Automated Translation Validation of a Compiler for Statically Scheduled Accelerators

Jackson Melchert, Stanford University, Stanford, United States, melchert@stanford.edu
Caleb Terrill, Stanford University, Stanford, United States, cterrill@stanford.edu
Aron Ricardo Perez-Lopez, Stanford University, Stanford, United States, arpl@cs.stanford.edu
Clark Barrett, Stanford University, Stanford, United States, barrettc@stanford.edu
Priyanka Raina, Stanford University, Stanford, United States, praina@stanford.edu

***Abstract*—Compilers for programmable hardware accelerators are complex and involve progressively lowering an application down to the hardware. Bugs can be introduced at many different stages, but simulation does not provide full bug coverage and has poor bug localization. We propose a methodology for automated formal translation validation of compilers for statically scheduled accelerators. This includes generating symbolic representations of every stage in the application compiler and the hardware, leveraging scheduling information for automatically generating translation validation queries, and implementing performance enhancements for effective formal verification. This work provides a blueprint for rigorous verification of compilers and generators for hardware accelerators.**

I. Introduction

With the slowdown in technology scaling, domain-specific hardware accelerators have become key to improving the performance and efficiency of compute-intensive applications. Recent research explores programmable accelerator generators, with accompanying compilers providing the means for designing these systems productively. These accelerators, while diverse (e.g., [21], [31], [37], [36], [27], [14], [16]), all use complex multistage processes to compile code from high-level languages down to accelerator instructions. Verification of these systems typically relies on comparing "golden" outputs from CPU implementations against accelerator outputs in register-transfer level (RTL) simulation, which is a slow and incomplete process for nontrivial systems. Bug root cause analysis is also challenging due to the numerous compiler stages involved.

Formal translation validation [32], [29], [38] offers a more rigorous approach. While prior work exists for software compilers (e.g., GCC [29], Halide [11]) and high-level synthesis (HLS) [23], [22], [20], ours is the first work to apply translation validation to the complex software-hardware compilation flow found in accelerator application compilers. Importantly, our work requires tackling new challenges, including dealing with many heterogeneous intermediate representations and handling notions of equivalence modulo scheduling differences.

In this paper, we present a full case study demonstrating an automated formal translation validation methodology for a statically scheduled accelerator compiler targeting a coarse-grained reconfigurable array [21]. Our approach provides guarantees against potential bugs introduced by the compiler and also enables quick localization of any detected bugs to a specific stage in the compiler. Additionally, as our system uses a formal representation of the accelerator hardware together with various intermediate representations (IRs) in the same semantic framework, both the hardware and software compiler results are verified simultaneously.

To treat the heterogeneous multistage pipeline uniformly, we encode everything, including the application, the compiler's IR, and the hardware RTL design, as transition systems represented using satisfiability modulo theories (SMT) formulas. This translation is automated and can adapt to both hardware and compiler changes. We define what it means for two representations to be equivalent by taking into account scheduling information present in the application compiler and then use this definition to check the equivalence of successive stages using an SMT-based model checker. Finally, we extend our technique with *symbolic starting states*, which vastly improves the performance of the system.

The contributions of our case study include:

- A uniform SMT-based formal representation for every intermediate stage in the compiler, encompassing both hardware and software, including the front-end application code, the dataflow IR, and the RTL hardware design.
- A methodology for formal verification via translation validation across every major stage of the application compiler. Our technique leverages scheduling information present in the compiler to automatically manage timing differences across stages.
- A technique for improving translation validation performance through the use of symbolic starting states with automatically generated state constraints.

We evaluate our translation validation system on 20 applications with a wide range of complexity. We verify every stage of the application compiler for these applications. The runtime overhead of these verification checks ranges from 1 minute for simple applications to 375 minutes for complex applications. The use of symbolic starting states significantly reduces memory consumption and runtime and is necessary for the verification of large applications. We introduced bugs in each stage of the compiler and confirmed that our approach can find all of them. Additionally, we found several previously unknown real bugs in our application compiler.

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_26

II. Related Work

This work focuses on translation validation [32], [29], [3], a formal technique for proving the equivalence of two pieces of code, before and after translation by a compiler. Translation validation aims to prove that a particular run of a compiler pass introduces no bugs, rather than trying to prove in advance that the compiler pass will never introduce bugs. In order to apply translation validation, the code that is being validated before and after the translation needs to be represented in the same semantic framework. We leverage the language of SMT [5] for this purpose.

Translation validation has been used to verify software like Halide as it gets compiled from a high-level description into C code [32], [29], [3], hardware as it goes through optimization passes [15], and HLS systems that compile C to Verilog [23], [22], [20], [17]. However, no prior work addresses the problem of performing translation validation throughout a complex end-to-end application compiler flow targeting hardware accelerators.

HLS systems generate Verilog hardware from a C program, while the application compilers targeted in this work compile applications to existing hardware. This important distinction means that much of the infrastructure created for verifying HLS is not applicable to the problem addressed in this work. Translation validation systems designed for HLS cannot be directly applied to our problem for at least two reasons. First, translation validation requires a method to translate each compiler IR into symbolic expressions, so systems based on other IRs cannot be applied directly. Second, the definition of equality in this work differs significantly from the definition of equality in prior work. In particular, we use dataflow graphs as intermediate representations in our application compiler, while prior works primarily use control flow graphs or custom representations. Each of these has different definitions of equality and therefore different validation infrastructure.

There have also been prior attempts at verifying the hardware and software of an application compiler, but these attempts either do not provide the same benefits as translation validation or do not provide formal guarantees. For example, [10] uses hardware/software co-verification to functionally verify an application as it gets mapped to hardware. However, it uses only simulation-based verification, not formal methods. ILA [18], [19], a formal software/hardware interface for accelerators, can be used to verify an instruction-level specification of an accelerator with respect to the hardware implementation. While ILA also can be used to do complex, system-level verification of a hardware/software system, its aims are different from those of this project. ILA can be used to formally verify that the specification of an instruction matches the hardware implementation, while this work aims to verify that each stage of an application compiler produces formally correct results. One noteworthy difference is that translation validation across every major compilation stage has the benefit of enabling much easier bug localization.

