FPGA Technology Mapping Using Sketch-Guided Program Synthesis

Gus Henry Smith
University of Washington
Seattle, USA
gussmith@cs.washington.edu

Benjamin Kushigian
University of Washington
Seattle, USA
benku@cs.washington.edu

Vishal Canumalla
University of Washington
Seattle, USA
vishalc@cs.washington.edu

Andrew Cheung
University of Washington
Seattle, USA
acheung8@cs.washington.edu

Steven Lyubomirsky
OctoAI
Seattle, USA
slyubomirsky@octo.ai

Sorawee Porncharoenwase
University of Washington
Seattle, USA
sorawee@cs.washington.edu

René Just
University of Washington
Seattle, USA
rjust@cs.washington.edu

Gilbert Louis Bernstein
University of Washington
Seattle, USA
gilbo@cs.washington.edu

Zachary Tatlock
University of Washington
Seattle, USA
ztatlock@cs.washington.edu

Abstract

FPGA technology mapping is the process of implementing a hardware design expressed in high-level HDL (hardware design language) code using the low-level, architecture-specific primitives of the target FPGA. As FPGAs become increasingly heterogeneous, achieving high performance requires hardware synthesis tools that better support mapping to complex, highly configurable primitives like digital signal processors (DSPs). Current tools support DSP mapping via handwritten special-case mapping rules, which are laborious to write, error-prone, and often overlook mapping opportunities. We introduce Lakeroad, a principled approach to technology mapping via sketch-guided program synthesis. Lakeroad leverages two techniques—architecture-independent sketch templates and semantics extraction from HDL—to provide extensible technology mapping with stronger correctness guarantees and higher coverage of mapping opportunities than state-of-the-art tools. Across representative microbenchmarks, Lakeroad produces 2–3.5× the number of optimal mappings compared to proprietary state-of-the-art tools and 6–44× the number of optimal mappings compared to popular open-source tools, while also providing correctness guarantees not given by any other tool.

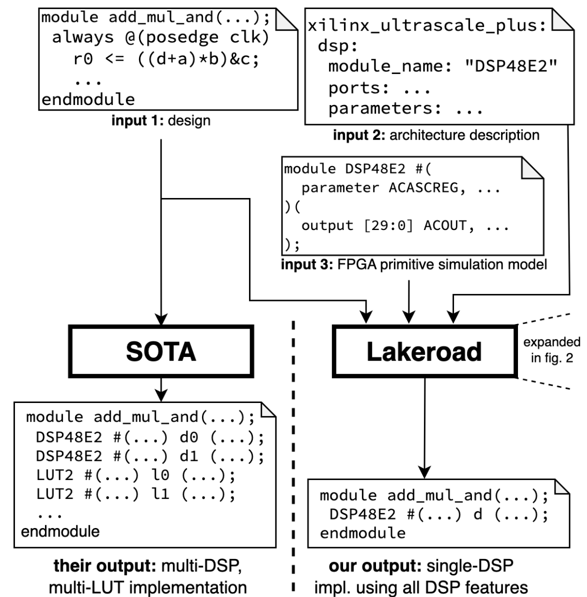


Figure 1. Even given a simple input design (input 1), the state-of-the-art (SOTA) hardware synthesis tool for Xilinx FPGAs frequently fails to efficiently use programmable primitives like DSPs. Lakeroad, on the other hand, can utilize all features of programmable primitives given just a short description of an FPGA architecture (input 2) and the vendor-provided simulation models of the primitive (input 3).

ACM Reference Format:
Gus Henry Smith, Benjamin Kushigian, Vishal Canumalla, Andrew Cheung, Steven Lyubomirsky, Sorawee Porncharoenwase, René Just, Gilbert Louis Bernstein, and Zachary Tatlock. 2024. FPGA Technology Mapping Using Sketch-Guided Program Synthesis. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640387

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0385-0/24/04.
https://doi.org/10.1145/3620665.3640387

1 Introduction

Given a high-level hardware design specification (e.g., expressed in behavioral Verilog), FPGA technology mappers search for an equivalent low-level implementation in terms of the target FPGA's primitives. See Figure 1 for an example, where the high-level, behavioral `add_mul_and` module ("input 1") is converted into FPGA-specific implementations ("their output" and "our output") using Xilinx-specific `DSP48E2` and `LUT2` primitives.

Historically, FPGAs consisted of relatively simple primitives, such as lookup tables (LUTs) and carry chains. Tools like ABC [14, 30, 37] *automatically* map to these basic primitives by translating designs to a library of simple logic gates and then packing those gates into LUTs.

However, FPGAs are becoming increasingly heterogeneous via the inclusion of specialized and diverse primitives such as digital signal processors (DSPs). Utilizing these specialized primitives effectively is now crucial for achieving high performance [49]. These specialized primitives make FPGA technology mapping far more challenging since technology mappers must now explore a much larger search space while also satisfying each primitive's complex set of restrictions and dependencies. For example, Xilinx's DSP48E2 is a multifunction DSP with nearly 100 ports and parameters, whose numerous configurations enable support for a large variety of computations. The manual for the DSP48E2 alone is 75 pages long, where considerable text details the complex restrictions between the settings of the nearly 100 ports and parameters.

Existing technology mapping tools frequently fail to map designs to specialized primitives like DSPs, requiring manual work for the hardware designer to recover the performance of their design. While existing toolchains have the ability to automatically infer locations where specialized primitives can be used in large designs, inference often fails [1, 2, 4]. In these cases, the designer can either accept lower performance and higher resource utilization, or they can perform what we call *partial design mapping*. During partial design mapping, the designer manually identifies and separates out the module that should be mapped to a DSP. They can attempt to re-run technology mapping on that module alone, in the hopes that mapping succeeds. Yet existing toolchains often fail *even in the partial design mapping case:* Figure 1 shows a simple module `add_mul_and` which *should* fit on a single DSP48E2 according to the DSP's manual, but is instead mapped to multiple DSPs and LUTs by current state-of-the-art tools.[1] In the worst case, hardware designers are forced to manually instantiate complex primitives by hand, e.g., by looking through the 75-page DSP48E2 user manual to manually configure the DSP's dozens of ports and parameters.

[1]Licensing restrictions forbid naming the specific proprietary tools, but they are familiar, standard packages used by many hardware designers.

Current state-of-the-art technology mappers are implemented via ad hoc, handwritten pattern matching procedures, which fall short in three primary ways. First, as we saw above, they are **incomplete:** they miss many mapping opportunities, even across microbenchmarks based on vendor documentation. Second, they **do not provide strong correctness guarantees:** recent work highlights the significant number of bugs found across all major hardware synthesis tools [24]. Third, they are **difficult to extend:** *each* new complex primitive requires supporting detailed semantics and adding hundreds of new, special-case syntactic pattern matching rules [50].

This paper's key observation is that technology mapping is well-suited for the application of automated reasoning procedures—specifically, *program synthesis* [23]. We observe that the configuration space of a programmable FPGA primitive is essentially a small, bespoke programming language, and that program synthesis could be applied to automatically generate primitive configurations. We explore how program synthesis can simplify the design and implementation of FPGA technology mappers while providing **correct**, **extensible**, and **more complete** support for mapping to diverse, highly configurable primitives like DSPs. Program synthesis techniques rely on automated theorem provers like SAT and SMT solvers [8, 17] to automatically generate programs satisfying a given specification. We demonstrate how *sketch-guided program synthesis* [41] can be adapted for FPGA technology mapping, leveraging the Rosette [46] program synthesis framework.

Sketch-guided program synthesis requires encoding the *semantics* of the target language: in our case, a machine-readable, mathematical model specifying the behavior of each FPGA-specific primitive being mapped to. In a typical synthesis tool, which generates programs for a single target language, this is a one-time cost. However, in our setting, each new FPGA primitive introduces yet another new target language, which in turn requires extending the tool to encode yet another formal semantics.

To support correct, extensible, and more complete technology mapping, we propose automating this process with **semantics extraction from HDL**, adapted from past work [16], to automatically extract complete primitive semantics from vendor-published HDL models (Figure 1, "input 3"). Traditionally, such models have been used only for simulation and validation *after* technology mapping; we show that using the semantics to *implement* technology mapping with a program-synthesis-based approach yields substantially more complete FPGA technology mapping.

Sketch-guided program synthesis also requires *sketches*, which are partially complete programs with "holes" to be filled in by the solver. Sketches primarily serve to scale synthesis by constraining the set of programs that solvers explore when searching for one that satisfies the given specification, i.e., performance at the cost of completeness. In our

setting, sketches correspond to arrangements of primitives, using holes as placeholders for some of the primitives' ports and parameters. This could be a single DSP with holes for its ports and parameters (as in the example in Section 2.2), or a number of LUTs with holes for their LUT memories, or even a mixture of LUTs, DSPs, and carry chains. The synthesizer "fills in the holes" as necessary for the low-level FPGA-specific primitive to implement a given high-level behavioral design fragment. Unfortunately, developing effective sketches still requires synthesis expertise [11, 47]. Naïvely, our approach would also require new sketches for each new FPGA primitive we target.

To address these challenges, we introduce **architecture-independent sketch templates.** Hardware designs are often implemented using high-level blueprints that are similar across most FPGA architectures—sketch templates capture these blueprints and make them reusable across architectures. Therefore, by using sketch templates, we greatly reduce the overhead of supporting new architectures and diverse primitives. Typically, when adding support for a new primitive or FPGA architecture in Lakeroad, the hardware designer need not write or modify any sketch templates.

We leverage semantics extraction from HDL and architecture-independent sketch templates to build Lakeroad,[2] a new FPGA technology mapper based on program synthesis.

Lakeroad's prototype implementation automatically imports semantics for the LUTs, arithmetic carry chains, and DSPs of the Xilinx UltraScale+, Lattice ECP5, Intel Cyclone 10 LP, and SOFA [43] FPGA architectures. The only additional user input to Lakeroad is a short architecture description that lists the target FPGA's primitives (Figure 1, "input 2"). Architecture descriptions only need to be written once per architecture, and Lakeroad pre-supplies architecture descriptions for the aforementioned architectures. With the automatically extracted primitive semantics and the user-provided architecture description, we demonstrate that Lakeroad is more complete than proprietary tools on a variety of microbenchmarks that are representative of program fragments implemented with DSPs during partial design mapping. In particular, Lakeroad maps up to 3.5× more microbenchmarks than state-of-the-art tools for Xilinx, Lattice, and Intel, and up to 44× more microbenchmarks than Yosys.

