 APPROACH TO VERIFICATION

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 [28], encompasses a broad range of applications such as linear algebra, image processing, data mining, machine learning, physics simulation, and more [37], as well as modern deep learning inference computations [25, 35]. Our coverage extends way beyond programs that are *syntactically analyzable* as polyhedral programs: in Sec. 3, 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 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 (e.g., scalarization, local and multi-buffering, data transfers 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 formal 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 equivalent 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* [12], 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 shall be statically *analyzable* (e.g., using polyhedral structures [15]), we typically require the problem sizes to be known at compile-time. We do not support parametric loop nest analysis. This requirement arises also when performing co-simulation or testing, and can be partially alleviated by proving equivalence once for each element in a set of problem sizes.

With these objectives and restrictions in mind, we now introduce the key principles of our approach, each developed later in this paper. We illustrate the concepts to prove equivalent the pair of simple programs shown in Lst. 1. 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.

```
// Program P_A, 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 P_B, a replacement for program P_A:
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 1: Illustrating example: dense matrix-vector product.

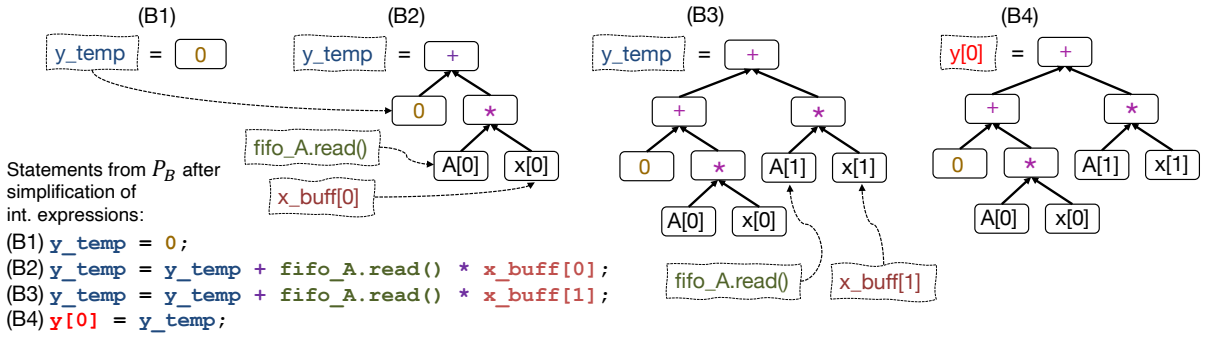

Figure 1: Excerpt of interpretation for `y[0]` with $N = 2$ for P_B — When interpreting a statement, first every variable referenced is replaced by its content from the interpreter memory, if any. Integer sub-expressions are then simplified with concrete evaluation, and the result is stored in memory: 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` by 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]`.

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 [42], based on static analysis, and PolyCheck, based on a more general dynamic analysis [6] 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 Sec. 4.
- We prove equivalence by computing if this canonical representation is *identical* between both programs, for *every live-in/live-out memory cell*. This is presented in Sec. 5.
- 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 as 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, then and only then we can prove the programs equivalent, if their per-cell computation representations are fully identical. This is presented in Sec. 3.
- We prove equivalence when using FIFOs of a given depth, non-blocking and blocking, sequentially and with coarse-grain dataflow-style concurrent execution of functions that write/read the FIFOs, as in programs generated by AutoSA. This is presented in Sec. 6.

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) [14, 34]. We formally define CDAG in Sec. 4.1. Fig. 1 shows an excerpt of CDAGs built for program P_B. Every variable in the program which can be *concretely* evaluated is, that is expressions such as `i*N + j` are *replaced by their result* during interpretation, giving values 0, 1, .. 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]`, `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.

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 Sec. 6 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 fully isomorphic. If so, we have proven the programs are equivalent. We can catch errors in the loop nests, handling of FIFO (e.g. incomplete data, deadlocks, etc.), in how statements are scheduled wrt. dependences, 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 Sec. 5, 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 agreed on.

3 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, 36] 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].

3.1 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 is made of:

- An AST traversal mechanism, that implements all control-flow operations of the program, including function calls, variable declarations, etc. This is presented in Sec. 3.2.
- 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 Sec. 3.4.
- A memory storage system, to store concrete and symbolic values for every variable touched during interpretation, this includes temporary/local variables. We implement a dynamically allocated sparse tensor approach to this end, this is presented in Sec. 3.3.
- A CDAG building system, which manipulates symbolic expressions building them by partial evaluation, presented in Sec. 4.

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

3.3 Interpreter Memory

The interpreter maintains a memory for the program. Every variable (or memory cell for an array) 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, and 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 $0-42\times 0-51$ would consume unnecessary memory.

3.4 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);` by `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 it would behave on the target hardware for our system to conclude a proof of equivalence, as 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 bug-free; using certified compilers such as CompCert [29] may be desired for increased confidence.

3.5 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, reusing prior computations whenever possible, and caching per-statement analysis.

We remark the complementary nature of approaches such as KLEE [38] 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 [31] 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 the control-flow can be completely computed by concrete interpretation at compile-time. Combining both approaches is the subject of future work. It 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: InterpretSequentialProgram

Result: Program Interpretation
Input : AST A for a program, using C semantics
Output : Set of all CDAGs for each memory cell