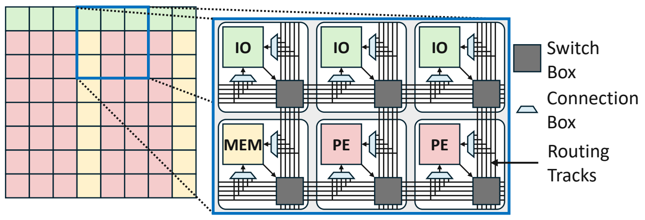


Fig. 1. Architecture of a CGRA with three types of tiles that communicate through a reconfigurable interconnect: IO tiles that pass data to and from the array, MEM tiles that buffer data, and PE tiles that do arithmetic operations.

III. Background

A. CGRA Architecture and Application Compiler

Our translation validation case study is carried out in the context of an open-source application compiler for coarse-grained reconfigurable array (CGRA)-based programmable accelerators [21], [1]. The implementation of our verification tool is also open-source [2]. We use an SMT-based model checker for verification. We briefly describe these components to provide context.

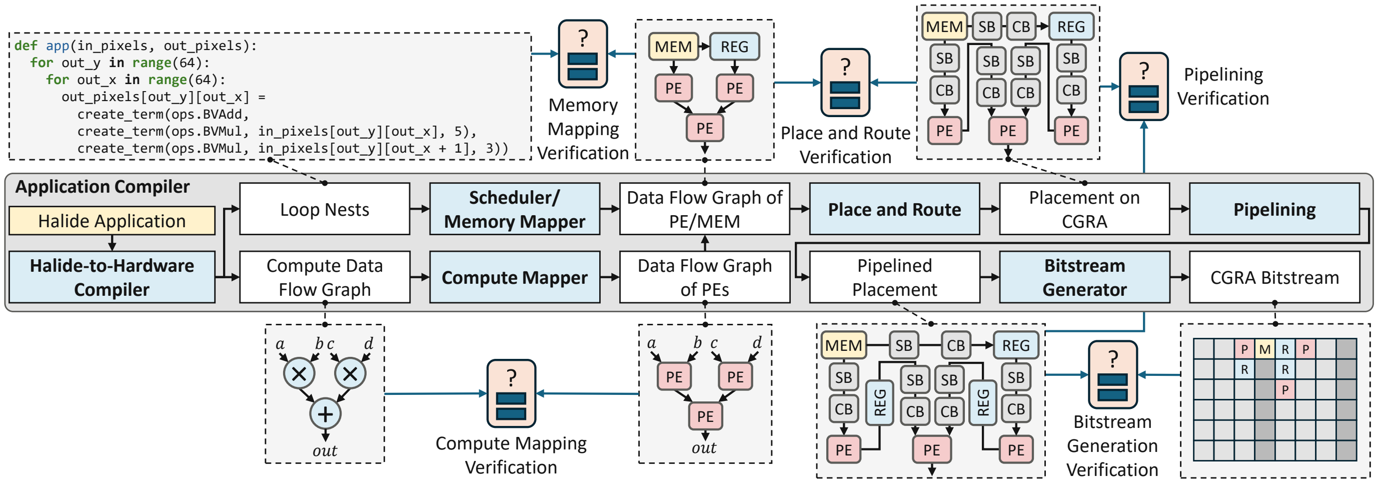
CGRAs are a class of programmable accelerators composed of a grid of tiles, as shown in Fig. 1. CGRAs have different types of tiles: processing element (PE) tiles, memory (MEM) tiles, and input/output (IO) tiles. PE tiles perform computations, MEM tiles buffer data, and IO tiles send data to and from the grid of tiles. These tiles communicate through a reconfigurable interconnect composed of horizontal and vertical routing tracks, switch boxes (SB), and connection boxes (CB). While many types of CGRA interconnects exist [28], the one we consider in this work is similar to FPGA interconnects in that it is statically configured. This means that, during the execution of the application, the configured routing connections act as wires. Any tile can send data to any other tile in a single cycle. Additionally, there are configurable pipelining registers in every switch box, allowing multi-cycle connections between tiles.

We use the custom CGRA application compiler illustrated in Fig. 2 and presented in detail in [28]. This compiler encompasses application specification, scheduling, compute mapping, memory mapping, place and route, pipelining, and bitstream generation [21].

B. SMT-based Model Checking

SMT is the problem of determining the satisfiability of a first-order logic formula with respect to a given theory. Introductions to SMT can be found in [6], [7]. In this paper, we assume a fixed background theory $\mathcal{T}$ and assume that all formulas are interpreted according to $\mathcal{T}$. In practice, we primarily rely on the theory of bit-vectors, which includes operators corresponding to primitives in our various IRs.

A *symbolic transition system* (STS) is a tuple $\mathcal{S} := \langle X, I, T \rangle$, where X is a finite set of state variables, $I(X)$ is an SMT formula denoting the initial states of the system, and $T(X, X')$ is an SMT formula expressing a transition relation. Here, X' is the set obtained by replacing each variable $x \in X$ with a variable x' of the same sort (x' is the variable for the



```
def app(in_pixels, out_pixels):
  for out_y in range(64):
    for out_x in range(64):
      out_pixels[out_y][out_x] =
        create_term(ops.BVAdd,
        create_term(ops.BVMul, in_pixels[out_y][out_x], 5),
        create_term(ops.BVMul, in_pixels[out_y][out_x + 1], 3))
```

Fig. 2. CGRA application compiler encompassing Halide compilation, compute mapping, scheduling and memory mapping, place and route, pipelining, and bitstream generation. The orange boxes represent the translation validation checks we introduced.

next state of x). It is often convenient to have multiple copies of the state variables X. We use $X@n$ to denote the set of variables obtained by replacing each variable $x \in X$ with a new variable called $x@n$ of the same sort.

Bounded model checking (BMC) is a bug-finding technique which attempts to find a counterexample of length k for a proposed invariant property, $P(X)$, of some STS [8]. A single BMC query at bound k for an invariant property uses an SMT solver to check the satisfiability of the following formula: $I(X@0) \wedge (\bigwedge_{i=0}^{k-1} T(X@i, X@(i+1))) \wedge \neg P(X@k)$. If the query is satisfiable, there is a bug.

IV. Symbolic Representations in SMT

To perform SMT-based translation validation for a specific compiler stage, both the input to and output from the compiler stage need to be representable as SMT terms or SMT-based symbolic transition systems. In this section, we describe our first contribution, which is to show how this can be done for each stage, from the high-level Halide specification down to the configured CGRA. We provide examples for each stage of the compiler in the appendix.

A. Application Specification

First, we need a representation of the application specification. This is written in Halide and compiled through the Halide-to-Hardware compiler [24] to generate the compute kernels in the application and the loop nests that specify how data is streamed through these kernels. These are consumed by different parts of the compiler, so we need symbolic representations for both.