This paper makes the following key contributions:

- The novel application of program synthesis to produce a technology mapper—Lakeroad—that is more **correct, complete,** and **extensible** than state-of-the-art tools.
- A technique for applying **semantics extraction from HDL** to automatically generate models of hardware usable by formal automated reasoning tools.
- The concept of **architecture-independent sketch templates,** which capture common patterns in hardware design in an architecture-independent way, plus **primitive interfaces** and **architecture descriptions**, the abstractions underlying these templates.
- A formalization of the Lakeroad toolchain and an argument for its correctness and sketch-completeness.
- The first notion of **technology mapping completeness** for FPGA compilers.
- **Empirical comparisons** of Lakeroad and existing hardware synthesis tools to evaluate both their relative completeness and ease of extensibility.

[2]Lakeroad is publicly available at https://github.com/uwsampl/lakeroad.

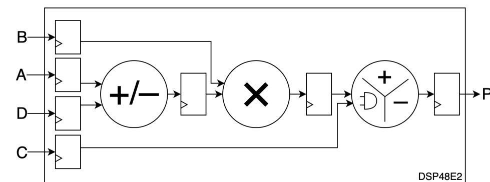
In the following sections, we walk through a real-world example using both existing tools and Lakeroad and highlight Lakeroad's design and key features (Section 2); formalize Lakeroad and demonstrate its correctness (Section 3); describe Lakeroad's implementation (Section 4); and evaluate Lakeroad on its completeness of mapping, extensibility, and expressiveness (Section 5) . Section 6 discusses related work, and Section 7 concludes.

2 Overview

We now walk through an example of how current FPGA technology mapping tools can fail a hardware designer (Section 2.1) and how Lakeroad overcomes these limitations (Section 2.2). In the process, we provide a high-level overview of Lakeroad's main components.

2.1 Compiling a Design to a DSP with Existing Tools

Consider the following scenario: A hardware designer is designing a large hardware block for the Xilinx UltraScale+ family of FPGAs. The designer is specifically aiming to use the UltraScale+'s specialized DSP48E2 units, which can implement combined multiplication, arithmetic, and logic operations, as captured in this simplified block diagram [51]:



The designer's hardware block involves the computation `(d+a)*b&c`, which the manual states is implementable with a single DSP. In particular, suppose the design consists of four separate instances of the following computation:

```
for(i=0; i<4; i++) begin
  r[i] <= (d[i] + a[i]) * b[i] & c[i];
end
```

It would be reasonable for the designer to expect the design to use a total of four DSPs.

Current tools fail. After compiling the design with existing tools, the designer is frustrated to find that the compiler returns a design that uses more resources than anticipated. It does use four DSPs, but it also uses 128 *registers* (which hold state) and 64 *lookup tables* (LUTs, which implement logic functions). **The compiler has thus failed to fully**

utilize the DSP—it has not configured a DSP48E2 to implement `(d+a)*b&c` but has instead implemented a portion of the computation with LUTs and registers. The designer now faces a choice: either accept the result or attempt to coax the compiler into returning a more optimal design.

Coaxing the compiler, to no avail. Though many may choose to accept a less optimal result, this tenacious[3] tries to coax the compiler into giving the expected results by placing the computation of interest into a separate module:

```
// add_mul_and.v: computes (a+b)*c&d in two clock cycles.
module add_mul_and(input clk, input [15:0] a, b, c, d,
                   output reg [15:0] out);
 reg [15:0] r;
 always @ (posedge clk) begin
   r <= (a+b)*c&d; out <= r;
 end
endmodule
```

This allows the designer to apply specific optimizations while mapping the module—a process we call *partial design mapping*. They attempt various strategies, including annotating the module with Xilinx's `use_dsp` Verilog attribute (to force the compiler to use a DSP where possible) and using a different synthesis directive (to apply a more resource-intensive synthesis procedure). **Despite these efforts, the compiler still cannot map the design to a single DSP,** instead using one DSP, 32 registers, and 16 LUTs. Again, the designer must decide: give up and accept suboptimal results, or press on?

Manual compilation. The hardware designer presses on and now has only one option remaining: manually instantiating a DSP48E2 with the desired behavior. Skimming through the daunting 75-page DSP48E2's online user manual, the designer quickly discovers that configuring even the "pre-add" `a+b` requires correctly setting multiple ports and parameters (`INMODE`, `AMULTSEL`, `BMULTSEL`, and `PREADDINSEL`), whose descriptions span 10+ pages and multiple tables. Correctly configuring the subsequent multiplier and logic unit proves even more difficult and time-consuming. After configuring the computational units, the designer must still manually ensure correct pipelining of the 10+ pipeline registers. After hours of frustration, a configuration is found that seems to work, which the designer inserts into the design. Precious time has been wasted, most of which will need to be repeated to configure the DSP again. Making matters worse, **the designer has no formal guarantees about the correctness of this DSP configuration.** It may work in a few simulated test cases, but are there corner cases that have been missed?

2.2 Compiling a Design to a DSP with Lakeroad

Lakeroad can save hardware designers the great effort involved in manual DSP configuration while also providing correctness guarantees. Let us imagine how the designer in this example, frustrated by conventional tools, can instead

[3]This may not be purely a personal preference. For example, a hardware design simply may not fit on an FPGA without manual optimizations!

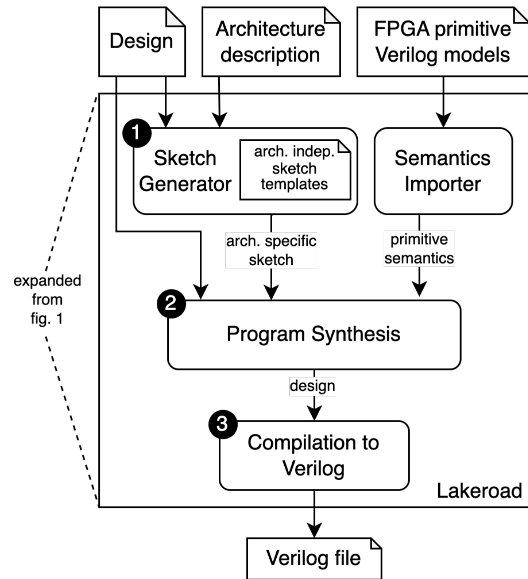


Figure 2. The components within Lakeroad.

proceed using Lakeroad during partial design mapping. After putting `add_mul_and` into its own module, the designer calls Lakeroad from the command line:

```
$ lakeroad --template dsp \
           --arch-desc xilinx-ultrascale-plus.yml \
           add_mul_and.v
```

The `lakeroad` command outputs `add_mul_and_impl`, an implementation of `add_mul_and` that uses a single UltraScale+ DSP48E2:

```
module add_mul_and_impl(input clk, input [15:0] a, b, c, d,
                        output [15:0] out);
 DSP48E2 #(
  .ACASCREG(32'd0), .ADREG(32'd0), .ALUMODEREG(32'd0),
  .AMULTSEL("AD"), .AREG(32'd0), .AUTORESET_PATDET("NO_RESET"),
  // ...plus 30+ more parameters
 ) DSP48E2_0 (
   .A({ 14'h0000, a }), .ACIN(30'h00000000), .ALUMODE(4'hc),
   .B({ 2'h0, b }), .BCIN(18'h00000), .C({ 32'h00000000, c }),
   .CARRYCASCIN(1'h0), .CARRYIN(1'h0), .CARRYINSEL(3'h6),
   // ...plus 30+ more ports
 );
endmodule
```

Unlike current compilers, Lakeroad has produced an implementation using a single DSP48E2 by utilizing more of the DSP's features. Importantly, this compiled design is also formally guaranteed to implement the input `add_mul_and` design.

How does Lakeroad provide *verified, more complete* support for the DSP48E2 over existing tools? At the core of Lakeroad's correctness and completeness is *sketch-guided program synthesis*, a technique that begins with a program *sketch*, which captures a rough outline of a program and uses automated reasoning tools (e.g., SMT solvers) to fill in the sketch's *holes*. As shown in Figure 2, Lakeroad uses the following three-step process to generate an efficient and correct DSP48E2 implementation of the `add_mul_and` design.

Step 1: Generating a Sketch. In the `add_mul_and` example, Lakeroad generates the following sketch, which we refer to as `sketch`:[4]

```
module sketch(input clk, input [15:0] a, b, c, d,
              output [15:0] out);
 DSP48E2 #(
  .ACASCREG(??), .ADREG(??), .ALUMODEREG(??), .AMULTSEL(??),
  .AREG(??), .AUTORESET_PATDET(??), ...
 ) DSP48E2_0 (
   .A({ 14'h0000, a }), .ACIN(??), .ALUMODE(??),
   .B({ 2'h0, b }), .BCIN(??), .C({ 32'h00000000, c }),
   .CARRYCASCIN(??),  .CARRYIN(??), .CARRYINSEL(??), ...
 );
endmodule
```

This sketch consists of a single DSP48E2 instance with *holes* (represented by `??`) serving as placeholders for most of its ports and parameters. It is easy to see the parallels between `sketch` and `add_mul_and_impl`; `sketch` is simply `add_mul_and_impl` with holes. But how does Lakeroad generate `sketch` in the first place?

To maximize portability across architectures, Lakeroad does not store sketches like `sketch` directly; instead, it *generates* sketches from architecture-independent **sketch templates.** Instead of storing the preceding UltraScale+–specific sketch, Lakeroad generates the sketch from the DSP sketch template, which the designer has chosen to use with the `--template dsp` flag. A simplified form of this template looks like the following:

```
module dsp_sketch_template(input clk,
                           input [n-1:0] a, b, c, d,
                           output [n-1:0] out);
 DSP dsp_instance(.clk(clk), .A(a), .B(b), .C(c), .D(d), .out(out));
endmodule
```

Sketch templates capture hardware design patterns that are common across FPGA architectures in an architecture-independent way. `dsp_sketch_template`, for example, captures a basic pattern, i.e., instantiating a single DSP. Lakeroad includes sketch templates of varying complexity, from the simplicity of the one above to the complexity of LUT-based multipliers. Though new sketch templates can be added easily, in most cases (as in this example) users can simply apply Lakeroad's provided templates.

