Loop Rerolling for Hardware Decompilation

ZACHARY D. SISCO, University of California, Santa Barbara, USA
JONATHAN BALKIND, University of California, Santa Barbara, USA
TIMOTHY SHERWOOD, University of California, Santa Barbara, USA
BEN HARDEKOPF, University of California, Santa Barbara, USA

We introduce the new problem of *hardware decompilation*. Analogous to software decompilation, hardware decompilation is about analyzing a low-level artifact—in this case a *netlist*, i.e., a graph of wires and logical gates representing a digital circuit—in order to recover higher-level programming abstractions, and using those abstractions to generate code written in a hardware description language (HDL). The overall problem of hardware decompilation requires a number of pieces. In this paper we focus on one specific piece of the puzzle: a technique we call *hardware loop rerolling*. Hardware loop rerolling leverages clone detection and program synthesis techniques to identify repeated logic in netlists (such as would be synthesized from loops in the original HDL code) and *reroll* them into syntactic loops in the recovered HDL code. We evaluate hardware loop rerolling for hardware decompilation over a set of hardware design benchmarks written in the PyRTL HDL and industry standard SystemVerilog. Our implementation identifies and rerolls loops in 52 out of 53 of the netlists in our benchmark suite, and we show three examples of how hardware decompilation can provide concrete benefits: transpilation between HDLs, faster simulation times over netlists (with mean speedup of 6x), and artifact compaction (39% smaller on average).

CCS Concepts: • **Hardware → Hardware description languages and compilation**; *Software tools for EDA*.

Additional Key Words and Phrases: hardware decompilation, program synthesis, loop rerolling

ACM Reference Format:
Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. 2023. Loop Rerolling for Hardware Decompilation. *Proc. ACM Program. Lang.* 7, PLDI, Article 123 (June 2023), 23 pages. https://doi.org/10.1145/3591237

1 INTRODUCTION

Hardware description languages (HDLs) are a key tool in the hardware development process. HDLs provide high-level programmatic abstractions for designing, simulating, verifying, and synthesizing hardware. Synthesizing HDL code generates a layout of wires and logical gates represented as a graph called a *netlist*. After synthesis, the resulting netlist loses many of the high-level details from the HDL code such as loops, functions, and modules. The netlist is also considerably larger than the HDL code that generates it.

This paper introduces a new research problem: *hardware decompilation*, that is, transforming a netlist into a semantically identical HDL program at a higher level of abstraction. The idea is analogous to software decompilation, wherein an executable binary is lifted back to source code in a high-level programming language, but targets netlists (rather than executables) and HDLs (rather than general-purpose programming languages).

Authors' addresses: Zachary D. Sisco, University of California, Santa Barbara, USA, zsisco@ucsb.edu; Jonathan Balkind, University of California, Santa Barbara, USA, jbalkind@ucsb.edu; Timothy Sherwood, University of California, Santa Barbara, USA, sherwood@cs.ucsb.edu; Ben Hardekopf, University of California, Santa Barbara, USA, benh@cs.ucsb.edu.

2475-1421/2023/6-ART123
https://doi.org/10.1145/3591237

1.1 Motivating Hardware Decompilation

If we had a solution to the hardware decompilation problem, there are a number of applications that would benefit hardware designers. This paper only begins to explore hardware decompilation and its applications, but our work shows that these ideas have potential. Here is a non-exhaustive list of such applications:

Transpilation Between HDLs. A netlist serves as a common target for all HDLs (e.g., SystemVerilog, Chisel [Izraelevitz et al. 2017], PyRTL [Clow et al. 2017], etc). When decompiling a netlist, the target HDL does not need to be the same as the original HDL from which the netlist was generated. Thus, hardware decompilation enables an automated translation between two different HDLs: take the original HDL code, synthesize it into a netlist, then decompile that netlist into the second HDL.

Speeding Up Simulation Time. Simulation is a huge part of the hardware design process, both for exploration and validation of designs. Designs are repeatedly simulated, edited, and simulated again. However, simulating netlists can be significantly slower than simulating HDL source code [Beamer 2020]; thus, being able to recover HDL code from a netlist can help improve simulation time.

Artifact Compaction. A netlist can be significantly larger than the HDL code that generated it [Ganai and Kuehlmann 2000]. One way to compress a netlist design (above and beyond using standard compression algorithms) would be to recover the much smaller, but equivalent, HDL-level code.

The work presented in this paper shows the feasibility of these three applications: transpilation (we convert designs between SystemVerilog and PyRTL HDL code); simulation time (we demonstrate that simulation of recovered HDL code is faster than the corresponding netlist, with mean speedup of 6x); and artifact compaction (we demonstrate that the recovered HDL takes significantly less space than the corresponding netlist representation, with mean compaction of 39% across our benchmark suite). Further, a hardware decompiler opens the way to other potential applications:

Understanding and Analysis. In industry hardware development, it is common to use third-party component libraries (also known as Intellectual Property or IP catalogs). These components are provided only as netlists, without the higher-level HDL source code. Creating designs using these components makes human analysis and automated static analysis difficult; being able to recover high-level HDL would greatly benefit such efforts. In this way, a hardware decompiler enables security and verification analyses designed for higher-level HDL code but over a decompiled netlist (this is one of the main motivators for software decompilation as well).

Propagating Netlist Edits Back to HDL. A common occurrence in hardware design is to synthesize HDL code to a netlist and then discover that the resulting design needs to be tweaked for various reasons (e.g., physical layout or timing closure). Given a change made directly to the netlist, it can be extremely difficult to reason about what specific parts of the original HDL code would need to be modified, and in what specific way, in order to ensure that the updated HDL would then generate the desired new netlist. Using hardware decompilation, that process can be entirely automated.

In a departure from the software decompilation analogy, hardware decompilation has unique value during the *design* process, not just for reverse engineering or post-design analysis. For instance, hardware synthesis and backend tools often mangle the semantics of the HDL code, producing a netlist that is not semantically equivalent to the original design. Hardware designers then need to run simulations and logic equivalence checks over the netlist for functionality and correctness verification. Thus, hardware decompilation is a valuable tool during this design phase for speeding up simulation time by lifting the netlist to a more compact and higher-level representation.

```
module ripple_carry_adder #(parameter N)
 (input [N-1:0] a, input [N-1:0] b, output logic
     [N-1:0] sum);
 logic cin, cout;
 always_comb begin
   cin = 1'b0;
   for (int i=0; i < N; i++) begin
     sum[i] = a[i] ^ b[i] ^ cin;
     cout = a[i] & b[i] | a[i] & cin | b[i] & cin;
     cin = cout;
   end
 end
endmodule
```

```
module accumulator (input [3:0] x, input clk, output
     reg [3:0] acc);
  logic [3:0] sum;
  ripple_carry_adder #(.N(4)) adder(acc, x, sum);
  always @(posedge clk) begin
    acc <= sum;
  end
endmodule
```

Fig. 1. An accumulator circuit instantiated with a 4-bit ripple-carry adder written in SystemVerilog.

1.2 Hardware Loop Rerolling

There are a number of sub-problems that need to be solved to completely translate all aspects of a netlist back to idiomatic HDL, namely identifying and recovering a range of abstractions such as: loops; procedures; modules; protocol interfaces; and clean divisions between control and data-path logic. We do not solve all of these sub-problems in this work; instead we identify one of them as a stepping stone towards solving the others. Our specific focus in this paper is on recognizing repeated logic in netlists and decompiling them into loops in higher-level HDL code. We call this process *hardware loop rerolling*, as it mirrors an analogous operation seen in software compilers at the source and binary level [Ge et al. 2022; Hu et al. 2016; Rocha et al. 2022; Stiff and Vahid 2005; Su et al. 1984, 1986]. However, loop rerolling for a hardware decompiler occurs in the context of hardware design, where the execution model differs significantly from software. We discuss the differences between hardware loop rerolling and software loop rerolling in Section 7. For brevity, henceforth in the paper we use "loop rerolling" to refer specifically to hardware loop rerolling.

In the original HDL code that synthesized the netlist, repeated logic is syntactically expressed as loops (or recursion in the case of functional HDLs [Bjesse et al. 1998; Gill et al. 2010; Mycroft and Sharp 2003; O'Donnell 2006]). For example, Figure 1 presents code for an n-bit ripple-carry adder written in SystemVerilog. To generalize this function to arbitrary n-bit designs, the code uses a `for`-loop parameterized over the length, or bitwidth, of the wire vectors. The body of the `for`-loop generates a repeated pattern of add and carry operations which compute the sum of each bit and push the carry-out bit forward to the next iteration.

During hardware synthesis, loops in the HDL code are completely unrolled in the resulting netlist. The `accumulator` module in Figure 1 parameterizes the ripple-carry adder over wire vectors of width 4. When this accumulator circuit is synthesized, it results in the netlist found in Figure 2. Upon closer inspection, we find that the `for`-loop in Figure 1 is *unrolled* in the resulting netlist in Figure 2. There are four repetitions of the same set of `xor`, `and`, and `or` operations with each of the four bits of the input `x` and register acc. Each add and carry bit computation feeds into the proceeding one, starting from the zeroth bit to the third bit, until they are concatenated together and connected to the output register. Our goal, then, is to transform the netlist in Figure 2 into HDL code similar (though not necessarily identical) to that in Figure 1.