Compute Kernels: CoreIR [12] is an LLVM-style hardware compiler and IR. The Halide-to-Hardware compiler uses it as the target for compute kernels, which contain dataflow graphs consisting only of arithmetic operations, e.g., add, sub, mul, shift, etc. When designing CoreIR, the semantics of these operations were based on the SMT-LIB standard [4]. One benefit of this decision is realized here, as it makes the translation of each node in the CoreIR graph to its symbolic representation straightforward. A compute kernel can be translated into a symbolic representation by traversing the dataflow graph and constructing the SMT term corresponding to each node. Edges in the dataflow graph are transformed into equality statements. An AST-like visualization of the symbolic representation of a simple sum-of-products compute kernel for a convolution application is shown in the bottom left corner of Fig. 2.

Loop Nests: We create a new code generation target for Halide to formally represent the loop nests. We generate Python code to create the formal representation. This has the benefit of enabling better metaprogramming and integration with other compiler infrastructure, but requires an SMT solver API in a high-level language. We use Pono [25], an SMT-based model checker which provides an API for creating and reasoning about symbolic transition systems. Pono is built on top of smt-switch [26], a tool that makes it easy to use a variety of different SMT solvers as back ends. An example of the generated code for the convolution application conv_1_2 is shown in the top left corner of Fig. 2.

B. Compute-Mapped Dataflow Graph of PEs

The compute mapper replaces the CoreIR operations in the compute kernels with PE tiles configured to perform the same operations. To represent the compute-mapped graph symbolically, we need a symbolic representation of the processing elements. As described in [21], the PEs in our target CGRA are specified in PEak [13], a domain-specific language for processing elements. PEak provides direct support for interpreting a program as an SMT-based STS. We leverage the formal interpretation of PEak to symbolically represent the compute-mapped dataflow graphs. We handle any state that may be present in the PEs by creating new state variables in the STS. An AST-like visualization of a symbolic representation of the compute-mapped dataflow graph is shown in the bottom middle of Fig. 2. Not shown is that each "PE" block is configured to perform an arithmetic operation. These blocks are translated to SMT terms using PEak's ability to provide formal models.

C. Fully-Mapped Dataflow Graph of PEs and MEMs

After applying the memory-mapping stage to the loop nests, we can construct a fully-mapped dataflow graph which contains both configured PE tiles and memory tiles. The memory tiles are specified in Lake [21], a Python-embedded language for specifying streaming memories. Unlike CoreIR and PEak, Lake has no direct support for generating SMT. Thus, to represent these memories in SMT, we first generate Verilog and then translate the Verilog to SMT-based symbolic transition systems using Yosys [34], an open-source synthesis tool that can consume Verilog and produce SMT.

To obtain the symbolic representation for the fully-mapped dataflow graph, we traverse each node of the graph in topological order, substituting in the corresponding SMT representation, either the one generated by PEak or the one from the Yosys SMT flow. A visualization of a fully-mapped dataflow graph is shown in the top middle of Fig. 2.

D. Place and Route and Pipelining

The next stage of the application compiler is place and route (PnR). The PnR tool adds new nodes to the dataflow graph for routing data between the tiles. The types of nodes introduced in PnR are SB nodes for switch boxes, CB nodes for connection boxes, RMUX nodes for register multiplexers, and REG nodes for pipelining registers.

Each of the PnR nodes has a straightforward symbolic representation as an SMT expression. The configurable interconnect is made up of multiplexers, and configured multiplexers are essentially just wires. Therefore, SB, CB, and RMUX nodes can be represented as equalities (relating the nodes at either end) in SMT. REG nodes just need to delay the input to the node by one clock cycle and can be implemented with a simple modification of the STS. The PnR graph is translated into SMT in the same manner as the fully-mapped dataflow graph.

The symbolic representation of the pipelined placement result is generated using the same method as the PnR graph.

E. CGRA Verilog

The final stage of application compilation is bitstream generation. To symbolically represent the bitstream-configured CGRA hardware, we must represent the CGRA Verilog as SMT terms. CGRAs, like FPGAs, have a reconfigurable interconnect which, in its unconfigured state, contains combinational loops. Yosys cannot translate hardware circuits with combinational loops. However, the compiler guarantees that in a correctly configured application, there will be no combinational loops. Therefore, we first apply the configuration to the CGRA RTL design before using Yosys.

The flow for generating an SMT-based STS representation of the configured CGRA is as follows: load the CGRA Verilog into Yosys, use Yosys to flatten and simplify the design, output the flattened design to a new Verilog file, replace all configuration registers within the flattened design with constant values based on the bitstream, load the modified Verilog back into Yosys, use Yosys to propagate the configuration register values and simplify as much as possible, and finally, use Yosys to generate an SMT-based STS. Note that configuration bitstreams are composed of address-data pairs, where the address uniquely identifies a configuration register in the design. We can thus automatically identify which registers (in the flattened Verilog) to replace with constant values by leveraging the compiler's bitstream generation infrastructure to analyze the address and map it to a specific Verilog register.

V. Translation Validation

After symbolically representing the application at each stage, we can construct the corresponding translation validation queries at each stage. Each query relies on two symbolic representations that may be different in structure. In this section, we describe how the queries are created for each stage.

For simplicity, we assign the following names to the translation validation checks, as shown in Fig. 2. *Compute mapping verification* is between the compute dataflow graph and the dataflow graph of PEs. *Memory mapping verification* is between loop nests and the dataflow graph of PEs and MEMs. *Place and route verification* is between the dataflow graph of PEs and MEMs and the PnR result. *Pipelining verification* is between the PnR result and the pipelined PnR result. *Bitstream generation verification* is between the pipelined PnR result and the CGRA bitstream.

A. Compute Mapping Verification

We verify that the compute dataflow graph and the compute-mapped dataflow graph of PEs are equivalent. Here, equivalence means that the graphs have the same outputs given the same inputs, with a timing delay that is discussed later in the section.

First, we identify $\boldsymbol{I}_c$, the set of inputs to the compute dataflow graph, and $\boldsymbol{O}_c$, the set of outputs from it. Every input i in $\boldsymbol{I}_c$ has a corresponding input i_m, representing the input to the mapped graph, and likewise every output o in $\boldsymbol{O}_c$ has a corresponding output o_m, representing the output from the mapped graph.

