Formal Verification of High-Level Synthesis

Yann Herklotz, James D. Pollard, Nadesh Ramanathan, John Wickerson

Imperial College London

CHICAGO
SPLASH2021

Imperial College
London

Outline

What is High-Level Synthesis

Definition (High-Level Synthesis (HLS))

Translation of a high-level programming language such as C/C++ into a hardware description language (HDL) such as Verilog.

What is High-Level Synthesis

Definition (High-Level Synthesis (HLS))

Translation of a high-level programming language such as C/C++ into a hardware description language (HDL) such as Verilog.

Benefits of HLS

- **Usability**: Use software ecosystem.
- **Speed**: Quickly design hardware.

What is High-Level Synthesis

Definition (High-Level Synthesis (HLS))

Translation of a high-level programming language such as C/C++ into a hardware description language (HDL) such as Verilog.

Benefits of HLS

- **Usability**: Use software ecosystem.
- **Speed**: Quickly design hardware.

Trade-offs of HLS

- **Performance**: Requires automatic parallelisation.
- **Correctness**: Hard to verify generated HDL.

Motivation

High-level synthesis is often quite unreliable:

- Intel’s OpenCL could not be fuzzed because of too many issues (Lidbury et al. [2015]).

Motivation

High-level synthesis is often quite unreliable:

- Intel's OpenCL could not be fuzzed because of too many issues (Lidbury et al. [2015]).
- We fuzzed HLS tools and found they failed on **2.5%** of simple random test cases.

Imperial College London

Motivation

High-level synthesis is often quite unreliable:

- Intel's OpenCL could not be fuzzed because of too many issues (Lidbury et al. [2015]).
- We fuzzed HLS tools and found they failed on **2.5%** of simple random test cases.

Difficult to debug HLS tools:

- Simulation can take a long time.

Imperial College London

Motivation

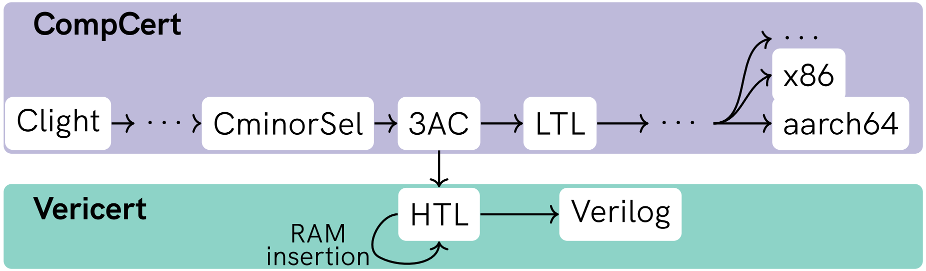
High-level synthesis is often quite unreliable:

- Intel's OpenCL could not be fuzzed because of too many issues (Lidbury et al. [2015]).
- We fuzzed HLS tools and found they failed on **2.5%** of simple random test cases.

Difficult to debug HLS tools:

- Simulation can take a long time.
- Correctness is important in hardware, testing is done at every level.

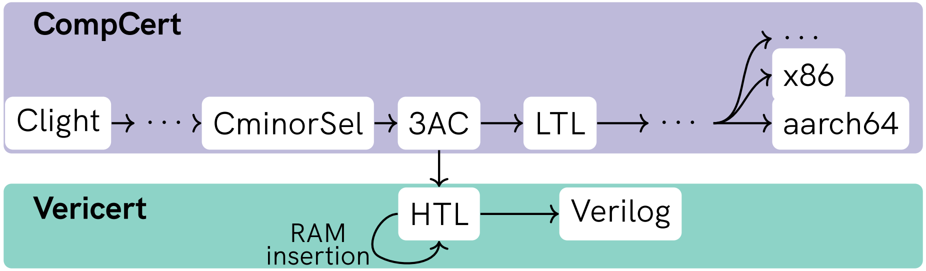
Solution



Use CompCert, a fully verified C compiler, and add an HLS backend.

Imperial College London

Solution



Current progress: fully verified HLS tool for a subset of C.
Support for: all **control flow**, **fixedpoint**, **non-recursive functions** and **local arrays/structs/unions**.

Imperial College London

Outline

Example

Verification

Imperial College London

Example: RTL

```
int main() {
    int x[2] = {3, 6};
    int i = 1;
    return x[i];
}
```

Example of a very simple program performing loads and stores.