We break the loop rerolling problem into two major subproblems, and for each subproblem we adapt a different existing programming languages technique to create a solution. The first subproblem is *loop identification*, i.e., analyzing the netlist to detect potential candidates for loops. We leverage techniques from software clone detection, applying them to netlists instead of source code text or abstract syntax trees [Baker 1995; Kamiya et al. 2002]. Once we have identified a candidate, the second subproblem is the actual *loop rerolling* itself, which requires reasoning about

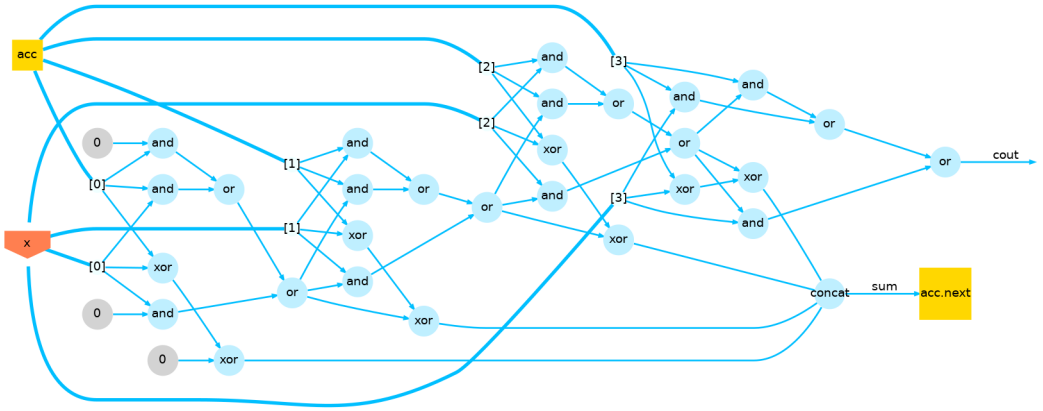


Fig. 2. Graph representation of a netlist for a 4-bit accumulator synthesized from the HDL code in Figure 1. We use acc.next to denote that the register acc receives the value and is updated in the *next* cycle.

pre-, post-, and intra-loop dependencies that need to exist in the generated code. We leverage techniques from solver-based program synthesis [Solar-Lezama 2013] (not to be confused with hardware synthesis) in order to generate semantically equivalent looping HDL code.

1.3 Contributions

The contributions of this paper are:

- We introduce the new research problem of hardware decompilation, focusing specifically in this paper on recovering loops in hardware designs.
- We describe a technique to identify candidate slices of a netlist that are suitable for lifting up as an HDL loop, based on software clone detection techniques (Section 3).
- We describe a technique to take a netlist slice and synthesize semantically equivalent looping HDL code, based on program synthesis techniques (Sections 4 and 5).
- We implement our techniques[1] and evaluate their effectiveness on a benchmark suite of SystemVerilog and PyRTL hardware designs (Section 6). Our evaluation examines the potential of hardware decompilation for transpilation, fast simulation, and artifact compaction.

2 THE MAKI INTERMEDIATE LANGUAGE

We present a bespoke intermediate language called Maki that is the common connecting format for each phase of our decompilation technique: the netlist is transformed into Maki code, each phase of decompilation operates on Maki code, and the final result is transliterated into HDL code. The purpose of Maki is to provide abstractions that sit between the low-level world of a netlist and the high-level world of a full-featured HDL. As such, it should be able to completely specify a netlist but also contain abstractions such as loops and arrays. This multi-level representation allows us to represent both low-level and high-level code in the same program representation.

Figure 3 describes the grammar for Maki. A program in Maki starts with three components: the input wire vectors (*in*), the output wire vectors (*out*), registers (*reg*), and a series of statements that describe the netlist ($stmt^+$). The input and output wire vector and register declarations are lists of pairs, with the first element being a variable identifier and the second element being the bitwidth of that wire vector (i.e., the length of the bit-vector that the wire can carry). Note that we

[1]Available as a free and open-source artifact: https://doi.org/10.5281/zenodo.7686503

$$
\begin{aligned}
\text{Netlist} &::= in\ out\ stmt^{+} \\
in \in \text{Input} &::= \textbf{(input}\ (name\ width,)^{*}\textbf{)} \\
out \in \text{Output} &::= \textbf{(output}\ (name\ width,)^{*}\textbf{)} \\
reg \in \text{Registers} &::= \textbf{(registers}\ (name\ width,)^{*}\textbf{)} \\
stmt \in \text{Statement} &::= var := (wexp \mid aexp) \mid \textbf{for-range}\ index, range\{stmt^{+}\} \\
wire \in \text{WireVector} &::= name \mid \textbf{const}\ value\ width \\
wexp \in \text{WireExpression} &::= wire \mid \texttt{AND}\ wexp\ wexp \mid \texttt{OR}\ wexp\ wexp \mid \texttt{NOT}\ wexp \mid \texttt{XOR}\ wexp\ wexp \\
&\quad \mid \textbf{mux}\ wexp\ wexp\ wexp \mid \textbf{concat (}wexp^{+}\textbf{)} \mid \textbf{select}\ wexp\ \textbf{(}aexp \mid nexp\textbf{)} \\
aexp \in \text{ArrayExpression} &::= \textbf{array-create}\ length \mid \textbf{array-store}\ nexp\ var \mid \textbf{array-ref}\ var\ nexp \\
nexp \in \text{NumExpression} &::= var \mid num \mid (+ \mid - \mid * \mid \div \mid \%)\ nexp\ nexp \\
index, name, var &\in \text{Identifier} \\
length, num, range, value, width &\in \text{Integer}
\end{aligned}
$$

Fig. 3. The grammar for Maki.

use the term *wire vector* to distinguish a bit-vector typed variable in a Maki program, as opposed to a single-bit *wire* in a netlist. Additionally, for expository purposes, the version of Maki presented here only describes a subset of HDL features. The full Maki language and our implementation supports sequential features including memory, and we include both sequential and combinational hardware designs in our evaluation.

A variable in Maki is one of three types: wire vectors (bit-vector of a set length), integers, or arrays of wire vectors. Maki has two kinds of statements: variable definitions (:=), and loops (**for-range**). A variable definition $var := (wexp \mid aexp)$ creates and binds a new wire vector or array expression to a variable identifier, deriving its value from the right-hand side expression. A loop defines an integer loop-counter variable *index* initialized to zero, a loop bound *range*, and a sequence of statements for the loop body.

The right-hand side of a variable definition can be either a wire expression *wexp* or an array expression *aexp*. A wire expression denotes the operations common for describing digital circuits (logical, arithmetic, bit select, wire concatenation). An array expression denotes array declarations, as well as reading from and writing to arrays. Array variables do not directly represent a specific hardware component, but are used to store accumulated results of wire expressions (e.g., storing each result of a one-bit add operation in a loop).

Figure 4 presents a selection of big-step structural operational semantics for Maki. There are four main types of rules in the presentation of the semantics: (1) rules for NumExpressions (NumExpOp), (2) rules for WireExpressions (WireExpOp, Mux0, Mux1), (3) rules for ArrayExpressions (ArrayRef, ArrayStore), and (4) rules for Statements. For space, we omit rules for NumExpression, WireExpression, and ArrayExpression. The more interesting rules are around Registers. For sequential elements, such as registers, there is a special store σ_{Reg} which holds updates to sequential elements until the end of one execution step. The rule StepCycle describes how Maki evaluates one step and handles register elements. Statements $s_1 \ldots s_n$ evaluate all combinational expressions, then $r_1 \ldots r_m$ update the register store σ_{Reg}. After evaluating all statements, the updates in σ_{Reg} are merged into the primary store σ so that the registers have the updated values ready for the next step.

Note that Maki programs describe one cycle of hardware execution (mapping from the state present at the beginning of the cycle to the state present at the end of the cycle) and are guaranteed to terminate. Specifically, all ForRange loops are finite and their range is known a priori (from the number of repetitions found in the netlist).

$$\frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)}\ \textsc{Lookup} \qquad \frac{\langle x_1, \sigma\rangle \Downarrow_n n_1 \quad \langle x_2, \sigma\rangle \Downarrow_n n_2}{\langle x_1 \oplus_n x_2, \sigma\rangle \Downarrow_n (n_1 \oplus_n n_2)}\ \textsc{NumExpOp} \qquad \frac{\langle w_1, \sigma\rangle \Downarrow_w v_1 \quad \langle w_2, \sigma\rangle \Downarrow_w v_2}{\langle w_1 \oplus_w w_2, \sigma\rangle \Downarrow_w (v_1 \oplus_w v_2)}\ \textsc{WireExpBinop}$$

$$\frac{\langle w_c, \sigma\rangle \Downarrow_w 0 \quad \langle w_0, \sigma\rangle \Downarrow_w v_0}{\langle \textbf{mux}\ w_c\ w_0\ w_1, \sigma\rangle \Downarrow_w v_0}\ \textsc{Mux0} \qquad \frac{\langle w_c, \sigma\rangle \Downarrow_w 1 \quad \langle w_1, \sigma\rangle \Downarrow_w v_1}{\langle \textbf{mux}\ w_c\ w_0\ w_1, \sigma\rangle \Downarrow_w v_1}\ \textsc{Mux1}$$

$$\frac{\langle e, \sigma\rangle \Downarrow v}{\langle x := e, \sigma\rangle \Downarrow \sigma[x \mapsto v]}\ \textsc{Define} \qquad \frac{r \in \text{Registers} \quad \langle e, \sigma\rangle \Downarrow_w v}{\langle r := e, \sigma, \sigma_{Reg}\rangle \Downarrow_r \sigma_{Reg}[r \mapsto v]}\ \textsc{RegUpdate}$$