We check equivalence using bounded model checking with an input constraint. The input constraint gets added to the BMC formula at each unroll. The input constraint is simply: $\forall\, i \in \boldsymbol{I}_c.\, i = i_m$. The property P to check is: $\exists\, o \in \boldsymbol{O}_c.\, o \neq o_m$. If this check is satisfiable, then our two dataflow graphs are not equivalent and there is a bug. Although the compute dataflow graph has no state, the compute-mapped dataflow graph does, as PEs may have pipelining registers. The STS for the mapped dataflow graph uses state variables to model the sequential behavior of the PE hardware. We do an analysis of the mapped dataflow graph to determine how many cycles are needed to propagate from the inputs to the outputs and use that bound in our BMC check.

B. Memory Mapping Verification

We verify that the loop nests compiled from Halide are equivalent to the mapped dataflow graph of PEs and memory tiles. Equivalence means that, given an input array of pixels that are fed to the loop nests, if you stream the pixels into the dataflow graph of PEs and memory tiles, the stream of pixels produced some time later is equivalent to the output array of pixels from the loop nests.

Compared to the compute mapping verification described in Section V-A, memory mapping verification is more challenging because the two sides of the validation check are structurally very different. For the loop nests, the input is a multidimensional array of pixels and the computation to get the output pixels is untimed. For the mapped dataflow graph of PEs and MEMs, the input is a stream of pixels in time, where the dataflow graph expects a different pixel every cycle, and the computation to produce the output stream has memories and pipelining registers.

To address this challenge, we add constraints to the STS to ensure that the inputs of the two sides of the verification check are equivalent. We flatten the multidimensional array of input pixels to the loop nests into a one-dimensional array. The flattening is based on the order in which the pixels are sent to the mapped data flow graph, which, importantly, is explicitly specified in the Halide application schedule. Next, we create a look-up table (LUT) within the STS that uses the array index as the LUT index and the symbol representing the pixel as the value. After creating this LUT, we constrain the input of the dataflow graph to be equal to the output of the LUT, using the current cycle as the input index. This models the simple behavior of sending one input pixel into an input of the dataflow graph each cycle.

If the pattern of input pixels is different, a more complex constraint is needed. For example, within our compiler, we have the ability to unroll each input stream into the dataflow graph. If we unroll a mapped dataflow graph input by 2, on cycle 0, pixel 0 will be sent to dataflow graph input 0, and pixel 1 will be sent to dataflow graph input 1; then, on cycle 1, pixel 2 will be sent to dataflow graph input 0, and pixel 3 will be sent to dataflow graph input 1, and so on.

Let $\boldsymbol{I}_h$, $\boldsymbol{O}_h$ be the sets of multidimensional arrays corresponding to the inputs and outputs of the loop nests. Each array in $\boldsymbol{I}_h$ has u corresponding symbols in the mapped dataflow graph, where u is the input unrolling factor. If each array in $\boldsymbol{I}_h$ is i, then the u corresponding symbols in the mapped dataflow graph are $i_{m,0}, i_{m,1}, \ldots, i_{m,u-1}$.

Similarly, each array in $\boldsymbol{O}_h$ has v corresponding symbols in the mapped dataflow graph, where v is the output unrolling factor. If each array in $\boldsymbol{O}_h$ is o, then the v corresponding symbols in the mapped dataflow graph are $o_{m,0}, o_{m,1}, \ldots, o_{m,v-1}$. The symbols $i_{m,j}$ and $o_{m,j}$ are bit-vectors representing the input and output signals of the mapped application.

The input constraint over the loop nest inputs from $\boldsymbol{I}_h$ is $\forall\, j \in [0, u).\, \forall\, i \in \boldsymbol{I}_h.\, i[c \cdot u + j] = i^c_{m,j}$, where u is the input unrolling factor, and c is the current cycle.

After constraining the inputs for this verification check, we can create the output verification term. We similarly flatten the loop nest's output multidimensional array into a one-dimensional array and create a LUT that uses the current cycle as the index and the pixel at that index as the output. The property that we check is: $\forall\, j \in [0, v).\, \exists\, o \in \boldsymbol{O}_h.\, o[c \cdot v + j] \neq o^{c+l}_{m,j}$, where l is the cycle offset determined during application scheduling.

Because applications may have pipelining registers, PEs with registers, and memories, output pixels do not immediately get produced, but get produced with a certain delay. Our key insight is that for programmable accelerators that are fully statically scheduled like the CGRAs we target, this delay is determined at compile time based on the scheduling decisions made by the compiler, and we can leverage this information from the compiler to aid translation validation.

C. Place and Route Verification

In the previous subsection, the inputs and outputs of the application on either side of the compiler stage were structurally different, while in this stage they are similar. $\boldsymbol{I}_m$ and $\boldsymbol{O}_m$ are again sets of bit-vector symbols that are inputs and outputs of the dataflow graph of PEs and MEMs. Each i and o in $\boldsymbol{I}_m$ and $\boldsymbol{O}_m$ have corresponding inputs and outputs in the PnR result, which we denote i_p and o_p, respectively.

Our input constraint is $\forall\, i \in \boldsymbol{I}_m.\, i^c = i^c_p$, and the output property is $\exists\, o \in \boldsymbol{O}_m.\, o^c \neq o^{c+l}_p$, where again c is the current cycle, and l is the offset in cycles between the valid output pixels from the mapped dataflow graph of PEs and MEMs and the valid output pixels from the PnR dataflow graph. Again, this offset is determined during the compilation of the application. The delay comes from hardware elements like IO tiles that have pipelining registers. These tiles get added into the application during PnR, and the delay is accounted for in the application compiler.

D. Pipelining Verification

The PnR and pipelined PnR results are also structurally similar. The input constraint and output property are therefore the same as in Section V-C, with new variables for the inputs and outputs of the pipelined version of the application. During pipelining, the compiler adds pipelining registers to the PnR result. This changes the cycle count for when outputs are produced. These additional pipelining registers are accounted for during the rescheduling stage of pipelining, and we use that scheduling information to calculate any additional timing information needed.

E. Bitstream Generation Verification

In the final stage, we have symbolic representations that are structurally very different: the pipelined PnR result is a dataflow graph translated into a symbolic representation, and the bitstream-configured CGRA is the hardware of the accelerator translated into a symbolic representation. However, the inputs and outputs of the two representations are very similar. Thus, we can again use the input constraint and output property from Section V-C, with new variables for the inputs and outputs of the configured RTL design.