To specialize `dsp_sketch_template` into `sketch`, Lakeroad translates the sketch template's generic `DSP` **primitive interface** into an UltraScale+–specific DSP48E2 using the UltraScale+ **architecture description.** The generic `DSP` module is an instance of a **primitive interface:** a Lakeroad-introduced abstraction that captures the similarities between primitives across diverse FPGA architectures. For example, Lakeroad's DSP primitive interface captures the facts that DSPs on all FPGA platforms generally have two to four data inputs (captured by `A`–`D`) and a clock input (captured by `clk`). To convert the sketch template's DSP primitive interface instance into a DSP48E2, Lakeroad utilizes the Xilinx UltraScale+ architecture description, which the designer has pointed to with the `--arch-desc xilinx-ultrascale-plus.yml` flag. An **architecture description** specifies how Lakeroad's various primitive interfaces are implemented for a given architecture. The following simplified snippet of the UltraScale+ architecture description, for example, tells Lakeroad that, when generating a sketch for UltraScale+, instances of the DSP primitive interface should be implemented with a DSP48E2:

```
- interface: {name: DSP, params: { out-width: 48, a-width: 30, ...}}
  holes: [?ACASCREG, ?ADREG, ?ALUMODEREG, ?AREG, ...]
  implementation:
    module: DSP48E2
    ports: [{ name: A, bitwidth: 30, value: A }, ...]
    parameters: [{ name: ACASCREG, value: ?ACASCREG }, ...]
    outputs: { O : P }
```

Thus, while converting `dsp_sketch_template` into `sketch`, Lakeroad reads this architecture description and converts the single DSP instance into a DSP48E2, filling the ports and parameters with the concrete values and holes contained in the architecture description. Architecture descriptions are usually short (100-400 LoC) and written only once per FPGA architecture; Lakeroad already contains such descriptions for Xilinx UltraScale+, Lattice ECP5, Intel Cyclone 10 LP, and the open-source FPGA SOFA [43].

To generate a sketch, Lakeroad takes an architecture-independent sketch template and specializes it using an architecture description. Once the sketch is ready, the designer can move on to synthesis.

Step 2: Program Synthesis. The next step fills in the holes to generate a complete, correct hardware design, which is done automatically using a technique called **program synthesis**. *Program synthesis* is the process of using automated reasoning tools (like SMT solvers) to generate correct programs by encoding program generation as a constraint solving problem. In our `add_mul_and` example, Lakeroad, aided by Rosette [45, 46], generates a query like the following:[5]

$$\exists\ \mathsf{ACASCREG}, \mathsf{ADREG}, \ldots\ .\ \forall \mathsf{inputs}.\quad \mathtt{add_mul_and}(\mathsf{inputs}) = \mathtt{sketch}(\mathsf{inputs}, \mathsf{ACASCREG}, \mathsf{ADREG}, \ldots)$$

The query asks: are there concrete values for `ACASCREG`, `ADREG`, etc., that will make our sketch's behavior equivalent to the input design's behavior on all inputs? If the solver finds such values, Lakeroad can use them to fill the holes in the sketch and produce a compiled design. However, if Lakeroad tries to pass the preceding formula to an SMT solver, the solver will throw an error since the query is not expressed at a level it understands, viz., as equalities between bitvector expressions, using simple Boolean or arithmetic operations. While it is conceivable that `add_mul_and` could be converted to a bitvector expression since its core computation is already expressed as `(a+b)*c&d`, it is unclear how to express `sketch` as an expression over bitvectors. In particular, Lakeroad must express bitvector-level semantics for Xilinx's DSP48E2 primitive.

To generate bitvector-level semantics for complex FPGA primitives, Lakeroad introduces the concept of **semantics**

[4] Though this example is presented in a Verilog-like language, Lakeroad's sketches are actually encoded in a Racket DSL that resembles structural Verilog.

[5] We formalize this synthesis query and explain it precisely in Section 3.

extraction. Rather than requiring manual effort to encode the semantics of the underlying hardware, which is notoriously difficult even for experts [10], Lakeroad's key insight is that these challenges can be avoided altogether by extracting low-level semantics directly from vendor-supplied simulation and verification models. Lakeroad builds on internal passes in Yosys [50] to automatically extract solver-ready semantics from these vendor-provided HDL models, which we detail in Section 4.4. For the `add_mul_and` example, the DSP48E2's semantics have already been imported into Lakeroad. Semantics need to be imported only when adding support for a new architecture, i.e., about as infrequently as writing a new architecture description. In most cases, Lakeroad users can rely on already-imported semantics.

With the sketch generated and the DSP48E2's semantics imported, program synthesis can begin. Lakeroad utilizes Rosette to drive program synthesis, as detailed in Section 3. In our example, Rosette returns a configuration for the DSP48E2. The last step, then, is to convert the compiled design to Verilog.

Step 3: Compilation to Verilog. Compiling Lakeroad's internal representation into Verilog is a purely one-to-one syntactic mapping; no optimizations are done at this stage, reducing the likelihood that bugs could be inserted. In our example, the final Verilog produced results in the `add_mul_and_impl` we saw at the start of Section 2.2.

In summary. Lakeroad delivered an implementation of the designer's `add_mul_and` module, improving upon both state-of-the-art compilers and manual approaches in multiple ways. Lakeroad's implementation is significantly more resource-efficient than the state-of-the-art compiler's—one DSP versus one DSP, 32 registers, and 16 LUTs. Lakeroad delivered its implementation in mere seconds, compared to the hours to days of work that manually instantiating a DSP might take. Lastly, Lakeroad's implementation is formally guaranteed to be correct. Meanwhile, Lakeroad did all of this while requiring no input from the user other than the Verilog to be compiled.

3 Formalization

We now formalize Lakeroad with functions f_{LR} and f^*_{LR}, and use these models to argue for the correctness and partial completeness of Lakeroad. We first define f_{LR} (Section 3.1) and then motivate and define the language $\mathcal{L}_{\text{LR}}$, specify its syntax and semantics, and define behavioral ($\mathcal{L}_{\text{BEH}}$), structural ($\mathcal{L}_{\text{STRUCT}}$), and sketch ($\mathcal{L}_{\text{SKETCH}}$) sublanguages (Section 3.2). We next explain the underlying queries Lakeroad uses to synthesize hardware programs that meet the desired specification (Section 3.3). We demonstrate the correctness and partial completeness of f_{LR}, enumerate our Trusted Computing Base (Section 3.4) and extend f_{LR} to f^*_{LR}, which ensures the generated program's semantics matches the design over multiple timesteps (Section 3.5). Finally, we highlight potential future applications that could be built on this section's formalization (Section 3.6).

3.1 The Lakeroad Function f_{LR}

We model the execution of Lakeroad with the partial function

$$f_{\text{LR}} : \textsc{Sketch} \times \mathcal{L}_{\text{BEH}} \times \mathsf{Time} \rightharpoonup \mathcal{L}_{\text{STRUCT}},$$

where $f_{\text{LR}}(\Psi, d, t)$ invokes Rosette to synthesize a t-cycle implementation of behavioral design d using sketch Ψ to guide the search, where a t-cycle implementation of d is a program that is equivalent to d at clock cycle t. By not requiring program equivalence before clock cycle t we allow the synthesized program's semantics to differ from the design during an initialization period (e.g., as the pipeline is being filled). To get guarantees beyond a single point in time t, we generalize f_{LR} to f^*_{LR}, which synthesizes a program that is equivalent to the design from time t to $t + n$. We formalize a sketch $\Psi \in \textsc{Sketch}$ as a tuple (ψ, h), where ψ is a program in $\mathcal{L}_{\text{SKETCH}}$ and h is a map from the holes in ψ to a finite set of valid hole-free nodes in $\mathcal{L}_{\text{STRUCT}}$ that can be used to fill the mapped hole. This mapping h is handled implicitly by Rosette's `choose` and `hole` constructs and need not be explicitly specified by the sketch writer. We write $f_{\text{LR}}(\Psi, d, t) = p$ to indicate that synthesis succeeded and produced Lakeroad program p. However, it is possible that sketch Ψ cannot implement d, in which case the synthesis fails (i.e., returns UNSAT) and f_{LR} does not return anything. Design d belongs to $\mathcal{L}_{\text{LR}}$'s behavioral fragment, $\mathcal{L}_{\text{BEH}}$ (see Section 3.2). When $t = 0$, f_{LR} synthesizes a *combinational design*; when $t > 0$, f_{LR} synthesizes a *sequential design* over t clock cycles. The rest of this section considers sequential design synthesis since its combinational counterpart is a special case covered by our general approach.

3.2 Defining $\mathcal{L}_{\text{LR}}$

Lakeroad uses the $\mathcal{L}_{\text{LR}}$ language to translate behavioral HDL programs to structural, hardware-specific HDL programs. To facilitate this translation, we designed $\mathcal{L}_{\text{LR}}$ to satisfy the following properties:

- **P1**. *Easy translation to/from HDLs:* we must be able to translate designs from a behavioral HDL to $\mathcal{L}_{\text{LR}}$ and translate synthesized implementations to a structural HDL.
- **P2**. *Support parallel stateful execution:* FPGA designs consist of potentially stateful elements executing in parallel. $\mathcal{L}_{\text{LR}}$ must allow unambiguous parallel execution.
- **P3**. *Support graph-like program structures:* An FPGA component's outputs can be wired to multiple other components, including back to itself. This means that FPGA programs can form arbitrary graphs, and $\mathcal{L}_{\text{LR}}$ must be able to express this.
- **P4**. *Support for sequential designs:* $\mathcal{L}_{\text{LR}}$ must handle designs that run over multiple clock cycles.

$$
\begin{aligned}
\text{Prog} &::= \langle\ \text{Id}, \langle \text{Id}, \text{Node}\rangle *\rangle \\
\text{Node} &::= \text{BV}\ b \mid \text{Var}\ x \\
&\mid \text{OP}\ op\ \text{Id}^* \\
&\mid \textbf{Reg}\ \text{Id}\ (\text{BV}\ b) \\
&\mid \textbf{Prim}\ \text{binds Prog} \\
&\mid \blacksquare_x
\end{aligned}
$$

Id	$id \in \mathbb{N}$
Bitvectors	$b \in \mathbb{BV}$
Variables	$x \in$ LegalVarNames
Operators	$op \in \text{OP}_{bv} \cup \text{OP}_w$
binds	$bs \in$ (Variables $\rightharpoonup$ Id)

Wire op	$\text{OP}_w = \{\texttt{concat},\texttt{extract}, \ldots\}$
Non-wire op	$\text{OP}_{bv} = \{+, -, \times, \ldots\}$

Figure 3. Syntax of $\mathcal{L}_{\text{LR}}$. $\blacksquare_x$ is a syntactic hole, labeled with variable x. $A \rightharpoonup B$ denotes the set of partial functions from A to B.