$$\frac{\langle x, \sigma\rangle \Downarrow_a a \quad \langle n, \sigma\rangle \Downarrow_n n'}{\langle \textbf{array-ref}\ x\ n, \sigma\rangle \Downarrow_a a[n']}\ \textsc{ArrayRef} \qquad \frac{\langle x_1, \sigma\rangle \Downarrow_a a \quad \langle e, \sigma\rangle \Downarrow_n i \quad \langle x_2, \sigma\rangle \Downarrow_w v}{\langle x_1 := \textbf{array-store}\ e\ x_2, \sigma\rangle \Downarrow \sigma[a[i \mapsto v]]}\ \textsc{ArrayStore}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \sigma_1 \quad \langle s_2, \sigma_1\rangle \Downarrow \sigma_2}{\langle s_1\ s_2, \sigma\rangle \Downarrow \sigma_2}\ \textsc{StmtSeq} \qquad \frac{\begin{array}{c}\langle i := 0, \sigma\rangle \Downarrow \sigma' \quad \langle s_1 \ldots s_n, \sigma'\rangle \Downarrow \sigma_1 \\ \langle i := i + 1, \sigma_1\rangle \Downarrow \sigma'_1 \ \ldots \ \langle s_1 \ldots s_n, \sigma'_{r-1}\rangle \Downarrow \sigma_r\end{array}}{\langle \textbf{for-range}\ i\ r\ s_1 \ldots s_n, \sigma\rangle \Downarrow \sigma_r}\ \textsc{ForRange}$$

$$\frac{\begin{array}{c}\langle s_1, \sigma\rangle \Downarrow \sigma_1\ \ldots\ \langle s_n, \sigma_{n-1}\rangle \Downarrow \sigma_n\ \ldots \\ \langle r_1, \sigma_n, \sigma_{Reg}\rangle \Downarrow_r \sigma^1_{Reg}\ \ldots\ \langle r_m, \sigma_n, \sigma^{m-1}_{Reg}\rangle \Downarrow_r \sigma^m_{Reg}\ \ldots\end{array}}{\langle s1 \ldots s_n\ r_1 \ldots r_m, \sigma, \sigma_{Reg}\rangle \Downarrow \sigma_n[\forall_{x \in \sigma^m_{Reg}} x \mapsto \sigma^m_{Reg}(x)]}\ \textsc{StepCycle}$$

Fig. 4. A selection of big-step structural operational semantics for Maki. The relations $\Downarrow_n$, $\Downarrow_w$, $\Downarrow_a$, and $\Downarrow_r$ are expressly for evaluating NumExpression, WireExpression, ArrayExpression, and register updates, respectively. The primary environment is σ, a store mapping variable identifiers to values. σ_{Reg} is a special store holding updates to sequential elements.

Translating a Netlist to Maki. We translate a netlist to Maki by performing a topological sort over the netlist, starting from the input wires. This sort linearizes the netlist graph in a way that gives the same ordering for the same netlist (i.e., it is deterministic), and, in practice, groups related operations together. Each gate (node in the graph) is translated to a Maki wire vector variable definition by making the left-hand side the outgoing edge (the wire vector being defined), while the right-hand side becomes a wire expression. We translate the wire expression according to the wire operation and its arguments (the incoming edges). The input and output wires are the initial wire vector variables. All other edges in the graph become intermediate wire vector variables that are defined exactly once. The result of the translation is a Maki program in SSA form. Figure 5 shows Maki code for the 4-bit accumulator netlist from Figure 2 after linearization. Note that the initial translation of the netlist to Maki only contains the low-level features of Maki (i.e., only wire expressions defining wire vector variables). The high-level features, like loops and arrays, will come from decompilation, at which point the design may not be in SSA form.

Linearizing a netlist is efficient and works well in practice; a topological sort tends to order related wires and operations together. However, it is possible a linearization may disguise some repeated logic if the netlist is linearized differently for different repetitions. Another possible approach without this limitation would be to operate on the netlist graph directly. However, detecting repeated logic would amount to detecting repeated subgraph isomorphisms, which is an NP-complete problem and expensive in practice. We choose to linearize the code in order to leverage efficient techniques from software clone detection.

```
(input x 4)
(register acc 4)
t3 := const 0 1
t8 := select acc (0)
t9 := select x (0)
t10 := XOR t8 t9
t11 := XOR t10 t3
t12 := AND t8 t9
t13 := AND t8 t3
t14 := OR t12 t13
t15 := AND t9 t3
t16 := OR t14 t15
t17 := select acc (1)
t18 := select x (1)
t19 := XOR t17 t18
t20 := XOR t19 t16
t21 := AND t17 t18
t22 := AND t17 t16
t23 := OR t21 t22
t24 := AND t18 t16
t25 := OR t23 t24
t26 := select acc (2)
t27 := select x (2)
t28 := XOR t26 t27
t29 := XOR t28 t25
t30 := AND t26 t27
t31 := AND t26 t25
t32 := OR t30 t31
t33 := AND t27 t25
t34 := OR t32 t33
t35 := select acc (3)
t36 := select x (3)
t37 := XOR t35 t36
t38 := XOR t37 t34
t39 := AND t35 t36
t40 := AND t35 t34
t41 := OR t39 t40
t42 := AND t36 t34
t43 := OR t41 t42
t44 := concat (t38 t29 t20 t11)
acc := t44
```

Fig. 5. Maki representation of the 4-bit accumulator netlist from Figure 2 after linearization.

3 LOOP IDENTIFICATION

Here we describe our technique for identifying slices of repeated logic in a netlist that may reasonably correspond to a loop in the higher-level HDL. Translating the netlist to Maki linearizes the graph of wires and gates into a straight-line program in SSA form. Our loop identification task is to find continuous segments of repeated statements in the program.

We take inspiration from software clone detection [Baker 1995; Kamiya et al. 2002]. In our case, a "clone" A is a segment of Maki code that is identical to some other Maki code segment B, modulo variable identifiers (i.e., identifiers are not considered). We specifically look for *tandem repeats* [Stoye and Gusfield 2002], that is, a sequence of consecutive clones without anything in-between the repeated code segments. The entire loop identification process consists of three phases:(1) tokenize the Maki program; (2) scan the resulting token stream for tandem repeats; (3) heuristically filter out tandem repeats that are undesirable candidates for loop rerolling. The end result is a set of slices of the Maki program, each slice being a candidate for loop rerolling.

3.1 Tokenization

We transform the Maki program into a sequence of *tokens*, similarly to lexical analysis in parsing but applying certain abstractions to ignore irrelevant differences such as variable identifiers (because hardware synthesis would have unrolled a loop and given each iteration its own wires, thus including identifiers would mean that no clones can exist). Since statements in Maki are only two types (variable definitions and loops), and there are no loops in the initial translation of the netlist to Maki, tokenizing a Maki program considers only one case: variable definitions.

If the right-hand side of a variable definition is a wire expression, then we create a token for the wire operation. Array expressions are ignored (because we start from a netlist there will be no array expressions initially). If the wire expression is assignment, as in `b := a`, we record the bitwidth in the token as well as if it is an output wire vector (see the last token in Figure 6 for an example). As an example, Figure 6 shows the accumulator netlist as a token stream.

3.2 Finding Tandem Repeats

A repeated sequence of tokens with no interruptions indicates a candidate for loop rerolling. Note that because Maki linearizes the netlist graph, not all loops in the original HDL code may result in identical code segments for each loop iteration (if different iterations are linearized differently); we show in our evaluation that in practice using the linearized form works well. We use a standard sequence alignment technique using suffix trees to detect longest common prefixes [Kasai et al. 2001]; this technique returns the location, length, and number of repeats contained in each tandem repeat present in the token stream, which when mapped back to the Maki code yields a potential

⟨**select**⟩ ⟨**select**⟩ ⟨XOR⟩ ⟨XOR⟩ ⟨AND⟩ ⟨AND⟩ ⟨OR⟩ ⟨AND⟩ ⟨OR⟩
⟨**select**⟩ ⟨**select**⟩ ⟨XOR⟩ ⟨XOR⟩ ⟨AND⟩ ⟨AND⟩ ⟨OR⟩ ⟨AND⟩ ⟨OR⟩
⟨**select**⟩ ⟨**select**⟩ ⟨XOR⟩ ⟨XOR⟩ ⟨AND⟩ ⟨AND⟩ ⟨OR⟩ ⟨AND⟩ ⟨OR⟩
⟨**select**⟩ ⟨**select**⟩ ⟨XOR⟩ ⟨XOR⟩ ⟨AND⟩ ⟨AND⟩ ⟨OR⟩ ⟨AND⟩ ⟨OR⟩
⟨**concat**⟩ ⟨**register**, 4⟩

Fig. 6. A tokenized version of the 4-bit accumulator netlist from Figure 5.

loop rerolling candidate.[2] When we apply this process to the token stream in Figure 6, it shows that the boxed tokens in that figure represent the first repetition of a tandem repeat of length four.

Note that a tandem repeat may *not* represent a valid loop in HDL code, i.e., this loop identification process is an over-approximation of the token stream. This approximation is because our tokenization necessarily abstracts the Maki code and ignores things like variable identifiers; a tandem repeat may, once we look at the actual wire definitions it contains, not correspond to an iterative repetition of logic that is characteristic of a loop.

On the other hand, it may also be the case that a tandem repeat can be successfully rerolled but does not correspond to a loop in the original HDL code. The reason for this discrepancy is due to how hardware synthesis lowers HDL code to a netlist. During hardware synthesis, higher-level operations over wire vectors expand to lower-level operations over single-bit wires. For instance, consider an `AND` operation over two 4-bit input wire vectors `a` and `b`. In an HDL like SystemVerilog or PyRTL, this can be written simply as `c = a & b`, but in the resulting netlist this operation gets expanded into 4 repeated `AND` operations (for each bit in the wire vector) with the result concatenated into an output wire. While this repeat is not a loop in the original HDL code, it is nonetheless a sequence of repeated logic we can detect and reroll into a valid loop.

4 SKETCH GENERATION FOR LOOP REROLLING

Given a loop candidate, i.e., a tandem repeat in the Maki code as identified using the technique described in Section 3, we want to rewrite the candidate into Maki code that uses higher-level loop and array abstractions. One might think that we could simply take a single element of the tandem repeat (corresponding to a single iteration of the desired loop) and wrap it inside a loop expression. The reality is more difficult: the newly-created loop must maintain the correct pre-, post-, and intra-loop wire dependencies to guarantee semantic equivalence to the original Maki code, and must also infer non-trivial bit-selecting arithmetic (not present in the original low-level code) in order to allow a single loop body to compute different iterations of the loop correctly.