Our verification framework relies on several assumptions. Most importantly, it assumes that the Halide compiler, Yosys, and the SMT solver produce correct results. It also assumes that the translations from compiler IR to SMT described in

Section IV are correct. Given these assumptions, our approach will find any bug in the application compiler that manifests in incorrect outputs and can identify the stage of the compiler that introduced that bug. Our approach verifies that the entire application executes correctly, provided we use a BMC bound sufficiently large to ensure that all outputs are produced.

VI. Symbolic Starting States with Automatic Constraint Generation

To verify a full application with the queries described in the previous section, we need to run BMC for thousands of cycles. For example, an application that consumes 1 pixel per cycle and produces 1 output pixel per cycle needs to be run for at least 4096 cycles for an input/output image tile size of 64×64.

This requirement only holds for applications that start from a reset state. Resetting all state ensures that we start the verification process in a valid state. To improve performance, we enable BMC to start from a symbolic starting state, allowing verification from any point in the application's execution. This allows for parallelization by running multiple BMC instances from different starting points.

With symbolic starting states, any state in the design can start the verification check with any value. This allows the hardware to start in invalid states, and will cause memory elements that need synchronization, like the address generators in the memory tiles, to be out of sync. The address generators present within the statically scheduled memories need several constraints to work properly. First, they have a cycle counter that is reset to 0 and counts up every cycle for synchronization purposes. In a symbolic starting state, we want these registers to start at a symbolic value, and we want that value to be identical across tiles. We add a constraint that says that all of the cycle counters start at the same symbolic value. Additionally, there are address and schedule generator counters that consume the configuration of the memory tile and count in specific patterns according to the unified buffer abstraction described in [24]. For a given cycle within the application execution, the address counter, schedule counter, and dimension counter within the memory tile have one valid value. This value can be determined before the BMC property is created, and we can constrain the counters appropriately.

Given the set of state variables that are minimally required to constrain a valid starting state, constraint generation is automated and can adapt to any hardware design. In this domain, we can generate complete sets of constraints at each timestep using compiler-derived schedules. Our use of these schedules is not black-box: we rely on the precise, static knowledge of dataflow and execution timing these tools provide to extract exact invariants. The resulting symbolic constraints successfully rule out unreachable states and prevent false counterexamples in all of our evaluated applications.

This idea is similar to automatically generated inductive invariants [9], [35]. This technique allows for parallelization of the BMC check; its effects are demonstrated in the experimental evaluation section. Additionally, starting in a symbolic starting state enables more thorough verification. If any unconstrained register in the design has a value that could introduce a bug into the application, symbolic starting states ensure that we find that value. If any value in any register or memory could cause a data dependency bug, symbolic starting states would find that bug as well.

TABLE I
Benchmarks with number of PEs, MEMs, and output latency.

Small Test	#PE	#MEM	Latency
multiply	1	1	1
arithmetic	4	1	4
absolute	4	1	4
boolean	5	1	3
equal	4	1	3
ternary	3	1	3
compare	8	1	5
ucomp	8	1	5
minmax	4	1	2
uminmax	4	1	2
shift	4	1	4
ushift	3	1	3

Application	#PE	#MEM	Latency
conv_1_2	3	1	3
conv_2_1	2	2	65
conv_3_3	9	2	138
fast corner	20	2	208
pyramid	15	4	464
gaussian	12	2	141
harris	63	6	415
unsharp	57	12	429

VII. Experimental Evaluation

We evaluate the impact of the performance improvement from Section VI, the runtime of each verification check, and coverage of bugs in the application compiler. For these experiments we use the applications listed in Table I. The 12 applications on the left are small tests, while the applications on the right are larger workloads from [33]. For each application, the number of PEs and MEMs represent the number of resources utilized by each benchmark. We use input image tiles of size 64×64 with 16 bit values and map every application to a fixed CGRA architecture with 16 rows and 32 columns of tiles. We use the model checker Pono [25] with the Boolector backend [30]. We run every BMC check using 32 cores of a 2.5 GHz Intel CPU with 252 GB of memory.

A. Symbolic Starting States

The symbolic starting state optimization described in Section VI impacts all of the stages of verification except for compute mapping verification, as that does not have any memory tiles and very little state. This optimization enables parallel processing of the verification check, as we can start many different threads at different starting points.

Without symbolic starting states, memory mapping, place and route, pipelining, and bitstream verification cannot finish even with the simplest application. The memory cost of the verification check exceeds the total system memory of 252 GB. By using symbolic starting states to parallelize the BMC check, we can verify full applications. This optimization helps mitigate the state explosion associated with unrolling the BMC check for thousands of cycles.

B. Runtime of Translation Validation

We evaluate the runtime of each of the translation validation checks on applications that do not have any bugs. For each check, we report the runtime of the application compiler without verification, the construction of the symbolic representation, and the SMT solving.

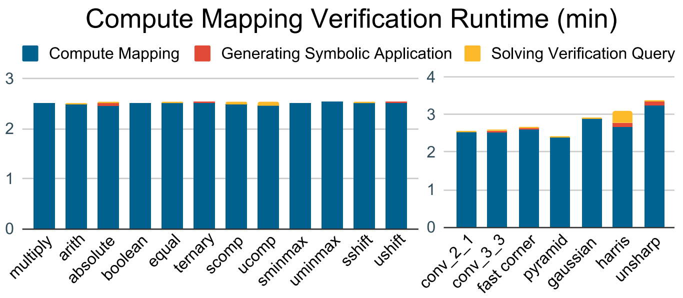


Fig. 3. Runtime of compute mapping verification, broken down into application compiler runtime without any verification, time for constructing the symbolic representation, and SMT solving time.

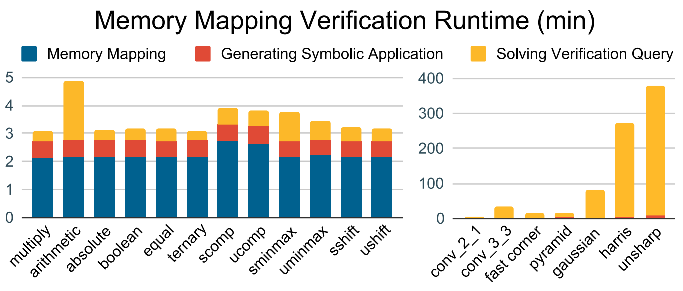


Fig. 4. Runtime of memory mapping verification.