P5. *Support for different architectures:* $\mathcal{L}_{\text{LR}}$ must handle FPGA components from different architectures.

We describe how $\mathcal{L}_{\text{LR}}$ satisfies P1-P5 when we define its syntax and semantics in the following subsections.

3.2.1 $\mathcal{L}_{\text{LR}}$'s Syntax. Figure 3 shows the $\mathcal{L}_{\text{LR}}$ syntax. An $\mathcal{L}_{\text{LR}}$ program Prog consists of a root node ID and a graph of nodes, each of which is referred to by its ID. A *node* can be a constant bitvector, input variable, combinational (pure) operator, sequential (stateful) register, primitive, or hole. Given a program $p = (r, \langle id_1, node_1\rangle \ldots \langle id_n, node_n\rangle)$, we use the notation $p.root = r$, $p.ids = \{id_1, \ldots, id_n\}$, and $p[id_i] = node_i$. We define the free variables of a program $p.fv = \{x_i\}$ as the set of variable names occurring in p's nodes of the form (Var x_i).[6] Finally, we use the notation $p.all_ids$ for $p.ids$ together with $p'.all_ids$ of any subprogram p' of p (p' is a subprogram of p if $\exists j, node_j = \textbf{Prim}\ bs\ p'$).

Given a node n, we specify its inputs with the following function:

$$
\begin{aligned}
&\textsc{inputs}(\text{BV}\ b) = \{\}, \\
&\textsc{inputs}(\text{Var}\ x) = \{\}, \\
&\textsc{inputs}(\text{OP}\ op\ i_1 \ldots i_n) = \{i_1, \ldots, i_n\} \\
&\textsc{inputs}(\textbf{Reg}\ i\ b_{init}) = \{i\} \\
&\textsc{inputs}(\textbf{Prim}\ bs\ p') = \{bs[x] \mid x \in \text{domain}(bs)\}
\end{aligned}
$$

Note that we use $A \rightharpoonup B$ to denote the set of partial functions from A to B; given $bs \in A \rightharpoonup B$, we write domain$(bs)$ to denote the set of $x \in A$ s.t. $bs[x]$ is defined.

A program p is well-formed if and only if all the following conditions hold:

W1. $p.root \in p.ids$;

W2. All ids are unique and distinct. (i.e. for any sub-program p', $p.ids$ and $p'.all_ids$ are disjoint, and for any two sub-programs p' and p'', $p'.all_ids$ is disjoint from $p''.all_ids$.)

W3. The inputs of all nodes in p are ids of other nodes in p: $\forall id \in p.ids$, inputs$(p[id]) \subseteq p.ids$;

W4. All primitive nodes contain well-formed programs;

W5. All primitive nodes bind exactly their free variables; i.e., for **Prim** $bs\ p'$, domain$(bs) = p'.fv$; and

W6. Program p is free of combinational loops (formalized below in Property 1).

Property 1 (Free of Combinational Loops)**.** *Formally, a program p is free of combinational loops if there exists a function $w : p.all_ids \to \mathbb{N}$, that satisfies the following properties (collectively "monotonicity"):*

1. *If $p[id] = \textbf{Reg}\ _\ _$, then $w(id) = 0$;*
2. *If $p[id] = \textbf{Prim}\ bs\ p'$, then $w(id) > w(p'.root)$;*
3. *if $p[id] = \textbf{Prim}\ bs\ p'$ and $p'[id'] = Var\ x$, then $w(id') > w(bs[x])$; and*
4. *Otherwise (e.g., $p[id] = \text{OP}\ op\ ids^*$), if $id' \in \textsc{inputs}(p[id])$, then $w(id) > w(id')$.*

The function w acts as a witness to the absence of combinational loops because it is impossible to define a strictly monotonic function without acyclicity. We consider only well-formed $\mathcal{L}_{\text{LR}}$ programs.

BV, Var, and OP nodes encode bitvectors, variables, and operators.

Reg $i_{data}\ b_{init}$ nodes let $\mathcal{L}_{\text{LR}}$ implement sequential designs (P4). i_{data} is the register's data input, which updates the stored value at the positive edge of each clock cycle, and b_{init} is the register's initialization value.

Prim $bs\ p$ nodes let $\mathcal{L}_{\text{LR}}$ programs use hardware-specific components from different architectures (P5). The bs component is a *variable map*, mapping Vars to input Ids. The p component is an $\mathcal{L}_{\text{LR}}$ program that defines the semantics of the hardware primitive. A **Prim** node also carries some metadata used during compilation to a structural HDL, which we omit for clarity.

$\mathcal{L}_{\text{BEH}}$ is the concrete *behavioral* fragment of $\mathcal{L}_{\text{LR}}$ used for writing specifications; it is formed by excluding **Prim** nodes and holes from $\mathcal{L}_{\text{LR}}$.

$\mathcal{L}_{\text{STRUCT}}$ is the concrete *structural* fragment of $\mathcal{L}_{\text{LR}}$ used for lowering $\mathcal{L}_{\text{LR}}$ to structural HDLs; it is formed by excluding **Reg** nodes, OP nodes, and holes from $\mathcal{L}_{\text{LR}}$, with the following exception: the p term in **Prim** $bs\ p$ must always be from the $\mathcal{L}_{\text{BEH}}$ since it is used to specify the semantics of the **Prim** node to the synthesis engine. The behavioral node p is not used during compilation to HDL, and this behavioral expression does not propagate to the structural HDL output.

[6] Note that this does not include variables of sub-programs occurring recursively inside of **Prim** nodes.

Time $t \in \mathbb{N}$ Env $e \in (\text{Var} \rightharpoonup \text{Time} \rightarrow \text{BV})$

$\textsc{Interp} : \text{Prog} \rightarrow \text{Env} \rightarrow \text{Time} \rightarrow \text{Node} \rightarrow \text{BV}$
$\textsc{Interp}\ p\ e\ t\ (\text{BV}\ b) = b$
$\textsc{Interp}\ p\ e\ t\ (\text{Var}\ x) = e\ x\ t$
$\textsc{Interp}\ p\ e\ 0\ (\textbf{Reg}\ _\ init) = init$
$\textsc{Interp}\ p\ e\ (t+1)\ (\textbf{Reg}\ id\ _) = \textsc{Interp}\ p\ e\ t\ p[id]$
$\textsc{Interp}\ p\ e\ t\ (\text{OP op}\ ids) = [\![op]\!]\ (\text{map}\ (\lambda id\ .\ \textsc{Interp}\ p\ e\ t\ p[id])\ ids)$
$\textsc{Interp}\ p\ e\ t\ (\textbf{Prim}\ bs\ p') =$
let $e' = \lambda x, t'\ .\ \textsc{Interp}\ p\ e\ t'\ (p[bs\ x])$ in
$\textsc{Interp}\ p'\ e'\ t\ p'[p'.root]$

Figure 4. Lakeroad's semantics as pseudocode.

$\mathcal{L}_{\text{SKETCH}}$ is another sublanguage of $\mathcal{L}_{\text{LR}}$ that is $\mathcal{L}_{\text{STRUCT}}$ but also including holes. Let s be a program in $\mathcal{L}_{\text{SKETCH}}$ with holes $\blacksquare_{x_1}, \ldots, \blacksquare_{x_k}$. These holes can be *filled* with nodes $n_1, \ldots, n_k$ in $\mathcal{L}_{\text{STRUCT}}$ by replacing each hole $\blacksquare_{x_i}$ with its corresponding node n_i to obtain a complete $\mathcal{L}_{\text{STRUCT}}$ program, denoted by $s[\blacksquare_{x_1} \mapsto n_1, \ldots]$.

The simplicity of this syntax makes translating to and from HDLs straightforward (P1). Section 4 describes how Lakeroad implements the translations to and from HDLs.

3.2.2 $\mathcal{L}_{\text{LR}}$'s Semantics.

Before discussing the formal semantics of $\mathcal{L}_{\text{LR}}$, we present key definitions. We assume a *bitvector type* and, for simplicity, we elide bitvector widths. We represent *time* as a natural number. A *stream* is a function from Time to bitvectors. An *environment* is a map from variable names to streams.

We give the semantics for $\mathcal{L}_{\text{LR}}$ as an interpreter in Figure 4. We define the function $\textsc{Interp}$ to interpret a program p in environment e at time t and node n. We do not define semantics for holes, as they are intended to be replaced by other constructs with well-defined semantics.

Most of the rules are straightforward. A bitvector BV b evaluates to its backing bitvector value b. A variable node Var x in an environment e at time t evaluates to the value returned by the stream associated with x in e at time t; using function notation, this is denoted by $e\ x\ t$. A k-ary operator node OP $op\ i_1 \ldots i_k$ recursively interprets each operand in the current environment at the current time and then applies op's semantics, denoted $[\![op]\!]$, to the resulting values. A register **Reg** $id\ b_{init}$ has two cases depending on the current time: at time $t = 0$, a register evaluates to its initial bitvector value b_{init}; at nonzero times $t+1$, a register evaluates to the value produced by the input i at the *previous* timestep t. A primitive **Prim** $bs\ p'$ in environment e at time t is evaluated by interpreting the program p' under the fresh environment e' formed by the binding map bs.