One potential strategy would be to use heuristic code transformations that attempt to preserve the necessary wire dependencies and semantic equivalence to the original code. However, our experience is that the required heuristics are very design-dependent and differ widely across different netlists, and that inferring the necessary arbitrary bit-selecting arithmetic using static analysis is non-trivial and often fails.

Instead, we leverage sketch-based program synthesis [Solar-Lezama et al. 2006]. This takes a sketch of the desired code (i.e., Maki code containing *holes* for synthesis to fill in) and an oracle for determining correctness (in this case, the original Maki program), then applies an SMT solver to produce a new Maki program based on the provided sketch, with the holes filled in, that is guaranteed to be semantically equivalent to the original Maki program. In the remainder of this section we discuss how to automatically create a suitable sketch given a specific tandem repeat. In the next section we discuss how to use that sketch for Maki-specific program synthesis.

[2] We filter out candidates of only two repetitions because these rarely correspond to useful loops.

```
(input x 4)
(register acc 4)
t3 := const 0 1
for-range i, 4 {
    t8 := select acc (0)
    t9 := select x (0)
    t10 := XOR t8 t9
    t11 := XOR t10 t3
    t12 := AND t8 t9
    t13 := AND t8 t3
    t14 := OR t12 t13
    t15 := AND t9 t3
    t16 := OR t14 t15 }
t44 := concat (t38 t29 t20 t11)
acc := t44
```

(a)

```
(input x 4)
(register acc 4)
t3 := const 0 1
for-range i, 4 {
    t8 := select ?w (?n)
    t9 := select ?w (?n)
    t10 := XOR t8 t9
    t11 := XOR t10 ?w
    t12 := AND t8 t9
    t13 := AND t8 ?w
    t14 := OR t12 t13
    t15 := AND t9 ?w
    t16 := OR t14 t15 }
t44 := concat (?w ?w ?w ?w)
acc := t44
```

(b)

Fig. 7. Intermediate reroll sketches of an accumulator design in Maki. (a) Initial reroll sketch of the accumulator. Note that the sketch has not been made generic yet. That is, before inserting any holes we start by just copying the first iteration's statements into a new loop body. (b) Reroll sketch of the accumulator after inserting holes from the reaching definitions pass Algorithm 1.

We add two new constructs to Maki's syntax to represent holes, ?w and ?n, defined as:

$$\texttt{?w} \ ::= \textit{wire} \mid \textbf{array-ref}\ \textit{var}\ \textit{nexp}$$
$$\texttt{?n} \ ::= \textit{nexp}$$

?w represents a WireVector hole and can take the place of any Maki expression that resolves to a wire vector-typed value (including reading from arrays of wire vectors). ?n represents a NumExpression hole and is used for filling in the indexing arguments to array references, array stores, and wire vector bit-selects.

We begin sketch generation by picking an arbitrary element of the tandem repeat to serve as the loop body and wrap it within a **for-range** expression to create an initial sketch. Sketch generation is split into two main passes based on (1) *reaching definitions* and (2) *liveness* properties of variables in the Maki program. These passes are made efficient by the fact that the netlist translated into Maki is already in SSA form. Figure 7a shows the initial sketch for our accumulator example.

4.1 Reaching Definitions Pass

The first pass for sketch generation uses reaching definitions information of wire variables in a Maki program. This pass focuses on variable *uses*, and will only insert holes into the right-hand side of variable definitions. We compute a *use-def* chain to capture the reaching definitions of a Maki program. This data structure is commonly used in compilers for data-flow analysis. The use-def chain of a program maps each use of a variable to definitions which reach that use. Because our initial Maki netlist is in SSA form, an element in a use-def chain maps to precisely one definition.

The use-def chain tells us where in the program we have broken data dependencies—in the form of unreachable or missing definitions—from inserting the new loop sketch. With this, we perform a pass over the Maki program using Algorithm 1. The map returned from the procedure informs which parts of which statements to update with the given hole. The first loop (lines 3–10) scans the right-hand side of variable definition statements in the loop body. First, the pass scans the right-hand side of each variable definition statement in the loop body. If the use of a variable has no reaching definition in the loop body, it replaces that variable use with a ?w hole. This pass also scans the right-hand side of variable definitions for numeric and constant arguments—replacing

any numeric bit-select argument with a ?n hole (since any NumExpression-typed value comes only from a wire select operation). It also replaces any **const**-typed expression with a ?w hole.

In lines 7–8 the algorithm replaces any numeric bit-select argument with a ?n hole (since any NumExpression-typed value will only come from a wire select operation). In lines 9–10, the algorithm replaces any **const**-typed expression with a ?w hole. The second loop (lines 11–14) scans the right-hand side of variable definition statements after the loop. Next, the pass scans the right-hand side of variable definition statements *after* the loop. The algorithm replaces any variable uses which have no reaching definition with a ?w hole. After running Algorithm 1 over the code in Figure 7a we get the intermediate sketch in Figure 7b.

Algorithm 1 Sketch generation pass based on reaching definitions. *Statements* is a list of Maki statements indexed from 0 to *n*. *UD* holds the use-def chain for the Maki code in *Statements*. Returns *holes*, a map of statement indices to a set of pairs of variable identifiers with their respective holes.

```
1:  procedure ReachingDefsPass
2:      holes ← ∅
3:      for all i ∈ {LoopStart, ..., LoopEnd} do
4:          for all use ∈ Statements[i].rhs do
5:              if UD[use] ∉ {LoopStart, ..., i} then
6:                  holes[i] ← holes[i] ∪ (use, ?w)
7:              if Type(UD[use]) = NumExpression then
8:                  holes[i] ← holes[i] ∪ (use, ?n)
9:              if Type(UD[use]) = const then
10:                 holes[i] ← holes[i] ∪ (use, ?w)
11:     for all i ∈ {LoopEnd + 1, ..., n} do
12:         for all use ∈ Statements[i].rhs do
13:             if UD[use] ∉ {0, ..., LoopStart − 1} ∪ {LoopEnd + 1, ..., n} then
14:                 holes[i] ← holes[i] ∪ (use, ?w)
15:     return holes
```

4.2 Liveness Pass

The second pass for sketch generation uses liveness information of wire variables in a Maki program. This pass focuses on program variable *definitions*, and will introduce new variable definitions into the sketch. This time, we compute a *def-use* chain to capture the liveness data in a Maki program. The def-use chain of a program maps each variable definition to the uses which reach that definition. Note that a use-def chain and def-use chain for the same program are not symmetric. There is information in one that is not captured in the other.

The information in the def-use chain tells us where in the program we have broken data dependencies—in the form of dead variables—from inserting the new loop sketch. With the def-use chain, we perform a second pass over the Maki program using liveness information Algorithm 2. The map returned from the procedure tells us which definitions to add to the sketch. There are two cases for modifying the sketch to fix liveness:

(1) Lines 5–6: The variable is dead. Any uses of the variable were removed when the remaining statements in the original unrolled code were discarded. This case indicates some data is feeding forward into subsequent loop iterations. From the accumulator example, this is the carry-in bit (variable `t3`). The solution is to provide a definition *before* the loop. The uses have already been replaced with ?w holes from the reaching definitions pass (Algorithm 1).
(2) Lines 7–10: The variable is defined within the loop and is live *after* the loop. This case indicates that the loop is accumulating some results each iteration. The solution is to declare an array

before the loop, and update the array each iteration with the just defined variable. From the accumulator example, the intermediate sums are stored in an array to be used after the loop.

Algorithm 2 Sketch generation pass based on liveness information. *DU* holds the def-use chain for the Maki code in *Statements*. Returns *NewDefs*, a map from statement indices to a set of pairs of variable identifiers with their respective definitions.

```
1:  procedure LivenessPass
2:      NewDefs ← ∅
3:      for all i ∈ {LoopStart, ..., LoopEnd} do
4:          for all def ∈ Statements[i].lhs do
5:              if DU[def] ⊈ {LoopStart, ..., i} ∪ {LoopEnd + 1, ..., n} then
6:                  NewDefs[LoopStart − 1] ← NewDefs[LoopStart − 1] ∪ (def, ?w)
7:              if DU[def] ⊆ {LoopEnd + 1, ..., n} then
8:                  x ← FreshVariable
9:                  NewDefs[LoopStart − 1] ← NewDefs[LoopStart − 1] ∪ (x, array-create LoopReps)
10:                 NewDefs[i] ← NewDefs[i] ∪ (x, array-store ?n def)
11:     return NewDefs
```

After applying the liveness pass (Algorithm 2) over the code in Figure 7b we get the final sketch in Figure 8a. Generating a sketch that preserves data dependencies before, after, and within the new loop is general enough to work for netlists that contain sequences of repeated logic. The resulting sketch over-approximates the number of unknowns but it ensures that no data dependencies are broken after inserting the loop and transforming the Maki program.

4.3 Properties of Sketch Generation

If a tandem repeat from loop identification is a valid, rerollable loop then, with one caveat, our sketch generation process introduces sufficient holes to reroll it. The caveat is that that we do not consider alternative schemes for bundling wires together in the netlist other than the one present in the original Maki code. Doing so could potentially allow more identified loops to be rerolled than our current technique, and is an interesting future direction.

We argue that, modulo the wire bundling scheme, this property is true because all points of dependencies between variables inside and outside the loop are addressed through `?w` and `?n` holes. That is, initially creating the loop sketch (as in Figure 7a) breaks some data dependencies in the Maki program. However, our pre-, post-, and intra-loop strategies for patching the dependencies cover all relevant points (aided by the reaching definitions and liveness analysis of the variables in the original SSA form of the program), and the holes are flexible enough that if there exists a way to reroll the candidate into a valid loop then the resulting sketch provides at least one way.