Example: RTL

- **three address code (RTL)** instructions are represented as a control-flow graph (CFG).
- Each instruction links to the next one.

```
main() {
    x5 = 3
    int32[stack(0)] = x5
    x4 = 6
    int32[stack(4)] = x4
    x1 = 1
    x3 = stack(0) (int)
    x2 = int32[x3 + x1 * 4 + 0]
    return x2
}
```

Imperial College London

Example: HTL Overview

The representation of the **finite state-machine with datapath (FSMD)** is abstract and called **HTL**.

```
Definition datapath := PTree.t Verilog.stmnt.
Definition controllogic := PTree.t Verilog.stmnt.
```

Example: HTL Overview

The representation of the **finite state-machine with datapath (FSMD)** is abstract and called **HTL**.

```
Definition datapath := PTree.t Verilog.stmnt.
Definition controllogic := PTree.t Verilog.stmnt.

Record module: Type := mkmodule {
    mod_datapath: datapath; mod_controllogic: controllogic;
    mod_wf: map_well_formed mod_controllogic
             /\ map_well_formed mod_datapath;
    mod_reset: reg;
    mod_ram: ram_spec;
    ...
  }.
```

Imperial College London

Example: Translation (RTL $\rightarrow$ HTL)

Translation from **control-flow graph (CFG)** into a **finite state-machine with datapath (FSMD)**.

Example: Translation (RTL $\rightarrow$ HTL)

Translation from **control-flow graph (CFG)** into a **finite state-machine with datapath (FSMD)**.

- **Control-flow** is translated into a **finite state-machine**.

Example: Translation (RTL $\rightarrow$ HTL)

Translation from **control-flow graph (CFG)** into a **finite state-machine with datapath (FSMD)**.

- **Control-flow** is translated into a **finite state-machine**.
- Each **RTL instructions** translated into equivalent **Verilog statements**.

```
x3 = x3 + x5 + 0   ⟶   reg_3 <= {reg_3 + {reg_5 + 32'd0}}
```

Imperial College London

Example: Translation (RTL $\rightarrow$ HTL)

Translation from **control-flow graph (CFG)** into a **finite state-machine with datapath (FSMD)**.

- **Control-flow** is translated into a **finite state-machine**.
- Each **RTL instructions** translated into equivalent **Verilog statements**.
- Function **stack** implemented as **RAM**.

Example: Translation (RTL $\rightarrow$ HTL)

Translation from **control-flow graph (CFG)** into a **finite state-machine with datapath (FSMD)**.

- **Control-flow** is translated into a **finite state-machine**.
- Each **RTL instructions** translated into equivalent **Verilog statements**.
- Function **stack** implemented as **RAM**.
- Pointers for loads and stores translated to RAM addresses.

```
x5 + x1 * 4 + 0
  ⟶   {{{reg_5 + 32'd0} + {reg_1 * 32'd4}} / 32'd4}
```

Imperial College London

Example: Translation (RTL $\rightarrow$ HTL)

Translation from **control-flow graph (CFG)** into a **finite state-machine with datapath (FSMD)**.

- **Control-flow** is translated into a **finite state-machine**.
- Each **RTL instructions** translated into equivalent **Verilog statements**.
- Function **stack** implemented as **RAM**.
- Pointers for loads and stores translated to RAM addresses.
    - **Byte** addressed to **word** addressed.

```
x5 + x1 * 4 + 0
  ⟶   {{{reg_5 + 32'd0} + {reg_1 * 32'd4}} / 32'd4}
```

Imperial College London

Example: Translation (HTL $\rightarrow$ Verilog)

```verilog
module main(reset, clk, finish, return_val);
  input [0:0] reset, clk;
  output reg [0:0] finish = 0;
  output reg [31:0] return_val = 0;
  reg [31:0] reg_3 = 0, addr = 0, d_in = 0,
             reg_5 = 0, wr_en = 0,
             state = 0, reg_2 = 0,
             reg_4 = 0, d_out = 0, reg_1 = 0;
  reg [0:0] en = 0, u_en = 0;
  reg [31:0] stack [1:0];
  // RAM interface
  always @(negedge clk)
    if ({u_en != en}) begin
      if (wr_en) stack[addr] <= d_in;
      else d_out <= stack[addr];
      en <= u_en;
    end
```

- Finally, translate the FSMD into Verilog.