3.3 Program Synthesis

f_{LR} performs sketch-based program synthesis [41]. Operationally, we implement the $\textsc{Interp}$ function from Figure 4 in Rosette, a solver-aided host language [46]. Let sketch $\Psi = (\psi, h) \in \textsc{Sketch}$, where $\psi \in \mathcal{L}_{\text{SKETCH}}$ has holes $\blacksquare_{x_i}$ and h maps ψ's holes to the set of structural nodes that can legally fill the mapped hole. Given a design d, we query Rosette if there are nodes $n_1, n_2, \ldots n_k$ such that $n_i \in h[\blacksquare_{x_i}]$ and $p = \Psi[\blacksquare_{x_1} \mapsto n_1, \ldots]$ is well-formed and equivalent to d (i.e., we ask Rosette to fill each hole with a node associated with the node in h). Program equivalence between well-formed programs p and d at time t, written $p \cong_t d$, is defined as

$$
\begin{aligned}
&p.fv = d.fv \wedge \\
&\forall e\ s.t.\ \text{domain}(e) = p.fv, \\
&\quad \textsc{Interp}\ p\ e\ t\ p.root = \textsc{Interp}\ d\ e\ t\ d.root.
\end{aligned}
$$

In Section 3.5, we use bounded model checking to extend f_{LR}'s guarantees beyond the single timestep at clock cycle t.

3.4 Correctness and Completeness of f_{LR}

Recall that the synthesis function f_{LR} is partial. We say that f_{LR} is *correct* if it returns a program $f_{\text{LR}}(\Psi, d, t) = p$ where p is a well-formed completion of $\Psi = (\psi, h)$, meaning $p = \Psi[\blacksquare_{x_1} \mapsto n_1, \ldots]$ such that $n_i \in h[\blacksquare_i]$ for all i and $p \cong_t d$.

Furthermore, we say that f_{LR} is *sketch-complete* if $f_{\text{LR}}(\Psi, d, t)$ is defined whenever there exists a well-formed completion p of Ψ such that $p \cong_t d$. That is, synthesis is correct if it never returns an erroneous result and sketch-complete if it returns a correct result whenever one exists.

We have implemented f_{LR} with Rosette (see Section 3.3), which guarantees our system is correct and complete under the following assumptions:

1. Correctness of Rosette and underlying SMT solvers;
2. That our encoding of Lakeroad is bug-free;
3. That the lowering of $\textsc{Interp}$ to SMT formulas by Rosette always terminates. This is possible when partial evaluation of $\textsc{Interp}$ on arguments p, t and n terminates (independently of the value of e).

Lemma 3.1. *Let p be a well-formed program, e an environment, t a Time, and n be a node belonging to p. Then $\textsc{Interp}$ is primitive recursive (i.e. terminates) in the arguments p, t, and n.*

Proof of Lemma 3.1. Recall that a function $f(x, y, z)$ is primitive recursive in arguments x and y (under a lexicographic ordering) if in the definition of f every recursive call $f(x', y', z')$ is made with values (x', y') such that $x' < x$ or $x' = x \wedge y' < y$. If x and y are drawn from the natural numbers (or another well-ordered set), then the recursion is guaranteed to terminate.

Under what order is $\textsc{Interp}$ primitive recursive? Because our program is well-formed, it must be free of combinational loops (see Property 1). Formally, this means we have an

acyclicity witness function $w : p.all_ids \to \mathbb{N}$ that monotonically increases in the direction of dataflow in our circuit. Each node n argument passed to Interp has an Id that is unique and distinct from the Ids used in p or any of p's subprograms (**W2**); we denote this Id as id_n. We can associate each n argument to a recursive call of Interp with a number $w(id_n)$. We claim that Interp is primitive recursive under the lexicographic ordering on $(t, w(id_n))$.

To prove this claim we need to demonstrate that if Interp with time and node arguments t' and n' makes a recursive call to Interp with time and node arguments t'' and n'', then the following condition holds:

$$t'' < t' \lor \left(t'' = t' \land w(id_{n''}) < w(id_{n'})\right). \quad (1)$$

To do this it suffices to examine each case of Interp's definition.

When n' is a BV constant, Interp makes no recursive calls, and the condition in Equation (1) holds vacuously.

When n' is a **Reg** node Interp either terminates (when $t' = 0$) or makes a recursive call with time value $t'' = t' - 1$, maintaining the condition in Equation (1).

When n' is an operator node, Interp recursively interprets the operands with time arguments $t'' = t'$. However, each operand's id id'' belongs to inputs(n'), and, by Property 1, $w(id_{n'}) > w(id'')$, so our condition holds.

This leaves us with the less obvious cases in which n' is either a **Prim** or Var, which work together in tandem. When $n' =$ **Prim** $bs\ p'$, Interp makes a recursive call with node argument $p'.root$ and time argument t. By Property 1, $w(p'.root) < w(id_{n'})$, and the condition in Equation (1) holds. Interp also defines a new environment for execution of p' via λ-abstraction, and this in turn will recursively invoke Interp. These environments are only invoked by the rule for variables, which we handle presently.

When $n' =$ Var x, the environment is invoked on variable x. Here, there are two possible cases. First, we are interpreting the top-level program p. As this is the initial, top-level environment, there is no further recursion. Second, we are interpreting a sub-program p' and $e'\ x\ t =$ Interp $p\ e\ t\ (p[bs\ x])$ is actually a recursive call into the program p one level up, with its environment e. In this latter case, note that w is defined such that $w(id_{p[bs\ x]}) = w(bs\ x) < w(id_{\text{Var}\ x})$ (item 3 of Property 1), satisfying our property. All cases are complete. ◻

From this, we conclude that all possible substitutions for Ψ are attempted, and f_{LR} is sketch-complete.

Trusted Computing Base. The *trusted computing base* (TCB) of a system is the set of components it assumes to be correct [29]. A bug anywhere in the TCB could cause the guarantees made by that system to be violated. Lakeroad's TCB includes: Rosette and the underlying SAT/SMT solvers that Rosette queries (Bitwuzla, cvc5, Yices2, and STP); the internal Yosys passes Lakeroad uses to extract primitive semantics and translate design specifications from behavioral Verilog into $\mathcal{L}_{\text{BEH}}$; the semantics for $\mathcal{L}_{\text{LR}}$, which we assume conservatively models non-cyclic (DAG) designs; our code to translate from the $\mathcal{L}_{\text{STRUCT}}$ to structural Verilog; and the vendor-provided Verilog simulation models for FPGA primitives. Each TCB component has also been thoroughly tested, as described in Section 5. Importantly, sketches and sketch generation are *not* in Lakeroad's TCB: even if there were a bug in Lakeroad's sketch-related components, it would not violate Lakeroad's correctness guarantees.

3.5 Multiple Clock Cycle Guarantees with f^*_{LR}

The preceding completeness and correctness properties for f_{LR} guarantee that running the synthesized program p and the design d for t clock cycles produces the same output. To extend this guarantee, Lakeroad supports a form of bounded model checking, where synthesis ensures that p is semantically equivalent to d for c additional clock cycles starting at time t. We formalize this with the function f^*_{LR}, which takes a sketch Ψ, a behavioral design d, a number of clock cycles t, and a model checking time bound $c \geq 0$ and returns an implementation $p \in \mathcal{L}_{\text{STRUCT}}$ that is equivalent to d at time steps $t, t+1, \ldots, t+c$.

Our correctness and completeness guarantees are similar to those for f_{LR}:

$$\begin{aligned}&p.fv = d.fv \land \\ &\forall e\ s.t.\ \text{domain}(e) = p.fv, \\ &\bigwedge_{i=t}^{i=t+c} \text{Interp}\ p\ e\ i\ p.root = \text{Interp}\ d\ e\ i\ d.root.\end{aligned}$$

3.6 Beyond Lakeroad

$\mathcal{L}_{\text{LR}}$, its semantics, and the synthesis approach we describe here are useful for applying program synthesis to other hardware design problems. For example, the synthesis problem detailed above could be "flipped" to decompile structural designs back to higher-level behavioral designs, i.e., synthesizing from $\mathcal{L}_{\text{STRUCT}}$ to an expression in $\mathcal{L}_{\text{BEH}}$. Such decompilation has seen recent interest for recovering equivalent but faster-to-simulate models and for porting models across different architectures [40]. As another example, the synthesis approach could be adapted to help port designs by synthesizing expressions in $\mathcal{L}_{\text{STRUCT}}$ that use one set of primitives on one architecture from other designs in $\mathcal{L}_{\text{STRUCT}}$ that use a different set of primitives from a different architecture. Thus, the formalization in this section transcends the particular challenges of FPGA technology and provides a reusable foundation for exploring a much broader range of hardware design challenges from a program synthesis perspective.

```
implementations:
  - interface: { name: LUT, num_inputs: 4 }
    internal_data: { sram: 16 }
    modules:
      - module_name: frac_lut4
        filepath: SOFA/frac_lut4.v
        ports:
          - { name: in, direction: in, width: 4,
              value: (concat I3 I2 I1 I0) }
          - { name: mode, direction: in,
              width: 1, value: (bv 0 1) }
          - { name: lut4_out, direction: out,
              width: 1 }
        parameters: [{ name: sram, value: sram }]
    outputs: { O: lut4_out }
```

Figure 5. SOFA architecture description.

4 Implementation

Lakeroad is composed of approximately 13K lines of Racket plus approximately 58K lines of Racket automatically generated from vendor-supplied Verilog. Vendor-supplied Verilog was obtained from Lattice Diamond, Intel Quartus, and Xilinx Vivado sources. We used Vivado version v2023.1, Quartus 22.1std.1 Build 917 02/14/2023 SC Lite Edition, Diamond version 3.12, Yosys version 0.36+42 (commit `70d3531`), the cvc5 [8] and Yices2 [18, 19] solvers included in the 2023-08-06 release of `oss-cad-suite` from YosysHQ, the Bitwuzla solver at commit `b655bc0` [32], the STP solver at commit `0510509a`, Racket version 8.9 [20, 21], and Rosette version 4.1 [36].