5 PROGRAM SYNTHESIS FOR LOOP REROLLING

With our generated sketch of the hardware design with loops, we want to find a solution that fills in the holes and produces a design equivalent to the original netlist. We use an established program synthesis technique called *counterexample-guided inductive synthesis* (CEGIS) [Solar-Lezama 2013]. CEGIS completes a program sketch by searching for a candidate solution (i.e., a way to fill the holes) and then verifying it against the reference specification. Here, the reference specification is the unmodified netlist. CEGIS searches for a solution by translating the candidate to constraint formulas and feeding them into an SMT solver. While verifying a candidate solution, if the SMT solver finds a counterexample that falsifies the solution, CEGIS generates a new candidate solution taking into account previously found counterexamples. This solve-verify loop continues until a solution is synthesized that satisfies the specification, or it determines that a solution does not exist.

```
(input x 4)
(register acc 4)
t3 := const 0 1
t4 := array-create 4
t16 := ?w
for-range i, 4 {
    t8 := select ?w (?n)
    t9 := select ?w (?n)
    t10 := XOR t8 t9
    t11 := XOR t10 ?w
    t4 := array-store ?n t11
    t12 := AND t8 t9
    t13 := AND t8 ?w
    t14 := OR t12 t13
    t15 := AND t9 ?w
    t16 := OR t14 t15 }
t44 := concat (?w ?w ?w ?w)
acc := t44
```

(a)

```
(input x 4)
(register acc 4)
t3 := const 0 1
t4 := array-create 4
t16 := t3
for-range i, 4 {
    t8 := select acc (i)
    t9 := select x (i)
    t10 := XOR t8 t9
    t11 := XOR t10 t16
    t4 := array-store i t11
    t12 := AND t8 t9
    t13 := AND t8 t16
    t14 := OR t12 t13
    t15 := AND t9 t16
    t16 := OR t14 t15 }
t44 := concat ((array-ref t4 3) (array-ref t4 2)
     (array-ref t4 1) (array-ref t4 0))
acc := t44
```

(b)

Fig. 8. (a) Final reroll sketch of the accumulator design after inserting holes from the liveness pass Algorithm 2. (b) The rerolled designed for the accumulator sketch in (a) after program synthesis.

5.1 Solver-Aided Maki

We wrote a symbolic interpreter that "runs" programs written in Maki. The symbolic interpreter keeps track of the program's state—that is, the variable definitions—where all input wires to the netlist are symbolic bitvectors. This process compiles a netlist specification into a set of symbolic constraints. These constraints enable solver-based verification and synthesis. These solver-aided functions come from Rosette, a Racket framework for CEGIS [Torlak and Bodik 2014].

Rosette is a language-driven framework for building program synthesizers. By defining a language and an interpreter for that language, Rosette can lift the evaluation of programs in that language to work with symbolic values. This "symbolic evaluation" is the process that converts a program into a set of constraint formulas that an SMT solver can understand.

5.2 The Loop Rerolling Pipeline

Continuing the accumulator example, loop rerolling via program synthesis works as follows. Given a netlist translated to Maki, and a sketch of that netlist with loops generated by the methodology in Section 4, do the following:

(1) First, symbolically interpret the netlist.
(2) Using a CEGIS solver, produce a candidate solution for the sketch.
(3) Symbolically interpret the candidate solution and verify it against the reference netlist. Check the equivalence of each of the symbolically defined output wires.
(4) If the solver finds a counterexample, add it to the current set of constraints, then generate a new candidate. Repeat until it determines a solution does not exist.
(5) Otherwise, if the candidate satisfies the reference netlist specification, substitute the holes in the sketch according to the expressions found in the solution. The resulting program is the semantically-equivalent synthesized Maki code with loops.

Figure 8b presents the synthesized code with a rerolled loop for the 4-bit accumulator design. Note that `t16` is initialized to a constant before the loop (as a result of the pre-loop dependency check), and is updated at the end of each iteration inside the loop. This variable holds the carry-in and carry-out values for the adder. The `?n` holes inside the loop body fill in with loop variable `i`. `?w`

```
module accumulator (input  clk, input [3:0] x,
      input reg [3:0] acc);
 logic t0; logic [3:0] t1;
 always_comb begin
   t0 = 1'b0;
   for (int i=0; i < 4; i++) begin
     t1[i] = (acc[i] ^ x[i]) ^ t0;
     t0 = ((acc[i] & x[i]) | (acc[i] & t0)) |
          (x[i] & t0);
   end
 end
 always_ff @(posedge clk) begin
   acc <= {t1[3], t1[2], t1[1], t1[0]};
 end
endmodule
```

(a)

```
from pyrtl import *
acc = Register(bitwidth=4)
x = Input(bitwidth=4)
t0 = Const(0, bitwidth=1)
t1 = [None]*4
for i in range(4):
  t1[i] = (acc[i] ^ x[i]) ^ t0
  t0 = ((acc[i] & x[i]) | (acc[i] & t0)) | (x[i] & t0)
acc.next <<= concat(t1[3], t1[2], t1[1], t1[0])
```

(b)

Fig. 9. Decompiled SystemVerilog (a) and PyRTL (b) code for the 4-bit accumulator.

holes inside the loop body fill in with previously defined wire variables. The ?w holes after the loop in the **concat** operation resolve to **array-ref** operations that retrieve the stored sum of each of the four bits of the input wires. These **array-ref** holes correspond to the post-loop dependencies introduced in sketch generation.

5.3 Output to HDL Code

After loop rerolling we can easily translate Maki code to an HDL. For instance, our tool translates Maki to SystemVerilog and PyRTL [Clow et al. 2017]. Figures 9a and 9b present the rerolled and decompiled 4-bit accumulator in SystemVerilog and PyRTL, respectively. In Section 1, we presented the original SystemVerilog code for the accumulator in Figure 1. Note the similarity between the rerolled code in Figure 9a and the original code[3].

Also note that the original SystemVerilog code splits the design into two separate modules (or two functions, for PyRTL). Here, the decompiler emits the design as one flattened module. Procedural abstraction is one branch of future work, whereby a candidate fragment of the netlist is wrapped into a module or function body, and all occurrences of the fragment are replaced with a module instantiation/function call.

Table 1. PyRTL benchmark information. “Loops” are the number of loops present in the original source code. The “small”, “medium”, and “large” columns denote a particular parameterization of the design and the number of wires and gates in the resulting netlist. A benchmark noted as ’recursive’ means that the loops are implemented via recursive function calls.

		Small		Medium		Large	
Module	**Loops**	**Wires**	**Gates**	**Wires**	**Gates**	**Wires**	**Gates**
Barrel shifter	1	44	42	196	194	839	836
Cache (directed)	1	199	158	391	310	775	614
Cache (n-way set associative)	1	165	125	326	249	645	495
Demultiplexer	1	88	83	274	269	507	502
Priority encoder (recursive)	1	79	57	146	107	277	205
Pseudo-random number generator	1	43	40	139	136	526	522
Ripple-carry adder (iterative)	1	79	74	295	290	583	578
Ripple-carry adder (recursive)	1	97	92	385	380	769	764
Shifter	1	160	140	320	284	640	572

[3]For readability, we rewrite expressions in the rerolled loops from the “three-address code” form into nested wire expressions.

6 EVALUATION

Here we evaluate our implementation of the loop identification and rerolling techniques. Our implementation has two parts that span two languages: we implement loop identification in Python 3.9 and loop rerolling (sketch generation plus program synthesis) in Racket 8.7 using the Rosette CEGIS framework [Torlak and Bodik 2014]. To evaluate our loop identification and rerolling we use two sets of hardware design benchmarks written in two different HDLs: PyRTL (Table 1) and SystemVerilog (Figure 10). The PyRTL benchmarks are a mix of combinational and sequential designs for basic components such as adders, shifters, and caches.

The SystemVerilog benchmarks are taken from the BaseJump Standard Template Library [Taylor 2018] (BaseJump STL) found in the BSG Micro Designs repository [Tang and Davidson 2019]. BaseJump STL is a standard template library of components commonly found across many different hardware designs. We chose designs from BaseJump STL without regard to their original SystemVerilog code containing loops in the interest of finding loop rerolling opportunities where there were none originally.

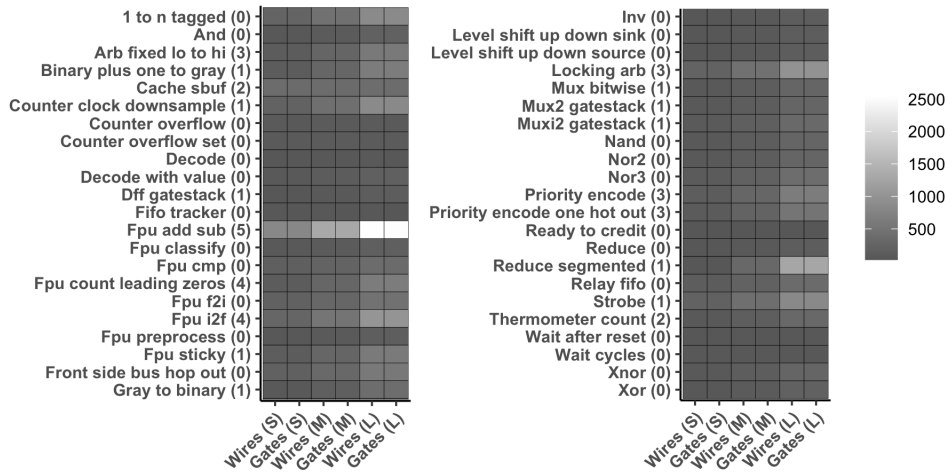


Fig. 10. Heat map of benchmark sizes for the BaseJump modules, shows count of wires and gates for each parameterization (S/M/L) of the module. The number of loops present in the SystemVerilog source code is shown in parentheses next to the module name.