```
1  s := GetFirstInstructionInCFG(A)
2  while s do
3      s' := clone(s)
4      for each variable v in s' in postfix do
5          s' := replaceWithConcreteValueIfExists(v, s')
6      s' := concreteEvaluationInPostfixOfOps(s')
7      <writeLocation,Expr> := extractFromStatement(s')
8      if Expr contains variable references then
9          if Expr used in branch or array subscript ABORT
10         Expr := BuildCDAGByReplacement(Expr)
11         storeAsSymbolicCDAG(writeLocation, Expr)
12     else
13         storeAsConcreteValue(writeLocation, Expr)
14     s := GetNextInstructionInCFG(A, s)
```

4 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 Sec. 5, 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.

4.1 Formal Definition

A CDAG captures an expression that computes a single value [14].

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 [14]. 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 Fig. 1.

4.2 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 operand 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 simplified from integer computations, e.g. the tree is now `y[0] + A[0] * x[0]`. For every variable left referenced in the AST, we replace it by 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 by 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.

4.3 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 $O(n)$ *operations*, its CDAG will have at least $O(n)$ nodes. Taking for example matrix multiplication, it has $O(n^3)$ FMA operations, therefore the CDAG will 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, leading to a final CDAG of $O(3^n)$ 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.

Finally, the complexity of our implementation for CDAG construction, both in time and space, is typically $O(n)$ for a program executing $O(n)$ operations for sequential verification. This aspect is fundamental to the performance of verification, as shown in Sec. 7.

5 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 Sec. 6. Intuitively the process amounts to checking full isomorphism of the CDAGs produced by P_A and P_B for the same memory cell, this 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.

5.1 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). *Given two programs 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 iff 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, which can only be a function of live-in data, per Th. 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, they 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 1 (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 for each CDAG perform a merged prefix+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.

5.2 CDAG Normalizations

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. That is, `y[i] += A[i*N+j]*x[j]` is not equivalent with `y[i] += x[j]*A[i*N+j]`: the CDAGs are not identical. To handle a wider class of equivalences, we augment the system with a post-normalization of CDAGs, if checking their isomorphism originally fails. This post-processing has a polynomial time complexity, while so far the entire processes presented had a time complexity linear in the number of operations executed by the programs.

Specifically, the *user* can declare valid semantics-preserving rewrite rules to be applied on the CDAGs. For example, commutativity may be allowed for floating point operations: $+_{fp}(a,b) \leftrightarrow +_{fp}(b,a)$ where a, b are arbitrary subtrees. Then, for every rewrite rule provided, which may include rules that change operation count such as distributivity/factorization, we modify the CDAGs to apply them greedily until no more change can be achieved. This does not ensure completeness, but computes quickly enough to maintain practicality for the framework. In contrast, techniques like equality saturation [45] may be able to saturate the representation and bring completeness, but at a very high computational cost that is impractical for trees of thousands/millions of nodes as we manipulate. Note for the special case of associativity/commutativity, we implement a much faster approach, by simply sorting every commutative node in the CDAG and their children using a lexicographic ordering of the subtrees, leading to a canonical CDAG representation under associativity/commutativity.

6 VERIFICATION WHEN USING 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 *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.

6.1 FIFOs in Sequential Verification

Our approach is a straightforward extension of our sequential verification approach presented earlier. 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 by our own emulating read/write implementation, replacing them simply 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, and for reads we do the converse with `start`. This simple approach is not able to catch concurrency bugs such as deadlocks or races, in contrast to our dataflow-style verification below, however it allows to catch bugs in program restructuring and how the FIFO is used, while maintaining our target complexity of $O(n)$ time/space for $O(n)$ statements executed in the program.

6.2 FIFOs in Dataflow-style Programs

Handling FIFOs of a fixed depth in a dataflow-like implementation creates the need to handle a form of concurrent execution between functions, to expose possible deadlocks and race conditions. We limit in this work to supporting a coarse-grain dataflow-style of concurrency, for example coarse-grain `#pragma HLS dataflow` regions as supported by Vitis. Yet our work paves the way for support of more general concurrency between functions, as to address this problem we (a) made our interpreter robust to multitasking and interrupt-based halt/resume of function interpretation; and (b) designed a novel approach to interpretation that involves heavy memory snapshots and bookkeeping so that *our verification holds for any possible valid concurrent schedule that can be executed*, this while interpreting a single of these valid concurrent schedules.