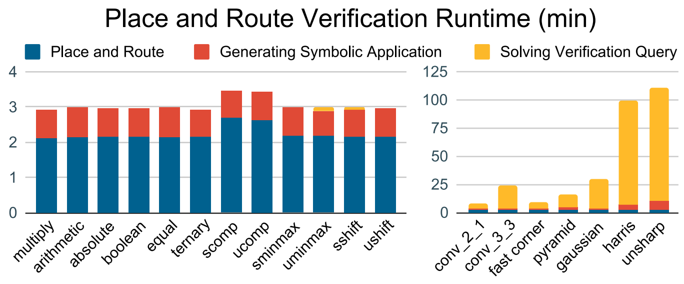


Fig. 5. Runtime of place and route verification.

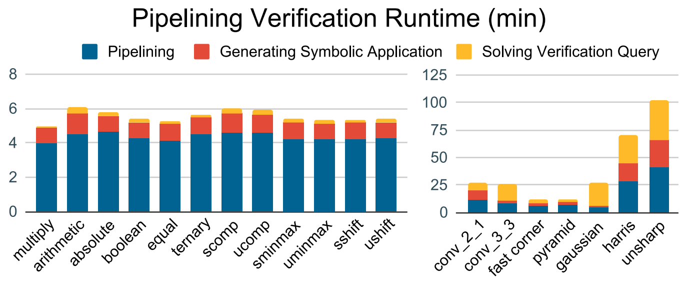


Fig. 6. Runtime of pipelining verification.

1) Compute Mapping Verification: The runtime of compute mapping verification is shown in Fig 3. Overall, as both the compute dataflow graph and mapped dataflow graph of PEs do not buffer data in memory elements, the verification check is very fast, as we do not need to run the BMC algorithm for many steps. The representation of both the input to and the output from the compute mapper can be translated into SMT very quickly, as we can leverage the PEak interpretation of each node. The runtime of compute mapping verification ranges from 1 to 20 seconds.

2) Memory Mapping Verification: The runtime of memory mapping verification is shown in Fig. 4. As discussed in Section V-B, the two sides of this validation check are structurally very different. Therefore, the validation check is much slower than compute mapping verification, particularly for the larger application-level tests. This verification check requires running BMC for the number of cycles required to produce every output pixel.

In contrast to compute mapping verification, this verification stage is very complex, both due to the highly temporal nature of this stage and the representation of the hardware. Memory tiles are more complex than PE tiles, requiring finite state machines and counters for implementing memory addressing and access patterns. Representing memory elements in SMT using the flow described in Section IV-C results in complex representations that take more time to verify.

Overall, for smaller applications, the overhead of the verification ranges from 1 to 3 minutes, and for larger applications it ranges from 2 to 375 minutes.

3) Place and Route Verification: Place and route verification runtime is shown in Fig. 5. The source and target for this translation validation query are more similar than those associated with memory mapping. Therefore, the verification runtime is shorter. For the small tests, running the application compiler and generating the symbolic representations take more time than the verification check itself. The overhead of this stage of verification ranges from about 1 minute for small applications to anywhere from 1 minute to 108 minutes for larger applications.

4) Pipelining Verification: Pipelining verification runtime is shown in Fig. 6. Like place and route verification, the representations of the application before and after pipelining are similar in structure. Pipelining will add pipelining registers to the application and reschedule the memory tiles, changing their configurations. Pipelining is a longer stage in the application compiler as it requires running PnR multiple times to find solutions with short critical paths. The overhead of pipelining verification is 1 minute for small applications, and ranges from 1 minute to 61 minutes for large applications.

5) Bitstream Generation Verification: Finally, the runtime of bitstream verification is shown in Fig. 7. The runtime of this verification stage is dominated by generating the symbolic application rather than the application compiler or the SMT solver. This is due to the procedure described in Section IV-E. Generating and simplifying the flattened Verilog for the accelerator using Yosys takes a long time, especially for larger applications. Larger applications also require a larger CGRA, which dramatically increases the runtime of Yosys. Overall, bitstream generation verification introduces an overhead of 8 to 9 minutes for small applications and 9 to 159 minutes for large applications.

C. Bug Coverage

1) Introducing Bugs into the Compiler: We report the runtime of finding various bugs in "fast corner," a moderately complex application in our benchmark suite. First, we intro-

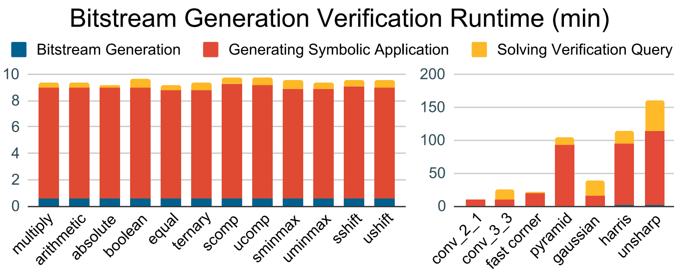


Fig. 7. Runtime of bitstream generation verification.

duce a bug in the compute mapping stage. We modify the mapping result to map a less-than operation to a PE that uses a less-than-or-equal-to operation. This bug is found 49 seconds. Next, we introduce a bug in the PE hardware where, in a three-input addition operation, the carry-in bit to the second adder is stuck at 0. This bug is found 35 seconds. Finally, we introduce a bug into the compiler where it assumes that each PE has a delay of 0 cycles when the PE takes 1 to produce an output. This bug is found 81 seconds. This stage of the compiler without verification takes 3 minutes.

Next, we discuss the bugs that are found during memory mapping verification. First, we introduce a bug in scheduling, adjusting the static schedule of a memory tile to start counting 1 cycle early. This bug is found in 3 minutes and 12 seconds. Next, we introduce a hardware bug into the memory tiles. This bug shrinks the bit width of one of the address generation counters from 5 bits to 4 bits, leading to the counter overflowing. This bug is found in 3 minutes. This stage of the application compiler without verification takes 2 minutes.

Next, we examine bugs that are found during place and route verification. First, we introduce a bug where a PE is connected to another PE through a connection that is supposed to be combinational communication but instead has a pipelining register, so it takes one cycle. This bug is found 5 minutes and 50 seconds. Next, we introduce a hardware bug into the interconnect where the configuration of a connection box is faulty and routes data to the wrong track. This bug is found 5 minutes and 55 seconds. This stage of the application compiler without verification takes 2 minutes.

Next, we examine the types of bugs that are found during pipelining verification. First, we introduce a bug where a pipelining register is added to the place and route result, but not accounted for in the scheduling of the application. This bug is found 7 minutes and 11 seconds. Next, we introduce a bug into the hardware where a single pipelining register is misconfigured and is bypassed when it should not be. This bug is found 7 minutes and 6 seconds. This stage of the application compiler without verification takes 6.5 minutes.