Our translator from netlist to Maki operates on PyRTL-format netlists. For BaseJump STL we converted each module into BLIF format [Berkeley 1992] via Yosys [Wolf 2021] and imported it into PyRTL to have the netlist in the correct format. Due to this multi-step conversion process we only evaluate a subset of the BaseJump STL modules. For instance, PyRTL does not support asynchronous designs and so those modules were not included.

We parameterize each benchmark over three configurations we denote as “small”, “medium”, and “large”; for many designs this meant setting the width parameter to 16, 32, and 64, respectively. Note that the size categories are benchmark-specific, not meant to be compared across benchmarks. We test our implementation on the netlists produced from these modules for each size configuration.

We rely on PyRTL’s compiler for netlist linearization; when building a netlist PyRTL performs a topological sort over wires in the graph and assigns them a deterministic order where related operations are close together in practice. With this sort, our tool lifts each netlist to Maki, tokenizes the code, and performs loop identification. The Maki program and loop information then feed into

Table 2. PyRTL benchmark loop identification and rerolling results. The "small", "medium", and "large" rows for each benchmark denote a particular parameterization of the design. Both the loop identification and loop rerolling phases have a timeout of 1 hour.

Module	Size	Loops found / expected	Loops rerolled	Loop detection time (s)	Loop rerolling time (s)
Barrel shifter	Small	1 / 1	1	0.1	6.8
	Med.	1 / 1	1	0.6	11.4
	Large	1 / 1	1	6.3	27.9
Cache, directed	Small	1 / 1	1	0.4	8.8
	Med.	9 / 1	1	1.8	53.1
	Large	1 / 1	1	7.1	198.3
Cache, n-way set associative	Small	1 / 1	1	0.3	13.8
	Med.	3 / 1	2	0.8	38.9
	Large	4 / 1	2	4.2	146.3
Demultiplexer	Small	1 / 1	1	0.2	6.3
	Med.	3 / 1	2	1.1	16.3
	Large	2 / 1	1	4.1	26.7
Priority encoder (recursive)	Small	3 / 1	2	0.1	9.1
	Med.	5 / 1	3	0.3	16.2
	Large	6 / 1	4	0.8	31.1
Pseudo-random number generator	Small	1 / 1	1	0.1	8.4
	Med.	1 / 1	1	0.2	14.2
	Large	1 / 1	1	2.5	78.9
Ripple-carry adder (iterative)	Small	1 / 1	1	0.1	5.9
	Med.	1 / 1	1	0.6	8.1
	Large	1 / 1	1	3.9	12.1
Ripple-carry adder (recursive)	Small	2 / 1	1	0.1	6.9
	Med.	2 / 1	1	2.0	24.1
	Large	2 / 1	1	15.6	102.5
Shifter	Small	9 / 1	1	0.3	21.1
	Med.	1 / 1	1	1.0	21.5
	Large	33 / 1	1	5.9	441.8

the loop rerolling phase. If the sketch is satisfiable, the loop rerolling tool outputs the decompiled HDL code. Note that our decompiler supports PyRTL and SystemVerilog as an output language regardless of which HDL generated the input netlist. Both phases of loop identification and rerolling were run on a machine with a 6-core Intel Xeon E5-2420, 32 GB RAM, running Ubuntu 18.04.5.

6.1 Loop Identification and Rerolling Results

Table 2 presents the loop rerolling results over the nine PyRTL benchmarks. Since a netlist represents a particular parameterization of a hardware design, we ran loop identification and rerolling on three configurations ("small", "medium", and "large") for each benchmark. Each PyRTL benchmark contained one loop in its source code, and our decompiler identified and rerolled that loop in each case. The decompiler often found more loops than were contained in the original HDL code; for example, on the large version of the priority encoder the decompiler identified 6 potential loops and rerolled 4 of them. Loop identification over-approximates the number of loops that can be rerolled. Thus, some potential loops are false positives and the decompiler does not always reroll as many loops as it finds.

PyRTL can also represent repeated logic using recursive functions. We include two recursive benchmarks (priority encoder and the recursive ripple-carry adder) in our evaluation to show that at the netlist level a recursive PyRTL function still gets unrolled into a set of repeated gates and wires, and our loop rerolling tool can still identify and reroll those operations into an equivalent loop (though converted into an iterative loop rather than recursion).

Figure 11 presents the loop rerolling results over the BaseJump STL benchmarks. These benchmarks are generally larger than the PyRTL benchmarks and that is reflected in the number of loops

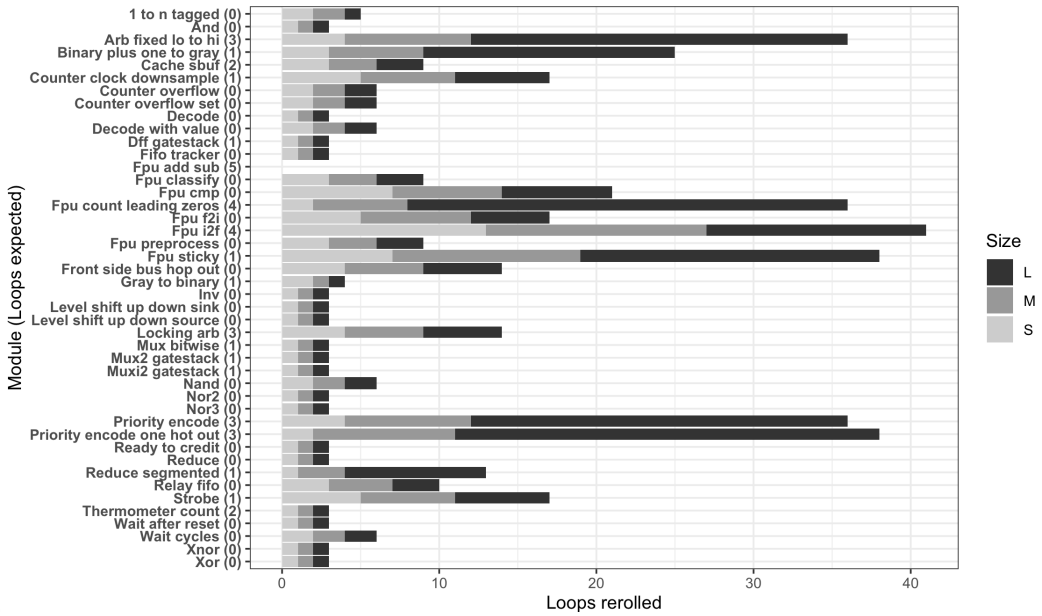
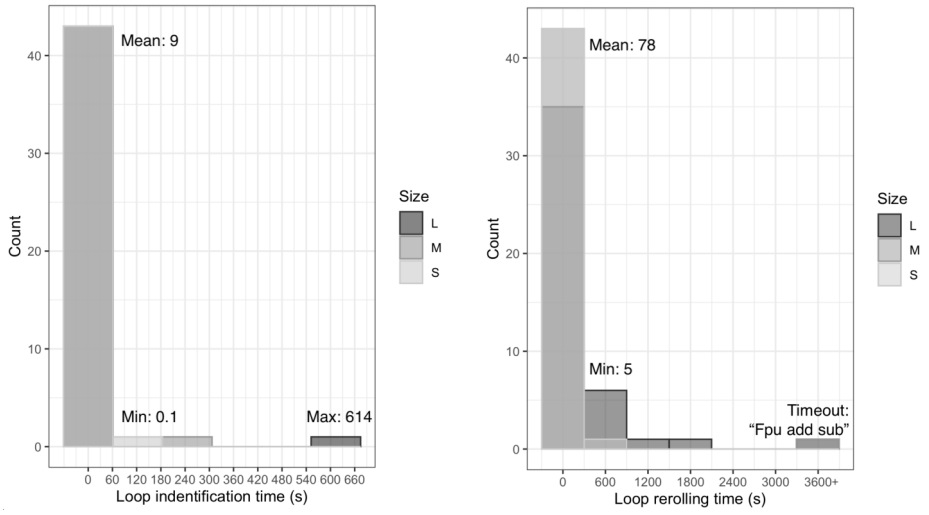


Fig. 11. Number of loops rerolled across all BaseJump benchmarks (across all parameterizations). The number of expected loops present in the SystemVerilog source code is shown in parentheses next to the module name.

rerolled. The large version of "Fpu count leading zeros" rerolled 28 of the 31 loops identified, the most rerolled in our evaluation. Also note that the BaseJump STL modules have instances where the original SystemVerilog code has no loops, but after analyzing the netlist our tool finds opportunities to reroll loops. Overall, with the exception of "Fpu add subtract", our tool identified and rerolled at least one loop in every module (often more).

Discussion. Considering the original HDL code in our benchmarks contained few loops, it is noteworthy that our tool often found and rerolled many more loops. There are two explanations for this phenomenon: (1) The HDL code may explicitly repeat the logic instead of parameterizing it over a loop. The BaseJump STL modules with zero expected loops often do this in the original SystemVerilog code. Nevertheless, this point is helpful to the decompiler user for understanding what the repeated operation is and how it is parameterized. (2) As noted in Section 3.2, when hardware synthesis lowers HDL code to a netlist, common higher-level operations in the original HDL code are expanded into chunks of lower-level repeated logic, increasing the loop identification count. The lowering process introduces non-obvious looping behavior that was not present in the higher-level design. Nonetheless, our tool identifies these repeated wire operations and attempts to reroll them. If successful, this kind of loop uncovers a repeated operation that is likely part of some larger structure that was lowered during synthesis. For these kinds of loops we argue this is beneficial for the decompiler user as the rerolled loop identifies some repeated logic (out of a large graph of similar nodes) and generates a concise representation in high-level code.