4.1 Primitive Interfaces

As described in Section 2, *primitive interfaces* describe abstract versions of common FPGA primitives, which allow sketch templates to be architecture-independent. To date, Lakeroad declares primitive interfaces for n-input LUTs, w-width carry chains, n-input muxes, and DSPs with up to four data inputs and one clock input. The next section includes a concrete example of Lakeroad's LUT4 primitive interface.

4.2 Architecture Descriptions

As described in Section 2, *architecture descriptions* convey the information required to convert each instance of a primitive interface into the corresponding architecture-specific module, which occurs while converting sketch templates into sketches. The architecture description is the only additional input that may be required from a user to support a new architecture; it is a one-time effort that is reusable for any designs in an architecture. Architecture descriptions are simply lists (provided as YAML files) of the primitive interfaces that an architecture implements, but, crucially, also include architecture-specific port and parameter values in a map called `internal_data`. Values in this map become symbolic values solvable by the SMT solver. Additional constraints can also be specified in the architecture description to rule out invalid configurations and minimize the solver's search space.

As an example, Figure 5 shows the architecture description for the SOFA [43] FPGA architecture. The description contains a single primitive interface implementation, i.e., LUT4. Lakeroad's LUT4 primitive interface standardizes the names of a LUT4's inputs and outputs, naming the inputs `I0` through `I3` and the output `O`. The SOFA implementation of the LUT4 primitive interface uses the SOFA-specific `frac_lut4` primitive. Primitive interface inputs `I0` through `I3` are mapped to the actual input port of the `frac_lut4`, named `in`. Likewise, the `frac_lut4` output `lut4_out` is mapped to the primitive interface output `O`. The `internal_data` field declares `sram`, the LUT's 16-bit internal memory, as an architecture-specific detail to be solved during synthesis.

If a sketch template uses a primitive interface not included in the architecture description (e.g., SOFA does not implement carries), Lakeroad may still be able to implement the primitive interface based on primitive interfaces the architecture *does* implement. To date, Lakeroad can implement any mux with LUTs, a larger LUT from smaller LUTs, a smaller LUT from a larger LUT, a carry from LUTs, and a smaller DSP from a larger DSP; it handles these conversions during sketch generation.

4.3 Sketch Templates, Sketches, and Sketch Generation

As described in Section 2, Lakeroad captures common FPGA implementation patterns in reusable, architecture-independent *sketch templates*. Thus far, we have described only the relatively simple `dsp` sketch template, which instantiates a DSP. As a more complex example of capturing common FPGA implementation patterns, consider the `bitwise-with-carry` sketch template, which uses n LUTs and a carry chain to implement designs such as addition or subtraction. As of the paper's publication date, Lakeroad provides 5 sketch templates: `dsp`, `bitwise`, `bitwise-with-carry`, `comparison` (LUT- and carry-based arithmetic comparison), and `multiplication` (LUT-based multiplication).

The process of converting sketch templates to sketches is implemented as described in Section 2 and Section 4.2. Lakeroad iterates over every primitive interface instance in the sketch and replaces it with the concrete primitive in accordance with the architecture's architecture description. If the architecture description does not implement the requested primitive interface, Lakeroad checks whether it can implement the primitive interface with other implemented interfaces (e.g., implementing a smaller LUT with a larger LUT) and raises an error otherwise.

Sketch templates and sketches alike are written in a domain-specific language (DSL) embedded into Rosette, whose implementation closely mirrors the syntax and semantics of $\mathcal{L}_{\text{LR}}$. The only significant difference is that the interpreter

implementation does not use bitvector streams natively. Instead, each invocation of the interpreter represents a single timestep, and all intermediate values from the previous timestep are taken as input. Streams are then built up using multiple invocations of the interpreter.

4.4 Importing Semantics from Verilog Modules

Lakeroad uses Yosys [50] to convert Verilog modules into the `btor2` format [33] and then converts the resulting `btor2` to Rosette/Racket code.

Due to the semantics of the Verilog language and the internal implementation of Yosys, extracting semantics from Verilog modules may require the following manual modifications to accommodate semantics extraction and synthesis:

- As Yosys converts parameters from variables to constant values immediately upon module import, module parameters should be converted to ports to ensure they remain variables (and thus solvable by the SMT solver). Note that not all parameters can always be converted to ports, meaning some parameters cannot be solved for.
- Strings should be converted to bitvectors.
- All registers should be initialized.
- All instances of `x` and `z` values should be converted to 2-state logic (0 or 1).

Note that these caveats apply only to our prototype implementation, not the general technique of semantics extraction from HDL. Once these manual modifications are made, the following series of Yosys passes can be used to convert the Verilog into suitable `btor2`: `prep; flatten; pmuxtree; opt_muxtree; clk2fflogic; prep; write_btor`.

We implement the translation from `btor2` to Rosette bitvector expressions as a 1:1 translation since both languages are simply operations over bitvectors.

4.5 Program Synthesis and Compilation to Verilog

We implement the synthesis procedure defined in Section 3.4 with Rosette. Multiple clock cycle guarantees, as described in Section 3.5, are implemented simply by making $c + 1$ total assertions, asserting the output of the input design and the sketch are equal after each of the $c + 1$ timesteps. We use a portfolio solving method, running Bitwuzla [31], cvc5 [8], Yices2 [18, 19], and STP [5] in parallel and using results from the first solver to terminate. To produce Verilog, Lakeroad compiles the program from its internal DSL to the JSON format defined by Yosys using a straightforward translation and then uses Yosys to output Verilog.

4.6 Integration with Other Tools

This paper describes Lakeroad as a standalone tool, but the core Lakeroad implementation could be integrated directly into existing tools. Though out of scope for this paper, we have early, encouraging results integrating Lakeroad as a Yosys pass that lets users tag modules with annotations similar to (and much richer than) Xilinx's `use_dsp` annotation. We then map annotated modules to primitives using Lakeroad, which let us easily apply Lakeroad to many fragments within a larger design. We plan to more fully integrate Lakeroad into Yosys in future work, which should radically improve the completeness of Yosys's DSP mapping ability, as shown in Figure 6.

5 Evaluation

We now evaluate Lakeroad in terms of completeness and extensibility. In the following experiments, we target four FPGA architectures: **Xilinx UltraScale+**, commonly used for large, high-performance workloads; **Lattice ECP5**, commonly used in low-power, low-cost scenarios; **Intel Cyclone 10 LP**, an FPGA designed for low-cost, high-volume use cases, and **SOFA** [43], a recent, open-source FPGA developed by the research community. We compare Lakeroad to existing technology mappers. For Xilinx Ultrascale+, Lattice ECP5, and Intel Cyclone 10 LP, we compare Lakeroad against both the open source toolchain Yosys [50] and the state-of-the-art, proprietary, closed source toolchains for each architecture.[7] The experiments were conducted on a system running Ubuntu 20.04.3 with an AMD EPYC 7702P 64-Core CPU. The resident set size of a single Lakeroad process did not exceed 300MB while running our evaluation. We use the software versions listed in Section 4.

5.1 Lakeroad Completeness

The reliance of many technology mappers, including state-of-the-art tools, on hand-written patterns leads them to fail when attempting to map many workloads that *should* be mapped to a single DSP. In particular, the process of partial design mapping (illustrated in Section 2) becomes a laborious endeavor because of this incompleteness: hardware designers hand-instantiate DSPs rather than rely on substandard automated tooling, repeating the work each time they identify a potential opportunity to use a DSP. Lakeroad's greater mapping completeness significantly reduces the burden on hardware designers during partial design mapping and marks the first step in automated mapping for full designs. We next evaluate how Lakeroad's program synthesis approach enables it to achieve greater completeness for these program fragments.

Evaluation Setup. We highlight three particularly complex DSPs for the Xilinx Ultrascale+, Lattice ECP5, and Intel Cyclone 10 LP architectures: the Xilinx DSP48E2, Lattice ALU54A/MULT18X18C (a single DSP composed of two primitives), and Intel cyclone10lp_mac_mult. SOFA provides no DSP, and is not included in this part of the evaluation. For each architecture's DSP, we enumerate a large subset of the

[7] Again, licensing restrictions prevent our naming the specific proprietary tools, but they are familiar, standard packages used by many hardware designers.

designs theoretically mappable to a single DSP according to its configuration manual. This microbenchmark set aims to capture the real-world designs which hardware designers would attempt to map to a platform's DSP. For each architecture, we compare Lakeroad to both the corresponding state-of-the-art toolchain for the architecture as well as to Yosys. For Xilinx Ultrascale+, the DSP48E2 configuration manual details the structure of designs mappable to the primitive. Our designs for Xilinx include all permutations of the design form $((a \pm b) * c) \odot d$, where $\odot \in \{\&, |, \pm, \oplus\}$, as well as designs of the forms $(a * b)$ and $((a * b) \pm c)$. We pipeline each of these workloads from zero to three stages and use bitwidths from 8 to 18 bits. For the DSP on Lattice, we similarly enumerate all designs of the form $(a * b) \odot c$, where $\odot \in \{\&, |, \oplus, \pm\}$, and of the form $(a * b)$. For each of these designs, we use zero to two stages and bitwidths from 8 to 18 bits. This results in 1320 microbenchmarks for Xilinx UltraScale+, 396 for Lattice ECP5, and 66 for Intel Cyclone 10 LP. Though Lakeroad's output is correct by construction, we further validate its output by simulating each Lakeroad-compiled design over thousands of consecutive cycles using Verilator.

Comparison to Existing Toolchains. As demonstrated in Figure 6 (top), Lakeroad maps 44× more designs than Yosys and 2.1× more designs than the proprietary, state-of-the-art toolchain on Xilinx Ultrascale+. On Lattice ECP5, Lakeroad maps 6.0× more designs than Yosys and 3.6× more designs than the proprietary, state-of-the-art toolchain. On Intel Cyclone 10 LP, Lakeroad successfully maps all designs: 3× more designs than the proprietary, state-of-the-art toolchain for Intel. Yosys fails to map a single design on Intel. State-of-the-art toolchains for all architectures fail to map more than half of the queried designs. Lakeroad times out on less than 20% of designs.[8] Note that Lakeroad returns "UNSAT" on approximately 260 designs on UltraScale+, i.e., Lakeroad claims there is *no* possible mapping to a DSP48E2 for the requested workload. In all of these cases, both Xilinx SOTA and Yosys agree with Lakeroad and do not map the designs to a single DSP. We conclude that the set of designs we presented in *Evaluation Setup* must be overly broad; though the documentation implies that all of these designs are mappable to a single DSP, all three Xilinx synthesis tools surveyed indicate that they are indeed not mappable.