Finally, we examine the bugs that are found during bitstream generation verification. First, we add a bug to the compiler where a single bit of the bitstream is flipped. This bug is found in 171 minutes. Next, we introduce a bug in the accelerator hardware that shrinks the bit width of a single configuration register in the interconnect from 16 to 15 bits. This bug is found in 19 minutes. This stage of the application compiler without verification takes 1 minute.

Without our verification system, these bugs would only be found during the Verilog simulation of the entire application. Our approach leads to much more thorough verification of the application compiler than traditional simulation. Bugs that are exercised by specific input values can easily be missed in simulation but will not be missed in our approach. Compiler passes that introduce bugs that are later masked can be missed with end-to-end simulation tests, while our approach finds bugs introduced by any stage. Simulation requires starting in a reset state, while our symbolic starting states with automated state constraints can start verification at any point in the application execution, so bugs that show up only late in the application execution may be found much faster with our approach.

2) Bugs Found in the Application Compiler: We discovered several real bugs in our application compiler that were previously unknown. Our application compiler has been developed for over six years [21] and until now relied on traditional simulation-based verification methods.

The first bug that we uncovered was in the PEs. There is an operation in the PE that is a multiply that takes in two 16-bit operands and returns the middle 16 bits of the 32-bit result. The bug that was found was a mismatch between the specification of the operation and the implementation within the PE. The operation was specified as two 16-bit operands that were zero padded to 32 bits, multiplied, and shifted right by 8 bits. The bottom 16 bits of the result were to be returned. This specification did not match the hardware; instead, the first step should have been to sign extend the two 16-bit operands to 32 bits. During compute mapping, this bug was found in 1.5 minutes.

The second bug was a hardware bug related to configuration register placement. The memory tiles in the CGRA can be configured to act as ROMs. In ROM mode, there is a write enable signal in the memory tile that should be unused during application execution. However, this signal was left enabled, and a configuration register present in a connection box was used to make sure no signal was routed to this input. This was a temporary fix that was forgotten about and never permanently addressed. In the memory mapping verification stage, this bug was found in 4 minutes and 12 seconds.

Finally, we discovered several scheduling bugs related to pipelining. When rescheduling an application after pipelining, any latency that is not accounted for may cause incorrect outputs. Identifying when the pipelining stage of the compiler causes the Verilog simulation to fail is difficult and often requires a significant amount of debugging. Our validation check found 5 rescheduling bugs in the rescheduling stage of pipelining in seconds for smaller applications.

VIII. Conclusion

In this work, we presented a methodology for automated translation validation of an application compiler for statically scheduled accelerators. We introduced an SMT-based formal representation for every stage in the compiler, a way to perform translation validation of each of the stages leveraging scheduling information in the compiler, and a technique for dramatically improving performance through symbolic starting states. This is the first work that we are aware of to apply translation validation for a complex accelerator application compiler that combines both hardware and software representations. Our work enables rigorous verification and much better bug localization than traditional simulation-based approaches.

IX. Acknowledgments

This work was supported by funding from the SRC JUMP 2.0 PRISM Center, an NSF CAREER Award (number: 2238006), a research gift from Cisco, an Apple Stanford EE PhD Fellowship in Integrated Systems, and the Stanford Agile Hardware Center.

References

[1] AHA Application Compiler. https://github.com/StanfordAHA/aha. Accessed: 2025-03-01.

[2] Verification Implementation. https://github.com/jack-melchert/verified_agile_hardware. Accessed: 2025-03-01.

[3] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A Translation Validator for Optimizing Compilers. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 291–295. Springer Berlin Heidelberg, 2005.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[5] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. *Handbook of Model Checking*, pages 305–343, 2018.

[6] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, 2018.

[7] Clark Barrett, Cesare Tinelli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, and Yoni Zohar. Satisfiability Modulo Theories: A Beginner's Tutorial. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Proceedings of the 26th International Symposium on Formal Methods (FM '24), Part II*, volume 14934 of *Lecture Notes in Computer Science*, pages 571–596. Springer Nature Switzerland, September 2024. Milan, Italy.

[8] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[9] Satrajit Chatterjee and Michael Kishinevsky. Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 321–338, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] Shuo-Hung Chen, Hsiao-Mei Lin, Hsin-Wen Wei, Yi-Cheng Chen, Chih-Tsun Huang, and Yeh-Ching Chung. Hardware/Software Co-designed Accelerator for Vector Graphics Applications. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 108–114, 2011.

[11] Basile Clément and Albert Cohen. End-to-End Translation Validation for the Halide Language. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022.

[12] Ross Daly, Leonard Truong, and Pat Hanrahan. Invoking and Linking Generators from Multiple Hardware Languages using CoreIR. In *Workshop on Open-Source EDA Technology (WOSET)*, 2018.

[13] Caleb Donovick, Jackson Melchert, Ross Daly, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. PEak: A Single Source of Truth for Hardware Design and Verification. *ACM Transactions on Embedded Computing Systems*, November 2024.

[14] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–564, 2022.

[15] Evguenii I Goldberg, Mukul R Prasad, and Robert K Brayton. Using SAT for Combinational Equivalence Checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 114–121. IEEE, 2001.

[16] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. EPIMap: Using Epimorphism to Map Applications on CGRAs. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1284–1291, New York, NY, USA, 2012. Association for Computing Machinery.

[17] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. Formal Verification of High-Level Synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[18] Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. INVITED: Generalizing the ISA to the ILA: A Software/Hardware Interface for Accelerator-rich Platforms. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4, 2023.

[19] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.*, 24(1), December 2018.

[20] Chandan Karfa, Dipankar Sarkar, Chittaranjan Mandal, and Pramod Kumar. An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):556–569, 2008.

[21] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Trans. Embed. Comput. Syst.*, 22(2), January 2023.

[22] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Validating High-Level Synthesis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 459–472. Springer Berlin Heidelberg, 2008.

[23] Alan Leung, Dimitar Bounov, and Sorin Lerner. C-to-Verilog Translation Validation. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 42–47, 2015.

[24] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. *ACM Trans. Archit. Code Optim.*, 20(2), March 2023.

[25] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark W. Barrett. Pono: A Flexible and Extensible SMT-Based Model Checker. In *International Conference on Computer Aided Verification*, 2021.