There are instances where we reroll loops that do not cover all of the iterations of the original loop. Either the first or last iterations may be missing. This mismatch is due to the limitations of our tandem repeat analysis during loop identification where token sequences must be *exact* matches. In some cases, the first or last iteration of a repeat may differ in a way that results in a different token sequence from the other repeats.



(a) (b)

Fig. 12. (a) Histogram of loop identification performance across all BaseJump benchmarks. (b) Histogram of loop rerolling performance across all BaseJump benchmarks. The two points in the "3600+" bin indicate a timeout for the "Fpu add subtract" module sizes "medium" and "large".

Based on our evaluation we find that nested loops in the original HDL will translate to larger non-nested loops in the decompiled HDL. For instance, the "Priority encode one hot out" benchmark actually has a nested loop inside one of its submodules. However, after conversion to Verilog and then to BLIF, the nested loops are essentially unrolled and the resulting netlist loses any notion of it. In this case, the best our technique recovers is a single loop with a larger body.

For netlist linearization, our evaluation empirically shows that a topological sort works well in practice. Alternative approaches to linearizing a netlist are interesting to consider (and might be part of future work). While we rely on the PyRTL compiler for netlist linearization, the utility of the topological sort is not limited to PyRTL-produced netlists as we also evaluate netlists generated from SystemVerilog code. These netlists have also undergone optimization passes in Yosys before being output to BLIF, showing that loop identification is also effective in the presence of optimizations.

Loop identification and rerolling times are shown in Figures 12a and 12b, respectively. One limitation of our loop rerolling tool comes from the constraints it sends to the SMT solver. For large netlists, solving times dramatically increase from a few seconds to minutes to over an hour. As Figure 12b shows, the "Fpu add sub" module timed out (mainly due to the netlist's size). Exploding solving times is a known problem in program synthesis and research has studied how to diagnose and fix performance issues related to symbolic evaluation [Bornholt and Torlak 2018]. To scale to larger netlists, our tool needs to overcome this bottleneck at the program synthesis stage.

While the benchmarks in our evaluation are small, their module-level behavior is representative of the kinds of components used in real-world hardware designs—this is one motivation for choosing the BaseJump STL. To scale module-level hardware loop rerolling to larger designs, there are techniques that can infer module boundaries in large netlists [Subramanyan et al. 2014]. These techniques are orthogonal and complementary to our work in hardware decompilation in that they can be used to decompose a netlist into modules which then can be decompiled by our technique.

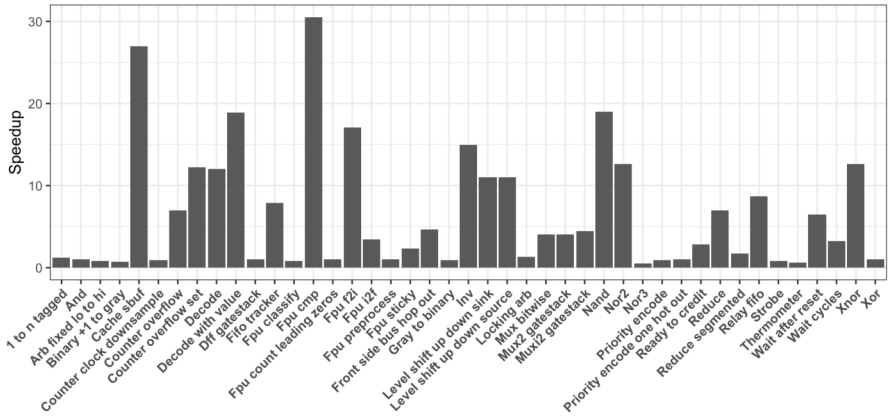


Fig. 13. Speedups in Verilator simulation times across all "large" versions of the BaseJump STL modules that successfully rerolled loops compared to the original netlist.

6.2 Transpilation Between HDLs

Since our hardware decompilation tool operates over a common IR, Maki, as well as outputs SystemVerilog and PyRTL, it enables automated translation between HDLs. For example, we can take a design that starts in SystemVerilog, synthesize it to a netlist, and decompile it into PyRTL code. The PyRTL code for the rerolled accumulator in Figure 9b is one example of transpilation from SystemVerilog. All of the PyRTL benchmarks in our evaluation can be synthesized and decompiled into SystemVerilog. The same is true for decompiling the SystemVerilog benchmarks into PyRTL.

As Maki is a small language, we mapped all Maki constructs to equivalent constructs in SystemVerilog and PyRTL. One point is that Maki is closer in design to PyRTL than SystemVerilog. Since the clock is implicit in Maki and PyRTL, we add a clock to the exported SystemVerilog code and wrap any sequential logic into a `always_ff @(posedge clk)` block where `clk` is the added clock. All combinational logic is wrapped in a `always_comb` block.

6.3 Speeding Up Simulation Time

In this section we compare simulation times for the BaseJump STL modules that successfully rerolled loops against their original netlists. We run the simulations using Verilator, an open-source SystemVerilog simulation tool [Snyder 2021]. For each module, we supply pseudorandom inputs to the netlist and decompiled HDL code.

Figure 13 presents the speedups for simulating each module. For a majority of the modules, hardware loop rerolling speeds up simulation time, with "Fpu cmp" seeing the largest speedup at 30x. Overall, the mean speedup is 6x. However, smaller modules such as "And" and "Xor" did not gain any speedup. The reason for the slower times ties back to some of the limitations of our approach. In particular, netlist linearization can affect the performance of rerolled loops in Verilator simulation. For instance, if a netlist linearization shifts the bit-select for a wire vector by 2 from the original monotonic ordering, our loop reroller will still reroll it into a loop. However, instead of referencing the bit-select with loop index `i` the synthesizer generates a more complicated arithmetic expression (e.g., `(i + 2) % n`, for a wire vector with bitwidth `n`). While still a correct loop with respect to the original netlist, the loop eludes easily applicable optimizations in Verilator. We can overcome these slowdowns in some cases by rewriting single-operation loops into one-line bitwise operations where feasible (e.g., `c = a & b`).

6.4 Artifact Compaction

To evaluate the benefits of loop rerolling for decompilation, we also measured artifact compaction. Since the goal of decompilation is to produce HDL code, we compare the rerolled code translated to PyRTL with the original Maki code translated to PyRTL without loop rerolling. We record the size of the artifact in bytes after the it is compressed with `gzip` on the highest compression level.

Overall, as parameter sizes grow, so does the degree of compaction between the rerolled code and the netlist. For some designs this is a significant difference, seeing up to 90% artifact compaction (the "large" versions of the PyRTL barrel shifter and iterative ripple-carry adder) and 39% compaction on average across the entire benchmark suite.

Loop rerolling alone makes a sizable impact here. However, a few outliers, typically smaller netlists, do not benefit as much from loop rerolling. For instance, as one of the smallest netlists, the BaseJump "Fifo tracker" module actually grew in size after rerolling. Due to the small size of the original netlist, loop rerolling has a proportionally small effect.

7 RELATED WORK

7.1 Netlist Reverse Engineering

Research in this area presents techniques to recover module functionality, control logic, and data flow from a netlist graph. Most netlist reverse engineering work focuses on security scenarios, such as finding Trojans in digital circuits. There are two primary approaches for analyzing netlists: structural and functional. A structural analysis considers the shape, or topology, of the circuit to identify subcircuits [Rubanov 2006] and recover control logic [Meade et al. 2016a,b; Shi et al. 2010].

Functional analyses recover data flow and match subcircuits to templates of commonly used components. These analyses leverage QBF/SAT solvers to identify library components and word-level data paths [Gascón et al. 2014; Li et al. 2013, 2012; Soeken et al. 2015]. Other work identifies high-level blocks through graph embeddings and connectivity information [Cakir and Malik 2018]. Subramanyan et al. [2014] combine structural and functional analyses for reverse engineering circuits—first identifying submodule boundaries using a structural analysis, then mapping potential modules to a component template library via functional analysis.

These techniques in netlist reverse engineering work by producing a more structured netlist graph or finite-state machine annotated with higher-level constructs—as opposed to generating HDL code. The analogy to software binary reverse engineering is akin to recovering a control-flow graph and annotating it without decompiling to C code.

Further, work in netlist reverse engineering focuses on extracting structural information of the circuit, but not necessarily recovering the HDL code that synthesized the netlist. Some recent work makes the step to recovering register-transfer level (RTL) code [Portillo et al. 2019; Zhang et al. 2019], but does not recover higher-level programming abstractions as our work does. We differentiate hardware decompilation from previous work that only recovers RTL code. Hardware decompilation lifts low-level details in the netlist to higher-level programmatic abstractions found in HDL code (such as loops, procedures, modules, etc).

7.2 Program Synthesis

Recent research has also used program synthesis techniques to automatically generate HDL code. Sketchilog generates Verilog code given a sketch, but is limited to combinational circuits [Becker et al. 2014]. VeriSketch is another sketch-based HDL code generation tool that uses CEGIS and information flow tracking to synthesize combinational and sequential circuits that adhere to a set of security properties [Ardeshiricham et al. 2019]. Both of these tools focus on the design aspects of hardware, whereas our work comes from the opposite direction with decompilation.

Although in a different domain, another area of research conceptually related to our work uses rewrite-driven equality saturation to find loops in 3D geometric models [Nandi et al. 2020]. The motivation is similar in that decompiled low-level triangle meshes used in 3D printing are large and unstructured. Instead of syntax-guided program synthesis, Nandi et al. [2020] use rewrite rules via an equality saturation engine to reroll loops in a DSL for Constructive Solid Geometry. Using an equality saturation engine such as egg [Willsey et al. 2021], a rewrite-driven approach may improve synthesis times in our loop rerolling tool.