Imperial College London

Example: Translation (HTL $\rightarrow$ Verilog)

```
module main(reset, clk, finish, return_val);
  input [0:0] reset, clk;
  output reg [0:0] finish = 0;
  output reg [31:0] return_val = 0;
  reg [31:0] reg_3 = 0, addr = 0, d_in = 0,
             reg_5 = 0, wr_en = 0,
             state = 0, reg_2 = 0,
             reg_4 = 0, d_out = 0, reg_1 = 0;
  reg [0:0] en = 0, u_en = 0;
  reg [31:0] stack [1:0];
  // RAM interface
  always @(negedge clk)
    if ({u_en != en}) begin
      if (wr_en) stack[addr] <= d_in;
      else d_out <= stack[addr];
      en <= u_en;
    end
```

- Finally, translate the FSMD into Verilog.
- This includes a RAM interface.

Imperial College London

Example: Translation (HTL $\rightarrow$ Verilog)

```
// Data-path
always @(posedge clk)
  case (state)
    32'd11: reg_2 <= d_out;
    32'd8: reg_5 <= 32'd3;
    32'd7: begin
      u_en <= ( ~ u_en); wr_en <= 32'd1;
      d_in <= reg_5; addr <= 32'd0;
    end
    32'd6: reg_4 <= 32'd6;
    32'd5: begin
      u_en <= ( ~ u_en); wr_en <= 32'd1;
      d_in <= reg_4; addr <= 32'd1;
    end
    32'd4: reg_1 <= 32'd1;
    32'd3: reg_3 <= 32'd0;
    32'd2: begin
      u_en <= ( ~ u_en); wr_en <= 32'd0;
      addr <= {{{reg_3 + 32'd0} + {reg_1 * 32'd4}} / 32'd4};
    end
    32'd1: begin finish = 32'd1; return_val = reg_2; end
    default: ;
  endcase
```

- Finally, translate the FSMD into Verilog.
- This includes a RAM interface.
- Data path is translated into a case statement.

Imperial College London

Example: Translation (HTL $\rightarrow$ Verilog)

```
// Data-path
always @(posedge clk)
  case (state)
    32'd11: reg_2 <= d_out;
    32'd8: reg_5 <= 32'd3;
    32'd7: begin
      u_en <= ( ~ u_en); wr_en <= 32'd1;
      d_in <= reg_5; addr <= 32'd0;
    end
    32'd6: reg_4 <= 32'd6;
    32'd5: begin
      u_en <= ( ~ u_en); wr_en <= 32'd1;
      d_in <= reg_4; addr <= 32'd1;
    end
    32'd4: reg_1 <= 32'd1;
    32'd3: reg_3 <= 32'd0;
    32'd2: begin
      u_en <= ( ~ u_en); wr_en <= 32'd0;
      addr <= {{{reg_3 + 32'd0} + {reg_1 * 32'd4}} / 32'd4};
    end
    32'd1: begin finish = 32'd1; return_val = reg_2; end
    default: ;
  endcase
```

- Finally, translate the FSMD into Verilog.
- This includes a RAM interface.
- Data path is translated into a case statement.
- Ram loads and stores automatically turn off RAM.

Imperial College London

Example: Translation (HTL $\rightarrow$ Verilog)

```
  // Control logic
  always @(posedge clk)
    if ({reset == 32'd1}) state <= 32'd8;
    else case (state)
           32'd11: state <= 32'd1;          32'd4: state <= 32'd3;
           32'd8: state <= 32'd7;           32'd3: state <= 32'd2;
           32'd7: state <= 32'd6;           32'd2: state <= 32'd11;
           32'd6: state <= 32'd5;           32'd1: ;
           32'd5: state <= 32'd4;           default: ;
         endcase
endmodule
```

- Finally, translate the FSMD into Verilog.
- This includes a RAM interface.
- Data path is translated into a case statement.
- Ram loads and stores automatically turn off RAM.
- Control logic is translated into another case statement with a reset.

Imperial College London

Outline

Example

Verification

Imperial College London

Verilog Semantics (Adapted from Lööw et al. (2019))

$$\text{Module}\ \frac{(\Gamma, \epsilon, \vec{m}) \downarrow_{\mathsf{module}^+} (\Gamma', \Delta') \qquad (\Gamma' \mathbin{//} \Delta', \epsilon, \vec{m}) \downarrow_{\mathsf{module}^-} (\Gamma'', \Delta'')}{(\Gamma, \texttt{module main(...);}\ \vec{m}\ \texttt{endmodule}) \downarrow_{\mathsf{program}} (\Gamma'' \mathbin{//} \Delta'')}$$

- Two separate association maps: current (Γ) and next (Δ).