For timing, we compared the mapping time for each of the tools and report the results in Figure 6 (bottom). The wide ranges for Lakeroad show that solver time for different program synthesis queries is highly variable. This is explored more deeply in Figure 7, which shows that most synthesis queries terminate quickly, with a long tail of slower queries. Note that the state-of-the-art technology mapper for Ultrascale+ has a slow running time due to its long start-up process.

Regarding which solvers in the portfolio were most useful, of all terminating (success or UNSAT) Lakeroad experiments, Bitwuzla was the first to complete for 671 of them, STP for 519, Yices2 for 464, and cvc5 for 64.

Lakeroad's greater completeness directly translates into resource reduction. On average, for each microbenchmark, Lakeroad uses 3.9 fewer LEs (logic elements: LUTs, muxes, or carry chains) and 7.5 fewer registers than the Xilinx SOTA, 7.2 fewer LEs/11.9 fewer registers than the Lattice SOTA, 8.2 fewer LEs/14.3 fewer registers than the Intel SOTA, and 33.3 fewer LEs/11.4 fewer registers than Yosys. In the real world, the small modules captured by our microbenchmarks may be reused dozens if not hundreds of times across a large design. Thus, the sizable resource reduction Lakeroad provides on a single microbenchmark will be multiplied significantly for an entire design.

Discussion. Compared to Yosys, it is clear that Lakeroad provides more complete support for programmable DSPs. However, Lakeroad's greater completeness over Yosys is perhaps not surprising since Yosys is an open-source tool still under active development. Part of the appeal of the Yosys toolchain is the diversity of backends it can target; these results show that, if incorporated into Yosys, Lakeroad would further increase Yosys's flexibility and generality. Perhaps most surprising is that Lakeroad is more complete than specialized proprietary toolchains. Even the UNSAT results Lakeroad produces can be useful to designers since they indicate potential flaws in the documentation or vendor-provided semantics. In the context of a larger synthesis tool, Lakeroad would provide stronger guarantees for mapping modules of larger designs.

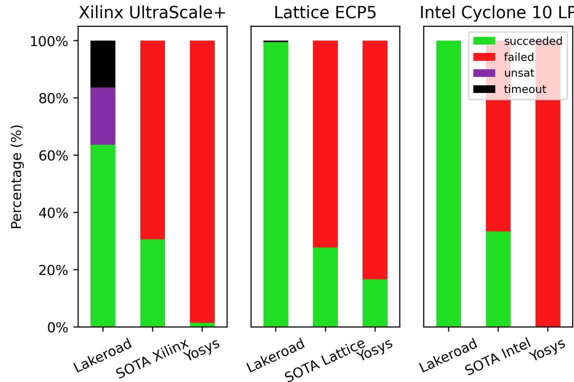
5.2 Lakeroad Extensibility and Expressiveness

In addition to being correct by construction (Section 3) and more complete than existing FPGA technology mappers (Section 5.1), Lakeroad can also easily extend to new FPGA architectures. Furthermore, automatic primitive semantics extraction from vendor-provided HDL simulation models enables Lakeroad to support diverse, highly configurable FPGA primitives.

The architecture descriptions vary in length from 20 to 240 source lines of code (SLoC). SOFA (20 SLoC) is the simplest, shown in full in Figure 5. The descriptions for Xilinx (185 SLoC), Lattice (240 SLoC), and Intel (178 SLoC) are longer since those FPGA architectures provide a wider range of configurable primitives.

As a point of comparison, the open-source Yosys toolchain, which has roughly 200 contributors on GitHub, provides technology mapping for Xilinx UltraScale+ across over a dozen complex Verilog, C++, and Python files (about 1300

[8]We restricted Rosette synthesis time to 120 seconds, 40 seconds, and 20 seconds for Xilinx, Lattice, and Intel respectively, and marked failure past that (though bitvector synthesis problems are decidable).



Tool	Median Time (s)	Min / Max Time (s)	
Xilinx			
Lakeroad	14.99	2.99	127.70
SOTA Xilinx	261.61	227.82	598.67
Yosys	14.97	6.66	21.10
Lattice			
Lakeroad	9.49	6.70	55.23
SOTA Lattice	2.32	0.95	4.52
Yosys	2.31	0.90	4.01
Intel			
Lakeroad	2.92	2.12	4.13
SOTA Intel	38.73	19.11	43.49
Yosys	0.96	0.48	1.88

Figure 6. Results of the completeness experiments described in Section 5.1, measuring the completeness of technology mapping tools for DSPs on Xilinx UltraScale+ and Lattice ECP5, plus timing information. A single bar in the bar chart communicates, for a given FPGA architecture and technology mapper, the proportion of the microbenchmarks that the given technology mapper could map to a single DSP. In Lakeroad's case, experiments can either succeed (Lakeroad maps the microbenchmark to a single DSP), timeout, or return UNSAT. For the other tools, experiments can either succeed or fail (i.e., the tool returns a mapping, but the mapping uses more than a single DSP). There are a total of 1320 experiments/microbenchmarks for Xilinx, 396 for Lattice, and 66 for Intel.

lines of code). We cannot provide similar numbers for state-of-the-art proprietary tools, but a developer of one such technology mapper shared that extending their tool to support new FPGA architectures was extremely difficult since it "spans millions of lines of low-level C." This is not surprising; Yosys aims to target a variety of vendor architectures, while proprietary tools have teams of engineers to extract better mapping (evident by Yosys' limitations in Section 5.1). By contrast, Lakeroad supports diverse architectures and is easy to extend. Even if a user wants to target a completely new architecture that Lakeroad does not support, architecture-independent sketch templates allow reuse of previously implemented mapping strategies, and the user is only required

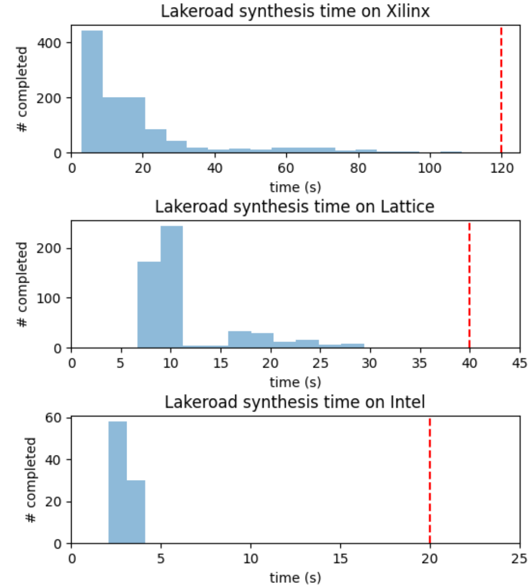


Figure 7. Histograms of Lakeroad program synthesis runtime for all terminating (success or UNSAT) Lakeroad experiments described in Section 5.1, with timeout thresholds indicated with a vertical dotted red line.

Table 1. FPGA primitives imported automatically by Lakeroad from vendor-provided Verilog models, with number of source lines of code (excluding comments and empty lines) of the original Verilog models.

FPGA	Primitive	Verilog SLoC
Xilinx Ultrascale+	LUT6	88
	CARRY8	23
	DSP48E2	896
Lattice ECP5	LUT2	5
	LUT4	7
	CCU2C	60
	ALU54A	1642
	MULT18X18C	795
Intel Cyclone 10 LP	cyclone10lp_mac_mult	319
SOFA	frac_lut4	69

to provide a few lines of high-level configuration for each primitive in the architecture description.

Table 1 further highlights Lakeroad's expressiveness, i.e., its ability to support a diverse range of configurable primitives by automatically extracting semantics from vendor-provided HDL simulation models. Lakeroad can import the semantics of large configurable primitives, such as the UltraScale+ DSP (896 lines of Verilog) or Lattice ECP5's ALU and multiplier units (1642 and 795 lines of Verilog, respectively). It is difficult and error-prone to manually formalize the full semantics for these primitives; partial support by ad hoc search procedures that rely on syntactic pattern matching leads to missing many mapping opportunities, as shown in Section 5.1.

6 Related Work

To the best of our knowledge, Lakeroad is the first work to apply the technique of program synthesis to FPGA technology mapping. Indeed, as noted by Sisco *et al.* [39], program synthesis has seldom been applied in the domain of hardware design although its underlying formal methods techniques are frequently used for the *formal verification* of hardware designs rather than compilation, as in Bluespec SystemVerilog [34], Kôika [12], and Kami [15]. Sisco *et al.* cite two examples of works that use program synthesis for hardware design, Verisketch [7] and Sketchilog [9], both of which apply program synthesis to produce HDL implementations from high-level designs. Other works use program synthesis to generate *software* that runs on low-powered hardware, like Chlorophyll [35], which targets extremely memory-constrained power-efficient processors, Chipmunk [22], which targets programmable network switches, and Diospyros [48],[9] which generates vectorized programs for standalone digital signal processors (more powerful and general-purpose devices than the DSP units in FPGAs). These works demonstrate the utility of program synthesis for generating code that handles specific wrinkles in hardware designs, as does the use of program synthesis in Lakeroad to harness the programmability of FPGA DSPs.

Lakeroad is also related to past work in FPGA compilation and techmapping, much of which does not entreaty to support programmable DSPs with as much generality. ODIN [26] and ODIN-II [25] are used in *hard-block synthesis* for FPGAs, which is the task of mapping portions of hardware designs to specialized units (*hard blocks*) like multipliers. They operate purely over syntax (e.g., mapping `*` to a multiplier) and so are greatly limited in their ability to handle programmable DSPs. The ABC [14] logic synthesis tool is used to lower hardware designs into LUT and carry-chain configurations; it is related to Lakeroad in that it also uses constraint solvers to find configurations, though it is not general enough to handle a wide variety of programmable DSPs, unlike the program synthesis techniques used in Lakeroad. Note also that the use of configuration files in Lakeroad to abstract away details of the FPGA architecture was inspired by past work in FPGA compilation, including OpenFPGA [42] and the Verilog-to-Routing project (VTR) [38], both of which use abstract architecture descriptions to facilitate portability across designs, though these projects are limited in their support for DSPs. Library-Parameterized Models [3, 6] define generic interfaces for common primitives and are also similar to Lakeroad's primitive interfaces, though they are limited in their ability to represent configurable units like DSPs.