7.3 Software Loop Rerolling

Research in software loop rerolling focuses on rerolling for code size reduction, targeting resource-constrained environments [Hu et al. 2016; Stiff and Vahid 2005; Su et al. 1984, 1986]. Modern compilers also have loop rerolling strategies. LLVM implements a heuristic-based loop rerolling pass which operates over LLVM IR and rerolls partially unrolled iterations of single-block loops. No previous work in loop rerolling uses program synthesis techniques to reroll loops.

Recent work looks at loop rerolling at the source-code [Rocha et al. 2022] and binary level [Ge et al. 2022]. RoLAG rerolls loops by aligning blocks of straight-line code in SSA form [Rocha et al. 2022]. Aligned SSA graphs correspond to isomorphic code and are then rolled into a single loop. RollBin rerolls loops at the binary level using a custom data-dependency analysis to handle shuffled instructions and loop-carry dependencies [Ge et al. 2022]. RollBin identifies loops and infers their unrolling factor by observing memory accesses.

Our work differs from software loop rerolling because the semantics and execution model of HDLs and netlists are different from that of software and binary executables. Importantly, the *semantics* of a loop in an HDL is different from loops in software. Unlike the existing work, our work specifically uses program synthesis to generate rerolled higher-level code—using symbolic evaluation to guarantee that the rerolled loop code is semantically equivalent to the original netlist.

8 CONCLUSION

In this paper we defined and explored a new problem—hardware decompilation. This problem is the task of lifting a low-level netlist back to structured, high-level HDL code. It is a large problem, so in this paper we tackle the first step for decompiling high-level HDL code with loops. Inspired by techniques in software clone detection, we find candidate loops in netlists using a token-based analysis and sequence matching algorithms. With loop information, we generate a sketch of the code with rerolled loops and send it to a program synthesis tool that can reason about hardware designs. We evaluate hardware loop rerolling on a set of SystemVerilog and PyRTL hardware design benchmarks, noting the number of loops successfully identified and rerolled, and its impact on transpilation between HDLs, faster simulation times over netlists, and artifact compaction.

This paper lays the groundwork for future research in hardware decompilation. The hardware-oriented program synthesis tool we developed opens the door to an entire class of problems that can be solved through this technique. In the future we envision developing more circuit-based analyses to recover other high-level programming features and extending the program synthesis tool to decompile those back into HDL code.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for a careful assessment of the work and helpful comments. We also thank Michael Christensen for help with the PyRTL compiler. This material is based upon work supported by the National Science Foundation under Grants No. 2006542, 1763699, 1717779.

DATA AVAILABILITY STATEMENT

The software and data to support this work are freely available on Zenodo [Sisco et al. 2023]. The artifact consists of four components: (1) source code for loop identification over the benchmark suite of netlists; (2) source code for loop rerolling over the benchmark suite; (3) scripts for comparing simulation times between decompiled HDL code with rerolled loops and the original netlist using Verilator; and (4) Yosys scripts for converting Verilog designs to netlists in BLIF. We provide instructions to reproduce the results reported in the evaluation.

REFERENCES

Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. 2019. VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1623–1638. https://doi.org/10.1145/3319535.3354246

B. Baker. 1995. On finding duplication and near-duplication in large software systems. *Proceedings of 2nd Working Conference on Reverse Engineering* (1995), 86–95. https://doi.org/10.1109/WCRE.1995.514697

Scott Beamer. 2020. A Case for Accelerating Software RTL Simulation. *IEEE Micro* 40, 4 (2020), 112–119. https://doi.org/10.1109/MM.2020.2997639

A. Becker, D. Novo, and P. Ienne. 2014. SKETCHILOG: Sketching combinational circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–4. https://doi.org/10.7873/DATE.2014.165

UC Berkeley. 1992. Berkeley logic interchange format (BLIF). *Oct Tools Distribution* 2 (1992), 197–247.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. *SIGPLAN Not.* 34, 1 (Sept. 1998), 174–184. https://doi.org/10.1145/291251.289440

James Bornholt and Emina Torlak. 2018. Finding Code That Explodes under Symbolic Evaluation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 149 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276519

Burcin Cakir and Sharad Malik. 2018. Reverse Engineering Digital ICs through Geometric Embedding of Circuit Graphs. *ACM Trans. Des. Autom. Electron. Syst.* 23, 4, Article 50 (July 2018), 19 pages. https://doi.org/10.1145/3193121

J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. https://doi.org/10.23919/FPL.2017.8056860

Malay K. Ganai and Andreas Kuehlmann. 2000. On-the-Fly Compression of Logical Circuits. In *International Workshop on Logic Synthesis*. 7 pages.

A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik. 2014. Template-based circuit understanding. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. 83–90. https://doi.org/10.1109/FMCAD.2014.6987599

Tianao Ge, Zewei Mo, Kan Wu, Xianwei Zhang, and Yutong Lu. 2022. RollBin: Reducing Code-Size via Loop Rerolling at Binary Level. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, CA, USA) *(LCTES 2022)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/3519941.3535072

Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. 2010. Introducing Kansas Lava. In *Implementation and Application of Functional Languages*, Marco T. Morazán and Sven-Bodo Scholz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–35. https://doi.org/10.1007/978-3-642-16478-1_2

Erh-Wen Hu, Bogong Su, and Jian Wang. 2016. Instruction Level Loop De-optimization. In *Computer and Information Science 2015*, Roger Lee (Ed.). Springer International Publishing, Cham, 221–234. https://doi.org/10.1007/978-3-319-23467-0_15

A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. 2001. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Combinatorial Pattern Matching*, Amihood Amir (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 181–192. https://doi.org/10.1007/3-540-48194-X_17

W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia. 2013. WordRev: Finding word-level structures in a sea of bit-level gates. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 67–74. https://doi.org/10.1109/HST.2013.6581568

W. Li, Z. Wasson, and S. A. Seshia. 2012. Reverse engineering circuits using behavioral pattern mining. In *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. 83–88. https://doi.org/10.1109/HST.2012.6224325

T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang. 2016a. Gate-level netlist reverse engineering for hardware security: Control logic register identification. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1334–1337. https://doi.org/10.1109/ISCAS.2016.7527495

T. Meade, S. Zhang, and Y. Jin. 2016b. Netlist reverse engineering for high-level functionality reconstruction. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 655–660. https://doi.org/10.1109/ASPDAC.2016.7428086

Alan Mycroft and Richard Sharp. 2003. Higher-level techniques for hardware description and synthesis. *International Journal on Software Tools for Technology Transfer* 4, 3 (2003), 271–297. https://doi.org/10.1007/s10009-002-0086-1

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

J.T. O'Donnell. 2006. Overview of Hydra: a concurrent language for synchronous digital circuit design. *International Journal of Information* 9, 2 (March 2006), 249–264. http://eprints.gla.ac.uk/3461/

J. Portillo, T. Meade, J. Hacker, S. Zhang, and Y. Jin. 2019. RERTL: Finite State Transducer Logic Recovery at Register Transfer Level. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. 1–6. https://doi.org/10.1109/AsianHOST47458.2019.9006699

Rodrigo C. O. Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. 2022. Loop Rolling for Code Size Reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 217–229. https://doi.org/10.1109/CGO53902.2022.9741256

N. Rubanov. 2006. A High-Performance Subcircuit Recognition Method Based on the Nonlinear Graph Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 11 (2006), 2353–2363. https://doi.org/10.1109/TCAD.2006.881335

Y. Shi, C. W. Ting, B. Gwee, and Y. Ren. 2010. A highly efficient method for extracting FSMs from flattened gate-level netlist. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. 2610–2613. https://doi.org/10.1109/ISCAS.2010.5537093

Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. 2023. Loop Rerolling For Hardware Decompilation artifact. https://doi.org/10.5281/zenodo.7823993.

Wilson Snyder. 2021. Verilator. https://www.veripool.org/verilator/.

M. Soeken, B. Sterin, R. Drechsler, and R. Brayton. 2015. Simulation graphs for reverse engineering. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. 152–159. https://doi.org/10.1109/FMCAD.2015.7542265

Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

G. Stiff and F. Vahid. 2005. New Decompilation Techniques for Binary-Level Co-Processor Generation. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design* (San Jose, CA) *(ICCAD '05)*. IEEE Computer Society, USA, 547–554. https://doi.org/10.1109/ICCAD.2005.1560127

Jens Stoye and Dan Gusfield. 2002. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science* 270, 1 (2002), 843–856. https://doi.org/10.1016/S0304-3975(01)00121-9

Bogong Su, Shiyuan Ding, and Lan Jin. 1984. An Improvement of Trace Scheduling for Global Microcode Compaction. In *Proceedings of the 17th Annual Workshop on Microprogramming (MICRO 17)*. IEEE Press, 78–85. https://doi.org/10.1145/800016.808217

B. Su, S. Ding, and J. Xia. 1986. URPR—An Extension of URCR for Software Pipelining. In *Proceedings of the 19th Annual Workshop on Microprogramming* (New York, New York, USA) *(MICRO 19)*. Association for Computing Machinery, New York, NY, USA, 94–103. https://doi.org/10.1145/19551.19541

P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik. 2014. Reverse Engineering Digital Circuits Using Structural and Functional Analyses. *IEEE Transactions on Emerging Topics in Computing* 2, 01 (jan 2014), 63–80. https://doi.org/10.1109/TETC.2013.2294918

Lingshu Tang and Scott Davidson. 2019. BSG Micro Designs. https://github.com/bsg-idea/bsg_micro_designs.

Michael Bedford Taylor. 2018. BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 73, 6 pages. https://doi.org/10.1145/3195970.3199848

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. *SIGPLAN Not.* 49, 6 (June 2014), 530–541. https://doi.org/10.1145/2666356.2594340

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

Claire Wolf. 2021. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

T. Zhang, J. Wang, S. Guo, and Z. Chen. 2019. A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code. *IEEE Access* 7 (2019), 38379–38389. https://doi.org/10.1109/ACCESS.2019.2901949

Received 2022-11-10; accepted 2023-03-31