Imperial College London

Verilog Semantics (Adapted from Lööw et al. (2019))

Module

$$\frac{(\Gamma, \epsilon, \vec{m}) \downarrow_{\text{module}^+} (\Gamma', \Delta') \qquad (\Gamma' \mathbin{//} \Delta', \epsilon, \vec{m}) \downarrow_{\text{module}^-} (\Gamma'', \Delta'')}{(\Gamma, \texttt{module main(...);}\ \vec{m}\ \texttt{endmodule}) \downarrow_{\text{program}} (\Gamma'' \mathbin{//} \Delta'')}$$

- Two separate association maps: current (Γ) and next (Δ).
- Maps are merged at the end of the clock cycle.

Imperial College London

How do we prove the HLS tool correct?

- We have an **algorithm** describing the **translation**.
- Have to **prove** that it does not change **behaviour** with respect to our language semantics.

Imperial College London

How do we prove the HLS tool correct?

- We have an **algorithm** describing the **translation**.
- Have to **prove** that it does not change **behaviour** with respect to our language semantics.

Behaviour	Guarantee
Converging	Means a result is obtained, Verilog and C results must be equal.
Diverging	C is in an infinite loop, Verilog must execute indefinitely.
Wrong	Such as undefined behaviour, no guarantees need to be shown.

Imperial College London

How do we prove the HLS tool correct?

- We have an **algorithm** describing the **translation**.
- Have to **prove** that it does not change **behaviour** with respect to our language semantics.

Theorem (Main Backward Simulation)

$$\forall C, V, B, \quad \mathit{HLS}(C) = \mathit{OK}(V) \wedge \mathit{Safe}(C) \implies (V \Downarrow B \implies C \Downarrow B).$$

where

$$\mathit{Safe}(C) : \ \forall B,\ C \Downarrow B \implies B \in \mathsf{Safe}$$

Imperial College London

How do we prove the HLS tool correct?

- We have an **algorithm** describing the **translation**.
- Have to **prove** that it does not change **behaviour** with respect to our language semantics.

Theorem (Forward Simulation)

$$(\forall C, V, B \in \mathtt{Safe},\quad \mathtt{HLS}(C) = \mathtt{OK}(V) \land C \Downarrow B \implies V \Downarrow B)$$
$$\land\, (\forall V, B_1, B_2,\quad V \Downarrow B_1 \land V \Downarrow B_2 \implies B_1 = B_2).$$

Imperial College London

RTL $\rightarrow$ HTL: Build a Specification

Assuming $\texttt{HLS}(C) = \texttt{OK}(V)$ requires reasoning about implementation details.

RTL $\rightarrow$ HTL: Build a Specification

Assuming $\texttt{HLS}(C) = \texttt{OK}(V)$ requires reasoning about implementation details.

Instead we build a model of the translation which we can use.

$$\forall C, V, \quad \texttt{HLS}(C) = \texttt{OK}(V) \rightarrow \texttt{tr\_hls}\ \ C\ \ V.$$

RTL $\rightarrow$ HTL: Build a Specification

Assuming $\texttt{HLS}(C) = \texttt{OK}(V)$ requires reasoning about implementation details.

Instead we build a model of the translation which we can use.

$$\forall C, V, \quad \texttt{HLS}(C) = \texttt{OK}(V) \rightarrow \texttt{tr\_hls}\ \ C\ \ V.$$

Example (RTL to HTL operator conversion)

Iop

$$\frac{\texttt{tr\_op}\ \ op\ \vec{a} = \texttt{OK}\ \ e}{\texttt{tr\_instr}\ fin\ rtrn\ \sigma\ \ stk\ (\texttt{Iop}\ \ op\ \vec{a}\ d\ n)\ (d \Leftarrow e)\ (\sigma \Leftarrow n)}$$

Imperial College London

RTL $\rightarrow$ HTL: Prove Forward Simulation

$$\begin{tikzcd}[column sep=huge, row sep=huge]
S_1 \arrow[r, dash, "\sim", "\texttt{match\_states}"'] \arrow[d] & R_1 \arrow[d, dashed, "+"'] \\
S_2 \arrow[r, dash, dashed, "\sim", "\texttt{match\_states}"'] & R_2
\end{tikzcd}$$

`match_states` defined as:

$$\mathcal{I} \wedge R \leq \Gamma \wedge M \leq \Gamma!stk \wedge pc = \Gamma!\sigma$$

Prove the simulation diagram correct:

RTL $\rightarrow$ HTL: Prove Forward Simulation

$$\begin{tikzcd}
S_1 \arrow[r, dash, "\sim", "\texttt{match\_states}"'] \arrow[d] & R_1 \arrow[d, dashed, "+"'] \\
S_2 \arrow[r, dash, dashed, "\sim", "\texttt{match\_states}"'] & R_2
\end{tikzcd}$$

`match_states` defined as:

$$\mathcal{I} \wedge R \leq \Gamma \wedge M \leq \Gamma!stk \wedge pc = \Gamma!\sigma$$

Prove the simulation diagram correct:

- Assuming an initial match between the RTL state S_1 and Verilog state R_1,

RTL $\rightarrow$ HTL: Prove Forward Simulation

$$\begin{tikzcd}
S_1 \arrow[rr, dash, "\sim", "\texttt{match\_states}"'] \arrow[d] & & R_1 \arrow[d, dashed, "+"'] \\
S_2 \arrow[rr, dash, dashed, "\sim", "\texttt{match\_states}"'] & & R_2
\end{tikzcd}$$

`match_states` defined as:

$$\mathcal{I} \land R \leq \Gamma \land M \leq \Gamma!stk \land pc = \Gamma!\sigma$$

Prove the simulation diagram correct:

- Assuming an initial match between the RTL state S_1 and Verilog state R_1,
- there exists 1 or more steps in Verilog,

Imperial College London

RTL $\rightarrow$ HTL: Prove Forward Simulation

$$\begin{tikzcd}
S_1 \arrow[r, dash, "\sim", "\texttt{match\_states}"'] \arrow[d] & R_1 \arrow[d, dashed, "+"'] \\
S_2 \arrow[r, dash, dashed, "\sim", "\texttt{match\_states}"'] & R_2
\end{tikzcd}$$

`match_states` defined as:

$$\mathcal{I} \wedge R \leq \Gamma \wedge M \leq \Gamma!stk \wedge pc = \Gamma!\sigma$$

Prove the simulation diagram correct:

- Assuming an initial match between the RTL state S_1 and Verilog state R_1,
- there exists 1 or more steps in Verilog,
- such that after 1 step in RTL, the resulting states match.

Results

Performance Results

- Ran on 27/30 PolyBench/C benchmarks and compared to LegUp.

Results

Performance Results

- Ran on 27/30 PolyBench/C benchmarks and compared to LegUp.
- $27\times$ slower and $1.1\times$ larger.

Results

Performance Results

- Ran on 27/30 PolyBench/C benchmarks and compared to LegUp.
- 27$\times$ slower and 1.1$\times$ larger.
- Ran on PolyBench/C with divisions replaced by iterative shifting.

Results

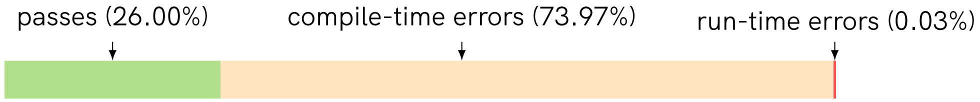
Performance Results

- Ran on 27/30 PolyBench/C benchmarks and compared to LegUp.
- 27$\times$ slower and 1.1$\times$ larger.
- Ran on PolyBench/C with divisions replaced by iterative shifting.
- 2$\times$ slower (on par with unoptimised LegUp).

Fuzzing Vericert with Csmith

Fuzzed Vericert with Csmith to check correctness theorem.

- One bug was found in the pretty printing.
- Many compile-time errors are expected.
- Mainly rejected because of wrong size.



Imperial College London

Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- Base translation **proven correct** by proving translation of CFG into FSMD.

Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- Base translation **proven correct** by proving translation of CFG into FSMD.
- Small optimisations implemented such as **Ram Inference**.

Imperial College London

Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- Base translation **proven correct** by proving translation of CFG into FSMD.
- Small optimisations implemented such as **Ram Inference**.
- Performance without divisions comparable to LegUp without optimisations.

Imperial College London

Thank you

Documentation

https://vericert.ymhg.org

GitHub

https://github.com/ymherklotz/vericert

OOPSLA'21 Preprint

https://ymhg.org/papers/fvhls_oopsla21.pdf

Imperial College London

References

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’15, pages 65–76, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737986. URL `https://doi.org/10.1145/2737924.2737986`.

Imperial College London