[26] Makai Mann, Amalee Wilson, Yoni Zohar, Lindsey Stuntz, Cesare Tinelli, and Clark Barrett. Smt-Switch: A Generic C++ API for SMT Solving. https://github.com/stanford-centaur/smt-switch, 2024.

[27] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, pages 166–173, 2002.

[28] Jackson Melchert, Yuchen Mei, Kalhan Koul, Qiaoyi Liu, Mark Horowitz, and Priyanka Raina. Cascade: An Application Pipelining Toolkit for Coarse-Grained Reconfigurable Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2024.

[29] George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 83–94. Association for Computing Machinery, 2000.

[30] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In Hana Chockler and Georg Weissenbacher,

editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

[31] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A General Constraint-Centric Scheduling Framework for Spatial Architectures. *SIGPLAN Not.*, 48(6):495–506, June 2013.

[32] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer Berlin Heidelberg, 1998.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530. Association for Computing Machinery, 2013.

[34] Claire Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.

[35] Jiahui Xu and Lana Josipović. Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[36] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 1041–1054. IEEE Press, 2021.

[37] Zhongyuan Zhao, Weiguang Sheng, Qin Wang, Wenzhi Yin, Pengfei Ye, Jinchao Li, and Zhigang Mao. Towards Higher Performance and Robust Compilation for CGRA Modulo Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2201–2219, 2020.

[38] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design*, 27(3):335–360, November 2005.

APPENDIX

Our 20 applications generate symbolic representations whose sizes range from a few lines to roughly 200,000 lines. Here, we show excerpts from the simplest application, conv_1_2. The representations for this simple application range from a few lines for the compute kernel to over 4000 lines for the final CGRA Verilog.

A. Compute Kernels

```
(bvadd
 (bvadd $e3
  (bvmul #h0005 $e2))
 (bvmul #h0003 $e1))
```

The single compute kernel in conv_1_2 has three inputs and computes the following: $(e3 + (5 * e2)) + (3 * e1)$, where 5 and 3 are the kernel weights in this application.

B. Loop Nests

This is an excerpt from a 61-line representation:

```
(let
  (($e0
    (bvadd
      (bvmul in_0_0 #h0005)
      (bvmul in_1_0 #h0003))))
  (let
    (($e1
      (bvadd
        (bvmul in_1_0 #h0005)
        (bvmul in_2_0 #h0003))))
    (let
      (($e2
        (bvadd
          (bvmul in_2_0 #h0005)
          (bvmul in_3_0 #h0003))))
      (let
        (($e3
          (bvadd
            (bvmul in_0_1 #h0005)
            (bvmul in_1_1 #h0003))))
...
```

Here, we assign each output variable a value based on the addition of the multiplication of two input pixels with kernel values.

C. Compute-Mapped Dataflow Graph of PEs

```
((_ extract 15 0)
  (bvadd
    (concat #b0
    ((_ extract 15 0)
      (bvadd
      (concat #b0
        ((_ extract 15 0)
        (bvmul #h00000005
          (concat #h0000 $e2))))
      (concat #b0 $e3))))
    (concat #b0
    ((_ extract 15 0)
      (bvmul #h00000003
      (concat
        (ite
        (= #b1
          ((_ extract 47 47) IVAR_V_415@0))
                #hfff #h000) $e1))))))
```

In this stage, the representation is generated by taking the dataflow graph of configured PEs and simplifying it to remove all unused hardware. In this application, we have two PEs performing multiply-add operations on 16-bit operands with a 32-bit multiplier.

D. Fully-Mapped Dataflow Graph of PEs and MEMs

This is an excerpt from a 1284-line representation:

```
...
(bvand
  (bvand
    (bvand clk_en$const0.next
      (bvand clk_en$const0
        (ite
          (and
            (= V_0$const0 V_0$const0_inter.next)
            (= bmc_counter.next
              (bvadd bmc_counter #h0001))
            (= V_0$const0.next V_0$const0_inter)
            (= V_1$const0 V_1$const0_inter.next)
            (= V_1$const0.next V_1$const0_inter)
            (= V_2$const0 V_2$const0_inter.next)
            (= V_2$const0.next V_2$const0_inter)
            (= V_3$const0 V_3$const0_inter.next)
            (= V_3$const0.next V_3$const0_inter)
            (= V_4$const0 V_4$const0_inter.next)
            (= V_4$const0.next V_4$const0_inter)
            (= V_5$const0 V_5$const0_inter.next)
            (= V_5$const0.next V_5$const0_inter))
              #b1 #b0)))
    (ite
      (= state5.next $e2) #b1 #b0))
  (ite
    (= state7.next
      (ite
        (= #b1 flush_mem) #h0000
        (bvadd $e2
          (concat #b000000000000000 $e6)))) #b1 #b0))
...
```

This stage of the compiler has a much more complex representation. This is an excerpt showing some pipelining logic within the PEs and some flushing logic in the memory tile.

E. Place and Route and Pipelining

This is an excerpt from a 1265-line representation:

```
...
(bvand
 (bvand
  (bvand
   (bvand
    (ite
     (=
      ((_ extract 15 0)
       (bvadd
        (concat #b0 V_18_p2)
        (concat #b0
         ((_ extract 15 0)
          (bvmul #h00000003
           (concat
            (ite
             (= #b1
              ((_ extract 15 15) V_16_p2))
               #hffff #h0000) V_16_p2))))))
               r1.reg_in.next) #b1 #b0))
   (ite
    (= r1.reg_in r1.reg_val.next) #b1 #b0))
  (ite
   (= r1.reg_val out.I0) #b1 #b0))
 (ite
  (= r1.reg_val.next out.I0.next) #b1 #b0))
...
```

This excerpt shows some PE calculations and the next state logic for some interconnect registers. These registers get added during the PnR and pipelining stages of the compiler.

F. CGRA Verilog

This is an excerpt from a 4598-line representation:

```
...
(let
  (($e9
    (ite
      (= #b1 $e1) state111 state110)))
  (let
    (($e10
      (ite
        (= #b1 $e2) $e9 #h0000)))
    (let
      (($e11
        (ite
          (= #b1 $e1) state119 state118)))
      (let
        (($e12
          (select |\$flatten\Tile_X01_Y03.
            \CB_PE_input_width_16_num_1.
            \CB_PE_input_width_16_num_1.
            \u_precoder.$auto$proc_rom.cc
            :155:do_switch$1345| #b01100)))
...
```

This final stage of the application is the most complex, even for this simple application. During this stage, we generate the representation from the Verilog of the accelerator itself, so this excerpt shows some of the complexity that gets added to the symbolic representation as a result.