Concurrent-capable interpreter. We equip our sequential interpreter with two important features. First, the ability to schedule a function to execute from a list of functions that are executing concurrently (e.g., those listed under the dataflow-style pragma). Second, the ability to interrupt and resume a particular function, at any point: this means we can implement waiting/interrupting a function until a particular semaphore has reached a value, and resume it when it did. We use this interrupt and semaphore approach to represent the blocking FIFO read/write, using semaphores to capture when a FIFO is ready to read or write.

Verification of concurrent functions. Our approach is fully implemented, and available in our open-source verification tool. Due to space constraints, we limit to passing the key intuitions regarding computing the earliest schedule of a concurrent program, and using this information to check for the absence of deadlocks and the absence of any read/write conflicts: data being read/written by different concurrent functions at possibly the same time in some valid concurrent schedule, and not present proofs here.

Concurrent verification approach. By design, our approach is fully independent of the target HLS toolchain used, we therefore implement a *conservative* verification: if there exists a schedule that can deadlock or race, amongst all possible valid scheduling or HLS approaches used, then we report so. In particular, we assume every instruction may execute in zero cycle, and only blocking reads/writes may trigger the necessity for some operations to happen before some others. That is, all synchronizations between concurrent functions that read/write to the same shared memory location must be explicit in the program, by using a blocking FIFO to synchronize these read/writes. To implement FIFOs, we use semaphores to communicate their readiness state between concurrent functions. FIFOs are implemented as arrays as for the sequential case, we now aim to ensure no deadlock nor read-write conflict.

Definition 3 (Semaphore and timestamps). *A semaphore is a counter with value v that represents the number of elements in a blocking FIFO with depth d; a blocking read is possible when $v > 0$ and a blocking write is possible when $v < d$. A timestamp $T_S^f \in \mathbb{N}$ is assigned to every semaphore S used in function f. Its value is 0 at start, and is incremented when a blocking read/write operation on S in f becomes ready.*

We use T_S^f to build a monotonically increasing timestamp $T_{S,f}(s)$ for all interpreted operations s (in their sequential order) in f that depend on the readiness of S, and perform subsequent checks for deadlocks and read-write conflicts. We have:

Definition 4 (Minimal Timestamp and concurrency). *Given two interpreted operations s_1, s_2 in the full program. Given $Tm_{f_i}(s_1) = \min_S(T_{S,f_i}(s_1))$ the minimal timestamp of s_1 in f_i (resp. s_2 in f_j). If $Tm_{f_i}(s_1) < Tm_{f_j}(s_2)$ then necessarily under any valid execution of the program s_1 must complete its execution before s_2 starts. Conversely, if $T_{S,f_i}(s_1) = T_{S,f_j}(s_2)$ then there is a possible concurrent schedule executing both operations at the same time.*

It is therefore sufficient to check the absence of read-write conflicts by considering any operation that accesses data shared between concurrent functions if they have the same timestamp:

Definition 5 (Read/write conflict). *Given s_1, s_2 such that they access the same shared memory location, that is data non-local to the concurrent functions. If $\exists\ T_{S,f_i}(s_1) = T_{S,f_j}(s_2)$ and one of these accesses is a write, then a read-write conflict can occur on a possible concurrent schedule.*

If all concurrent functions have the same timestamp for each semaphore for every operation, they can be executed in any order and the semaphore values will be valid, but this is not the case otherwise. We must restore the semaphore values appropriately before resuming a concurrent function. This bookkeeping is done for all timestamp values associated to shared variables accesses. It increases space consumption, but at the benefit of allowing to interpret a single (specific) concurrent schedule while still proving correctness for all possible valid schedules.

Executing all functions concurrently requires storing the value history for semaphore S from $T^{min}...T^{max}$, where T^{min} and T^{max} are the current minimum and maximum timestamp values for S over all concurrent functions. Before resuming interpretation of a function the value of each semaphore at the relevant timestamp can be restored, and read-write conflicts can be checked across a range of histories for a particular memory cell, allowing to emulate fully concurrent execution sequentially. Details are available in our implementation [36]. This backup/restore step is fundamental for the general correctness of our approach, however it incurs a low-polynomial complexity, slowing the verification, as shown in Sec. 7 below.

Finally, deadlocks are detected using a simple approach: if the interpreter has one or more currently executing functions (i.e., not completed by reaching the end of their control-flow) that cannot make further progress, because of a blocking operation which has not reached a ready-state, then we report a deadlock:

Definition 6 (Deadlock). *Given $f_1, ..., f_n$ a set of concurrently executing functions. If $\exists\ f_i$ in a blocking state which depends on semaphore S and no other f_j can update S, either because they completed or because they do not modify S, then the program deadlocks under a possible concurrent schedule.*

We remark the importance of histories on semaphore values and restoring them before resuming a concurrent function, to consider all possible values for a semaphore to change state, and therefore conclude the absence of deadlocks.

Table 1: AutoSA Systolic array verification results. Left is sequential-only verification, right uses blocking FIFOs.

Array Size	LoCs	#Stmts	#Nodes	Time	Mem.	#Workers	#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

Example and Discussions. 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:

```
float x; // shared
fifo_t f;
void foo(int a) { a += 1; x += 1; f.write(42); }
void bar(int a) { a *= 3; f.read(); x *= 3; }
```

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. Specifically, operations of `foo` are assigned a timestamp, 0, for all. Similarly for `bar`, up to the blocking read. When the read changes status (data is available to read after interpreting `f.write(42)`), the timestamp for the next operations in `bar` is incremented to 1. When checking whether a read/write conflict exists, we have the tuples $foo :: (0, x, read)$ and $bar :: (1, x, write)$ being distinct, so no conflict is reported here ($1 \neq 0$). Note the final CDAG for `x` is $(x + 1) * 3$ here.

But suppose the `f.write` and `f.read` are absent. We assume zero latency for operations: the accesses to `x` would occur at the same virtual timestamp: 0, for both `foo` and `bar`, and we would report a read/write conflict on `x`: $foo :: (0, x, read)$ and $bar :: (0, x, write)$ have identical prefix and a read-write sequence. 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 Def. 6.

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 any conflict.

7 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.

7.1 Verifying A Systolic Array Compiler

Systolic array compilers generate high-performance systolic designs from high-level functional descriptions [11, 26, 40, 43]. However, formally verifying the generated designs remains a challenge due to the complex transformations during compilation and the dataflow parallel nature of systolic architectures. We generate systolic arrays of different sizes for matrix multiply kernels with AutoSA [43], and verify the generated HLS program against the input high-level functional description in C, which is a 5-line matrix-multiply kernel. Tab. 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 Workers and number of FIFOs is displayed. Note 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 bug was found in the codes produced by AutoSA.

7.2 Verifying Customizations in HeteroCL

HeteroCL is a domain-specific compiler with decoupled customizations for hardware accelerator designs [16, 25, 46].

PolyBench/HCL: We implement and customize the PolyBench polyhedral benchmark suite [37] with HeteroCL, and 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 Tab. 3. 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: Transformer delivers state-of-the-art performance for various tasks in NLP and vision [41]. The building block of transformer models is matrix-multiplication, which provides abundant opportunities for hardware acceleration [22]. We build an FPGA accelerator for the BERT-base model [13] with 12 attention heads, an input feature dimension

Table 2: 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.

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

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 Tab. 3. We deploy the accelerator on an AMD U280 FPGA with three Super-Logic Regions (SLRs). To meet the timing requirement at the routing stage, we add 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.

Table 3: HLS optimizations considered in evaluation.

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

7.3 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 [18] 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.

7.4 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. We select 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 $(OC, IC, K, K) = (2, 3, 3, 3)$. Tab. 2 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.

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, we first insert a reuse buffer at height (H) loop to create a line buffer, then insert 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 we apply reordering 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 Tab. 2 case B.3, our tool detects this issue caused by optimization dependency violation.

Layout transformations. We transform 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.

7.5 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. Lst. 2 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 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.

```
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 2: Partitioning array with over-bound access.

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.

7.6 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 Sec. 7.1, 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 workers change status. 64x64 shows limits in memory usage.

8 RELATED WORK

Our framework can prove the equivalence of C-style programs under a wide set of code transformations, albeit limited to the class of Statically Interpretable Control-Flow. We are not aware of any tool which can support the same set of programs and/or transformations, but numerous prior works address a similar problem. PolyCheck is a system to prove equivalence of an affine program and its transformed variant via dynamic analysis [6] and supports "arbitrary" iteration reordering transformations. It is however fundamentally limited by the need to find a matching between statements in both programs, preventing it from supporting statement transformations, as well as data/storage transformations. In contrast, our framework does not require the programs to be affine, and supports a vastly richer class of transformations. ISA [1, 42] supports parametric loop bounds, and proves equivalence between a pair of programs. However it remains highly limited in transformation coverage [6], preventing its deployment for complex HLS optimizations. Other approaches have been developed, e.g. [20, 39] however they are typically limited in applicability, that is the space of program transformations supported. In general, numerous approaches to prove the equivalence of expressions, in restricted contexts, have been developed, including using equality saturation [44, 45], however the computational complexity of such approaches prevents manipulating complex programs with a rich transformation coverage. 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 [24].

An approach complementary to ours is KLEE [2, 38]. 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., [5, 23]. Preliminary extensions for floating point support have also been developed [30], however they do not scale to the problem sizes nor program complexity we manipulate. 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, as well as operating on C semantics. It therefore very significantly widens the class of programs supported for equivalence checking in feasible time.

Other approaches to check the correctness of a program transformation include translation validation [33], including for specialized languages like Halide [8], and HLS [7, 21]. Vericert is a verified HLS framework [17], akin to CompCert [29]. We target source-to-source equivalence, outside of a compiler framework, making our tool agnostic to the optimization framework used, and supporting user-implemented code transformations. Testing is also a classical approach, typically checking the output data produced by two programs is identical, but no proof is guaranteed. Finally, co-simulation is often performed before final deployment to ensure a design's correctness [19], but as shown in Sec. 7 this execution-based approach may miss bugs that our framework can detect.

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. 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.

ACKNOWLEDGEMENTS – This work was supported in part by an Intel ISRA award; U.S. NSF awards #1750399 and #2019306; ACE, one of seven centers in JUMP 2.0, an SRC program sponsored by DARPA; and Grant PID2022-136435NB-I00, funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", EU. We are particularly thankful to Jin Yang, Jeremy Casas, and Zhenkun Yang from Intel for their support and guidance on the ISRA project. We also thank Lana Josipović and the anonymous reviewers for their feedback on earlier versions of this manuscript.

REFERENCES

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

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

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

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

[5] 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*.

[6] 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*.

[7] 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).

[8] 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*.

[9] 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).

[10] 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* (2011).

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

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

[13] 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).

[14] 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*.

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

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

[17] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal Verification of High-Level Synthesis. *Proc. ACM Program. Lang.* (2021).

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

[19] 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*.

[20] 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).

[21] 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)*.

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

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

[24] 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).

[25] 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*.

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

[27] 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, Article 17 (sep 2021), 39 pages. https://doi.org/10.1145/3469660

[28] 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*.

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

[30] 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)*. IEEE, 601–612.

[31] 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*.

[32] 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

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

[34] 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*.

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

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

[37] Louis-Noel Pouchet and Tomofumi Yuki. 2023. PolyBench/C 4.2.1. https://polybench.sourceforge.net

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

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

[40] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. 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)*.

[41] 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).

[42] 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).

[43] 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*.

[44] 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 (jul 2020), 1919–1932.

[45] 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).

[46] 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* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 78–88. https://doi.org/10.1145/3490422.3502369

[47] AMD Xilinx. 2022. Merlin. https://github.com/Xilinx/merlin-compiler