[9] Diospyros uses symbolic evaluation, which is related to program synthesis, to lift imperative programs for digital signal processors into a high-level mathematical representation that can then be used with the technique of equality saturation [44] to generate optimized code for the target devices. This is also distinct from the program synthesis techniques referenced elsewhere in this paper.

Virtual FPGA overlays [13, 27, 28] are another approach to improving the mapping of hardware designs to hardware. Overlays present a "virtual" FPGA architecture; each actual architecture must then define a mapping from virtual to actual primitives. This required translation is similar to Lakeroad's requirement on users to implement primitive interfaces in an architecture description, though it requires more user effort. The translation from virtual to actual architecture often comes with a steep resource and performance overhead.

7 Conclusion

This paper presents Lakeroad, a novel approach to FPGA technology mapping that leverages program synthesis techniques to provide stronger correctness and completeness guarantees than state-of-the-art tools. Because program synthesis tools can efficiently explore large search spaces, Lakeroad can find mappings of hardware designs to FPGA DSPs in more cases than state-of-the-art tools, often finding more efficient implementations in the process. With our techniques of semantics extraction from HDL and architecture-independent sketch templates, users must expend little manual effort to apply Lakeroad to a given FPGA architecture and extend it to handle further primitives. Moreover, our formalization of Lakeroad fosters greater confidence in its correctness. Lakeroad hence enables the extensible, efficient, and correct lowering of hardware designs to FPGAs, highlighting the effectiveness of program synthesis for FPGA technology mapping.

Acknowledgements

This work was funded by generous grants and awards from Intel, the U.S. Department of Energy (award number DE-SC0022081), and the NSF (grant numbers 1836724 and 1749570).

We would like to thank our anonymous reviewers for their constructive feedback. Thank you to Jonathan Balkind for serving as our shepherd. Thank you to those who contributed code to early versions of Lakeroad, including David Cao and Zihao Ye. Thank you to Jin Yang and his team at Intel. Thank you to Daniel Petrisko, Scott Davidson, Rachit Nigam, and Adrian Sampson for sharing their deep knowledge of the hardware design workflow. Thank you to Chandrakana Nandi for her enthusiasm and unwavering support. Thank you to Claire Xenia Wolf, Nina Engelhardt, Jannis Harder, and the YosysHQ team. Finally, thank you to the entire PLSE lab for their support and camaraderie.

A Artifact Appendix

A.1 Abstract

Our artifact consists of a zipfile containing the code for our evaluation. Running the evaluation code will reproduce all of the figures present in this paper, which artifact evaluators can validate against our published data. The evaluation code is comprised largely of the following files: documentation in a README, a Dockerfile to automatically set up the evaluation environment, the Lakeroad codebase, and the evaluation scripts themselves (a mix of Python and shell scripts). The evaluation should be run on an x86 machine running Linux (ideally Ubuntu). The evaluation benefits from many CPU cores. The evaluation requires at least 300GB of free space, mostly for installing proprietary hardware toolchains.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Program synthesis via Rosette. Hardware synthesis via traditional hardware toolchains.
- **Program:** Lakeroad, the Rosette-based hardware synthesis tool presented in this paper, plus Yosys, Xilinx Vivado, Lattice Diamond, and Intel Quartus, the baseline hardware synthesis tools we compare against.
- **Run-time environment:** Linux, ideally Ubuntu.
- **Hardware:** x86 CPU, ideally with many cores.
- **Output:** Images and CSV files representing this paper's figures and tables.
- **Experiments:** Each experiment is a single run of a hardware synthesis tool (either Lakeroad or one of our baseline tools). The entire experiment consists of thousands of these tool runs.
- **How much disk space required (approximately)?:** 300GB.
- **How much time is needed to prepare workflow (approximately)?:** 4 hours: 3 hours to set up proprietary hardware tools, 1 hour to build Docker image.
- **How much time is needed to complete experiments (approximately)?:** 2 to 10+ hours, depending on the number of cores. On our 64-core machine, the evaluation takes about 4 hours.
- **Publicly available?:** Yes, at https://github.com/uwsampl/lakeroad-evaluation and archived publicly on Zenodo, see DOI link below.
- **Code licenses (if publicly available)?:** MIT.
- **Workflow framework used?:** Python DoIt.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.10515833

A.3 Description

A.3.1 How to access.

We recommend downloading the zipped code repository from the DOI link above. The code can also be cloned from the GitHub repository linked above.

A.3.2 Hardware dependencies.

x86 CPU, preferably with many cores.

A.3.3 Software dependencies.

Linux-based OS, ideally Ubuntu.

A.4 Installation

Please refer to the README in the artifact. A more readable version of the README can be viewed on the GitHub repository, or by converting the README using a tool like Pandoc.

A.5 Experiment workflow

Please refer to the README in the artifact.

A.6 Evaluation and expected results

Please refer to the README in the artifact.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] Can not correctly infer "A*B+C" to DSP48E2. https://support.xilinx.com/s/question/0D54U00006AqPXFSA3/can-not-correctly-infer-abc-to-dsp48e2?language=en_US. Accessed: 2023-12-07.

[2] DSP48E2 inference for convolution/multiplication of 8-bit operands. https://support.xilinx.com/s/question/0D52E00006hpnGVSAY/dsp48e2-inference-for-convolutionmultiplication-of-8bit-operands?language=en_US. Accessed: 2023-12-07.

[3] Eia-is-103 : Library of parameterized modules (lpm).

[4] Inferring SIMD accumulator with Xilinx DSP48e2. https://old.reddit.com/r/FPGA/comments/tr9vzn/inferring_simd_accumulator_with_xilinx_dsp48e2/. Accessed: 2023-12-07.

[5] The simple theorem prover.

[6] Altera. Lpm quick reference guide.

[7] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1623–1638, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. *cvc5: A Versatile and Industrial-Strength SMT Solver*, pages 415–442. 01 2022.

[9] Andrew Becker, David Novo, and Paolo Ienne. Sketchilog: Sketching combinational circuits. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4, 2014.

[10] Gilbert Louis Bernstein and Jonathan Ragan-Kelley. What are the semantics of hardware? In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*, 2021.

[11] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. *SIGPLAN Not.*, 52(6):467–481, jun 2017.

[12] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–257, 2020.

[13] Alexander Brant and Guy GF Lemieux. Zuma: An open fpga overlay architecture. In *2012 IEEE 20th international symposium on field-programmable custom computing machines*, pages 93–96. IEEE, 2012.

[14] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[15] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[16] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Synthesizing instruction selection rewrite rules from rtl using smt. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*, page 139, 2022.

[17] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[18] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[19] Bruno Dutertre and Leonardo De Moura. The yices smt solver.

[20] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, March 2018.

[21] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010.

[22] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 150–160, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*, volume 4. NOW, August 2017.

[24] Yann Herklotz and John Wickerson. Finding and understanding bugs in fpga synthesis tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 277–287, 2020.

[25] Peter Jamieson, Kenneth B Kent, Farnaz Gharibian, and Lesley Shannon. Odin ii-an open-source verilog hdl synthesis tool for cad research. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 149–156. IEEE, 2010.

[26] Peter Jamieson and Jonathan Rose. A verilog rtl synthesis tool for heterogeneous fpgas. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 305–310. IEEE, 2005.

[27] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J Rossbach, and Eric Schkufza. Compiler-driven fpga virtualization with synergy. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 818–831, 2021.

[28] Roman L Lysecky, Kris Miller, Frank Vahid, and Kees A Vissers. Firm-core virtual fpga for just-in-time fpga compilation. In *FPGA*, page 271, 2005.

[29] Donald MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust.* The MIT Press, 09 2001.

[30] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. Improvements to technology mapping for lut-based fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):240–253, 2007.

[31] Aina Niemetz and Mathias Preiner. Bitwuzla at the smt-comp 2020. *arXiv preprint arXiv:2006.01621*, 2020.

[32] Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023.

[33] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

[34] Rishiyur Nikhil. Bluespec system verilog: Efficient, correct rtl from high level specifications. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '04, page 69–70, USA, 2004. IEEE Computer Society.

[35] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. *ACM SIGPLAN Notices*, 49(6):396–407, 2014.

[36] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. A formal foundation for symbolic evaluation with merging. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.

[37] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, 2005. http://www.eecs.berkeley.edu/~alanmi/abc.

[38] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. The vtr project: architecture and cad for fpgas from verilog to routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 77–86, 2012.

[39] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. A position on program synthesis for processor development. In *Workshop on Languages, Tools, and Techniques for Accelerator Design–LATTE 2022*, 2022.

[40] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. Loop rerolling for hardware decompilation. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[41] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[42] Xifan Tang, Edouard Giacomin, Aurélien Alacchi, Baudouin Chauviere, and Pierre-Emmanuel Gaillardon. Openfpga: An opensource framework enabling rapid prototyping of customizable fpgas. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 367–374. IEEE, 2019.

[43] Xifan Tang, Ganesh Gore, Grant Brown, and Pierre-Emmanuel Gaillardon. Taping out an fpga in 24 hours with openfpga: The sofa project. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 400–400. IEEE, 2021.

[44] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 264–276, New York, NY, USA, 2009. Association for Computing Machinery.

[45] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[46] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices*, 49(6):530–541, 2014.

[47] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing jit compilers for in-kernel dsls. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, page 564–586, Berlin, Heidelberg, 2020. Springer-Verlag.

[48] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 874–886, 2021.

[49] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. Reticle: a virtual machine for programming modern fpgas. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 756–771, 2021.

[50] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[51] Xilinx. Ultrascale architecture DSP slice user guide, 2021. https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp.