# Verifying Hardware Optimizations for Efficient Acceleration

Qianzhou Wang
Imperial College London
London, UK
qianzhou.wang17@imperial.ac.uk

Yat Wong
Imperial College London
London, UK
yat.wong18@imperial.ac.uk

Zhiqiang Que
Imperial College London
London, UK
z.que@imperial.ac.uk

Wayne Luk
Imperial College London
London, UK
w.luk@imperial.ac.uk

## ABSTRACT

Verifying the correctness of optimizations is a key challenge in hardware acceleration. Incorrect optimizations can produce designs unfit for purpose. This paper presents a novel approach, Covoh, which captures families of hardware designs as parametric block descriptions, such that the behaviour of design instances can be verified by numerical and symbolic simulation. In this work, hardware optimizations are expressed as transformations of parametric descriptions, and their parametric verification based on the Coq proof assistant is guided by verification strategies. Repositories of design descriptions and verification strategies have been developed to facilitate design development in Covoh. Its use in verifying two optimizations illustrates the capability of Covoh. The first, a variation of Horner's Rule, maps an $O(n^2)$ design to an $O(n)$ design. The second, used in optimizing avionics monitoring, maps an $O(2^n)$ design to an $O(n)$ design. The effectiveness of such optimizations is demonstrated with FPGA implementations: varying the value of a single parameter that controls pipelining would, for example, lead to a family of functionally-verified designs with different trade-offs, from ones with low throughput, low resource usage and low power consumption to ones with high throughput, high resource usage and high power consumption.

## CCS CONCEPTS

• **Hardware → Theorem proving and SAT solving**; *Simulation and emulation*; *Equivalence checking*.

## KEYWORDS

hardware verification, theorem proving, formal methods, simulation, hardware optimization, functional programming

**ACM Reference Format:**
Qianzhou Wang, Yat Wong, Zhiqiang Que, and Wayne Luk. 2022. Verifying Hardware Optimizations for Efficient Acceleration. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2022), June 9–10, 2022, Tsukuba, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3535044.3535047


*HEART2022, June 9–10, 2022, Tsukuba, Japan*

ACM ISBN 978-1-4503-9660-8/22/06…$15.00
https://doi.org/10.1145/3535044.3535047

## 1 INTRODUCTION

Efficient implementations are often obtained from applying optimizations to design descriptions which are obvious. However, verifying the correctness of optimizations is challenging. Effective hardware accelerator design increasingly depends on new approaches that reduce the effort of verifying that hardware optimizations are fit for purpose.

While proof assistants such as Coq [2][3], ACL2 [9], HOL [7] and Isabelle [18] bring new opportunities to hardware verification, these proof assistants often require significant experience before they can be used for verifying designs. Is there an approach to make use of these proof assistants such that it is straightforward to verify instances of a hardware optimization (such as optimizing a 64-bit multiplier), and useful support would be provided for verifying optimizations captured parametrically (such as optimizing an n-bit multiplier where $2 \leq n \leq 128$)?

This paper addresses this challenge by introducing Covoh (short for "COq for Verifying Optimizations of Hardware"), an approach for verifying hardware optimizations based on the Coq proof assistant [2]. The novel features of Covoh includes the following. (1) It shows how Coq can be used in a straightforward way to obtain numerical and symbolic simulation for instance verification. (2) It provides repositories of designs and proof strategies to support parametric verification. (3) It illustrates how the above features can offer a foundation for practical hardware verification, starting from numerical instance verification and then symbolic instance verification, and finally parametric verification. The practical use of Covoh will be illustrated by verification and evaluation of two conditional optimizations: a variation of Horner's Rule that optimizes an $O(n^2)$ design to an $O(n)$ design, and an optimization for avionics monitoring that optimizes an $O(2^n)$ design to an $O(n)$ design. While this paper focuses on streaming and systolic designs, the approach can cover other styles such as instruction processors.

The paper is structured as follows. Section 2 covers background and related research. Section 3 introduces the Covoh library and the associated repositories of designs, tools, and verification strategies. Section 4 demonstrates how Covoh is used in verifying and evaluating two conditional optimizations, and compares Covoh with other systems. Section 5 concludes the paper and presents opportunities for further research.

# 2 BACKGROUND AND RELATED WORK

## 2.1 Formal Verification

Formal verification of hardware is the process of proving the correctness of designs via formal methods of mathematics. Model-checking [5] is one of the standard methods for hardware verification. A design model is verified by exploring the possible states and transitions. Our approach uses deductive reasoning supported by a proof assistant [2]. Proof assistant or interactive theorem prover is software where logical reasoning can be captured by machine-aided commands. The process involves axiomatization of the model, executable algorithms, and machine-checked logical transformations. Our goal is to model the hardware as mathematical functions and prove the optimizations like proving mathematical theorems.

## 2.2 Related Work

Functional programming has been widely used in describing digital circuits. Examples include the languages $\mu$FP [20], [13] and Ruby [11], [8], which can capture both the behavioural and structural aspects of digital designs. We adopt a similar notation with basic hardware elements modelled as functions, while higher-order functions describe patterns of their interconnections. Then hardware optimizations can be expressed as transformations involving higher-order functions. Recently higher-order functions have been adopted to support abstractions in C++ for hardware design [19].

There are many verification tools available. Some require more manual interventions to carry out the proof, while others are more automatic. The more manual ones like Coq and Isabelle can often verify a wider class of theorems than those which are more automatic, like CVC4 and Z3 used in CoSA [15]. Proof assistants like Coq make use of tactics as verification strategies to prove theorems.

Other noteworthy approaches for hardware verification include RubyZF [18], which supports Ruby verification in Isabelle. Quartz [17] is also based on Isabelle, for verifying layout generation [16]. FeSi [4] is a high-level hardware description language embedded in Coq. Recently, Google proposes Silver Oak [21], a Coq-based technique for formal specification and verification of hardware, especially for security and privacy.

# 3 COVOH

This section introduces the Covoh development system, which supports simulating, verifying and compiling designs captured in a concise parametric description of block diagrams. Some experience of functional languages such as Haskell [14] would help understand the features of this parametric description, but it is not necessary to have a complete understanding of this description to appreciate Covoh's capability to verify design optimisatons.

## 3.1 Parametric Design

Covoh uses parametric descriptions to model hardware designs. We provide the complete list of definitions in the Covoh repository[1], while highlight some key features.

*3.1.1 Blocks.* A block in a circuit is modelled by a given function with specified input and output. For example, AND2 and ADD2 are

[1]https://github.com/kingsalpaca/covoh

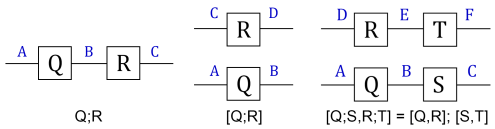


**Figure 1: Series composition and parallel composition, and a law involving them. A..F are the types of input, output or internal connections.**

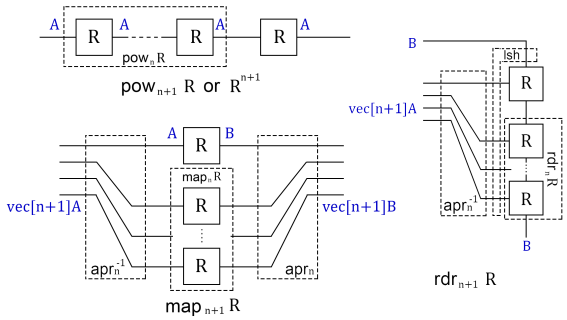


**Figure 2: Recursive higher-order functions: `pow`, `mapn`, `rdr`.**

2-input-1-output blocks defined at bit and integer level with the usual functional meaning. Wires, including identity (id), selection ($\pi_1$, $\pi_2$), replication (fork, mfork$_n$), and reordering (lsh, rsh, apl$_n$, apr$_n$, half, ...), form a family of blocks that convert types. Their converse, $R^{-1}$, corresponds to reversing the type conversion.

*3.1.2 Composing blocks.* Circuits are constructed from compositing blocks. "$Q;R$" and "$[Q,R]$" denote series and parallel compositions of blocks $Q$ and $R$ [13], [11], as shown in Fig. 1. Series composition executes one block followed by another, and parallel composition applies two blocks to pair of inputs. They correspond to `Q;;R` and `[[Q,R]]` in Covoh. The definition of parallel composition involves (`fst p`) which extracts the first element of a pair of values `p`, while (`snd p`) extracts the second element of `p`.

```
Definition series {A B C} (Q:A->B) (R:B->C) :=
  fun x : A ⇒ R (Q x)
where " Q ;; R " := (series Q R).
Definition para {A B C D} (Q:A->B) (R:C->D) :=
  fun p : A * C ⇒ (Q (fst p), R (snd p))
where " [[ Q , R ]] " := (para Q R).
```

*3.1.3 Higher-order functions.* Higher-order functions capture common patterns of combining blocks. They can be defined recursively to describe, for example, $R^n$, a repeated series composition with $n$ copies of $R$. Right reduction (`rdr`), also known as fold right in functional programming, has the following description over a 2-input-1-output block `R`.

```
Fixpoint rdr {A B} (n:nat) (R:(A*B)->B):
  (t A n * B) -> B :=
  match n with
    0 ⇒ (fun x:t A 0 * B ⇒ snd x)
  | S p ⇒ Fst (apr1 p);; lsh;; Snd R;; rdr p R
  end.
```

Notice that the type agreement between the second input and the output is a pre-condition for a block $R$ to use `rdr`, since the output is collected as the second input for the next computation. Hence the type of block $R$ is defined to be `(A*B)->B`. Fig. 2 shows further the use of wiring blocks in this definition. Repeated series composition (`pow n R` or $\text{pow}_n\ R$ or `R^^n` or $R^n$), repeated parallel composition (`mapn n R` or $\text{map}_n\ R$), triangular structure (`tri n R` or $\triangle_n\ R$), row (`row n R` or $\text{row}_n\ R$), and column (`col n R` or $\text{col}_n\ R$) are frequently used higher-order functions, some of them shown in Fig. 2. Covoh supports defining new higher-order functions together with their proof rules in describing and verifying designs.

## 3.2 Symbolic Simulation

Simulation is a commonly used technique for instance verification. Results from simulation often help understand the behaviour of designs and facilitate debugging.

Existing methods mainly focus on numerical input for simulation, which may not be easy to check the correctness of complex designs. We provide solutions for both numerical and symbolic simulation. Covoh benefits from Coq's capability to define free variables of specified types and the high-level parametric descriptions of blocks, making it possible to support symbolic input. While numerical simulation reveals numerical properties of a design, symbolic simulation offers a high degree of confidence in its correctness with the output of a circuit given as a symbolic expression of its input.

Covoh adopts the list type in Coq to model stream input/output. A stream is a function mapping time to signal. The list is suitable for modelling streams since its elements are ordered and of a uniform type. Our Covoh simulator takes a list of input and generates a list of output, where each element in the list corresponds to data at each clock cycle.

## 3.3 Parametric Verification

Parametric verification is achieved in Coq by built-in and user-defined tactics as verification strategies. A complex proof is often constructed from simpler proofs, enabling the system to be modular and flexible. Covoh adopts previously proved lemmas as rewrite rules via rewrite tactics to transform expressions with matched conditions. A pen-and-paper proof often involves applying multiple algebraic laws. Covoh can check the soundness of pen-and-paper proofs by chaining the corresponding rewrite rules. Verification is completed by applying all the relevant rewrite rules.

A straightforward rewrite rule is the equivalence between parallel of series composition and series of parallel composition: $[Q; S, R; T] = [Q, R]; [S, T]$. This is a lemma in Covoh with the following definition, which can be verified automatically via the built-in `auto` tactic of Coq. A graphical interpretation of the lemma is available in Fig. 1.

```
Lemma para_series: forall A B C D E F
  (Q:A->B) (R:D->E) (S:B->C) (T:E->F),
  [[Q ;; S, R ;; T]] = [[Q,R]] ;; [[S, T]].
```

Some rewrite rules are only valid if specific pre-conditions are met. An example is, given $Q; R = R; Q$, one can show by induction that $Q^n; R^n = (Q; R)^n$. This is a simplified version of Horner's Rule; see section 4.1 for details.

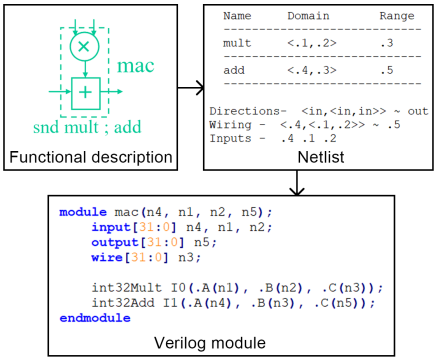


**Figure 3: The 2-step Verilog generation of a MAC block**

## 3.4 Verilog Generation

Covoh provides tools to generate Verilog code from verified designs for evaluation and deployment. The generation is a 2-step operation. We first compile Coq functional descriptions into a netlist format. Then a Python script converts the netlist to a Verilog module by adding appropriate decorations via string operations. The script also selects the involved blocks from a list of predefined Verilog modules. Options allow the user to choose the data type and the need for a clock since Covoh supports proofs at various levels (bit vs integer, timeless vs clocked, etc.). Case studies in section 4 show the capability of Covoh to generate hundreds of lines of Verilog code from a concise description.

Fig. 3 briefly demonstrates the process of compiling a Covoh description of multiply–accumulate (MAC) block into a Verilog module via a netlist.

## 3.5 Workflow

We outline the general workflow for verification (Fig. 4). The goal is to start with a functional design description and finish with a safe and trusted Verilog design description of the optimization. The original description is transferred into optimized description by manual and automated tools (A). The Covoh repository of designs (I.) provides verified optimizations as reference and rewrite rules. The descriptions are simulated for numerical and symbolic inputs (B). Unmatched results indicate bugs in the optimization. Matched circuits are manually examined by performing parametric pen-and-paper proof (C). We provide automated validation of each step of the pen-and-paper proof in Covoh by using the appropriate algebraic law as a rewrite rule (D). Unproved results cannot proceed, hence requiring reconsideration of the optimization. A rewrite rule can be either selected from the repository of verification strategies (II.) or proved using automated tactics. A lemma, once verified, can then be added to the repository (II.) for later use. Parametric tuning over the functional description chooses the most appropriate design coefficients (E). Covoh provides tools for generating structural Verilog code (F) from the description. Finally, FPGA design software helps synthesize the circuit for further evaluation and implementation (G). Designs failing to meet the constraints are sent back for rewriting the optimization.

While tools like Isabelle [18] and ACL2 [9] can support the proposed design flow, we demonstrate that Coq provides a simple,

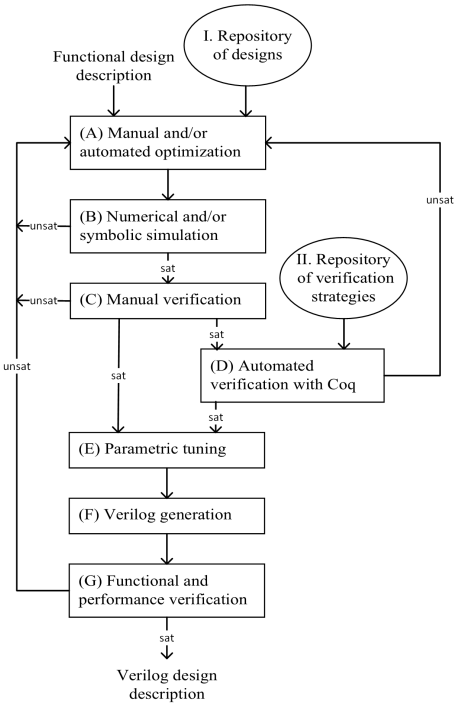


**Figure 4: General workflow for Covoh verification (sat/unsat: satified/unsatisfied)**

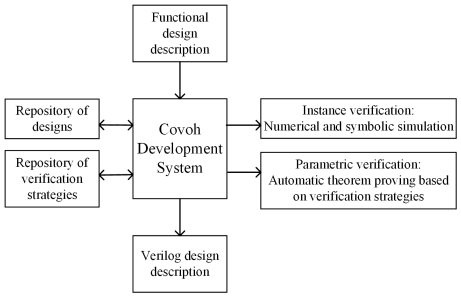


**Figure 5: The Covoh development system**

easy-to-use development system covering both instance and parametric verification (Fig. 5). Covoh contains repositories of designs and verification strategies to facilitate design development, avoid starting designs from scratch, and maximize the reuse of design and verification efforts. A list of Covoh definitions, pre-proved lemmas, and verification strategies are provided in our repositories for compositions, wires, simple and recursive higher-order functions, and delays.

# 4 CASE STUDIES

This section covers verifying and evaluating the optimizations of Horner's Rule and avionics monitoring, and compares Covoh with other approaches. Our approach shows efficient ways of verification via symbolic simulation and parametric methods. Section 4.1 and 4.2 cover the steps (A), (B), (C), and (D), and section 4.3 covers the steps (E), (F), and (G) in the Covoh verification workflow (Fig. 4).

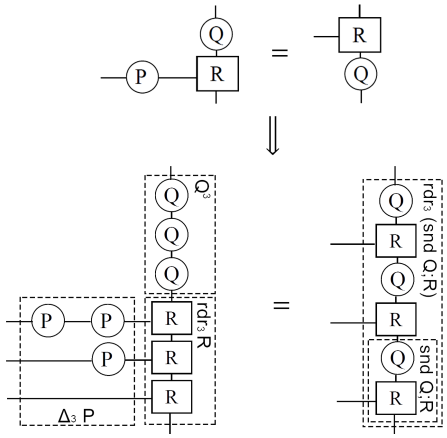


**Figure 6: Horner's Rule: an optimization from $O(n^2)$ to $O(n)$**

## 4.1 Horner's Rule

This method originates as a conditional optimization for polynomial evaluation, reducing the amount of multiplications from $n(n+1)/2$ to $n$. Note that this optimization is only valid if $ax + bx = (a+b)x$.

$$ax + bx = (a+b)x \implies$$
$$a_0 + a_1x + a_2x^2 + a_3x^3 = a_0 + x(a_1 + x(a_2 + a_3x)) \quad (1)$$

Horner's Rule can be used for optimizing and pipelining hardware design with a triangular array of blocks. The following expression and Fig. 6 show a general version of Horner's Rule (`Horners`) in terms of block description.

$$[P, Q]; R = R; Q \implies [\triangle_n P, Q^n]; \mathrm{rdr}_n R = \mathrm{rdr}_n(\mathrm{snd}\, Q\,; R) \quad (2)$$

The pre-condition for Horner's Rule holds if we substitute the blocks $P$ and $Q$ by a delay element (`@D` or $D$) such that $[D, D]$; $R$=$R$; $D$. Then, the right-hand circuit is a pipelined design obtained by retiming the left-hand circuit. If we replace $P$ and $Q$ with a 1-input block `mult x` which computes the product of $x$ with an input value, and replace $R$ with a 2-input adder (`ADD2`), then the circuits correspond to polynomial evaluation. We can confirm this result via symbolic simulation, which is an efficient method for instance verification. `polyval1` and `polyval2` are defined to the original and optimized circuits respectively.

```
Eval cbn in polyval1 3 x ([a0;a1;a2], a3).
  = a0 + (a1 * x + (a2 * x * x + a3 * x * x * x))
    : nat
Eval cbn in polyval2 3 x ([a0;a1;a2], a3).
  = a0 + (a1 + (a2 + a3 * x) * x) * x : nat
```

Parametric verification for the Horner's Rule using induction and rewrite tactics is supported by the repository of verification strategies. The same strategies are used for the second case study below, with more details of parametric verification.

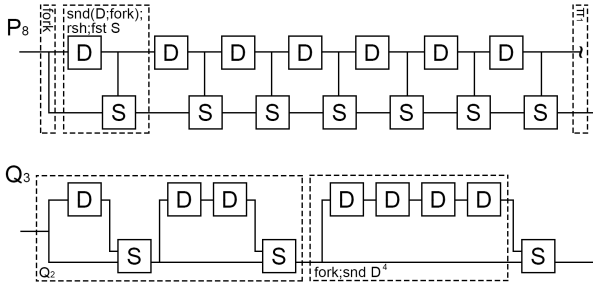


**Figure 7: Optimization for avionics monitoring: $O(2^n)(P_8)$ to $O(n)(Q_3)$.**

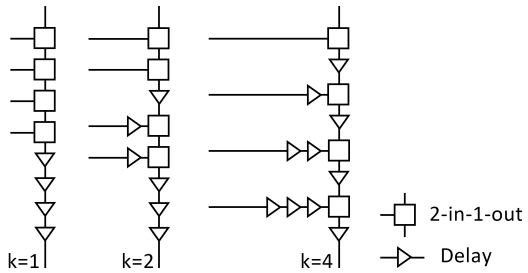


**Figure 8: Example ($n = 4$) of using Horner's Rule for partial pipelining at different levels: $k = 1$ shows no pipelining, and $k = n$ shows full pipelining**

## 4.2 Avionics monitoring

Runtime verification for avionics can be achieved by in-circuit temporal monitors [22]. Avionics is safety-critical, so it would benefit from verification to increase confidence in the correctness of optimizations.

An optimization has been proposed for such in-circuit monitors, which reduces the amount of its combinational elements from $O(2^n)$ to $O(n)$. The proof applies to not only adders for temporal monitors, but also any associative operator S (2-input-1-output), as shown in Fig. 5. We believe this is the first mechanized proof of this optimization.

Consider the following parametric designs $P$ and $Q$, where $Q_0$ is defined as identity (`id`). $Q_n$ is an optimization of $P_{2^n}$ since $P_{2^n}$ involves $(2^n - 1)$ $S$ blocks while $Q$ reduces its number to $n$. Fig. 7 shows these designs for $n = 3$.

$$P_m = \text{fork}; (\text{snd}(D; \text{fork}); \text{rsh}; \text{fst}\, S)^{m-1}; \pi_1 \tag{3}$$

$$Q_{i+1} = Q_i; \text{fork}; \text{snd} D^{2^i}; S \tag{4}$$

Parametric verification ensures the correctness for any value of $n$. Usually, designers will sketch a proof of the optimization with pen-and-paper, and then use Covoh's verification strategies and tactics to confirm the proof. We demonstrate this process by highlighting some key steps.

Mathematical induction is often used for proving regular hardware design since many higher-order functions are defined recursively. Coq supports proof by induction by unpacking recursive definitions into base case and inductive step. The proof proceeds by applying a series of algebraic laws. Coq validates the transformations of algebraic expressions by checking the conditions of previously-proved rewrite rules. One step of the proof by induction of the avionics monitoring design is illustrated below.

Base case: $P_1 = \text{fork}; \pi_1 = \text{id} = Q_0$.

Inductive step: $P_{2^{n+1}} = \ldots$

$$= \text{mfork}_{2^{n+1}}; \text{half}; [\triangle_{2^n} D; \text{apl}^{-1}_{2^n-1}; \text{rdl}_{2^n-1} R,$$
$$\triangle_{2^n} D; \text{map}_{2^n} D^{2^n}; \text{apl}^{-1}_{2^n-1}; \text{rdl}_{2^n-1} R]; R \tag{5}$$

$$= \text{mfork}_{2^{n+1}}; \text{half}; [\triangle_{2^n} D; \text{apl}^{-1}_{2^n-1}; \text{rdl}_{2^n-1} R,$$
$$\triangle_{2^n} D; \text{apl}^{-1}_{2^n-1}; \text{rdl}_{2^n-1} R; D^{2^n}]; R \tag{6}$$

$= \ldots = Q_{n+1}$ □

The use of the rewrite rule `pushD` is highlighted above. Due to the commutative property of delays with combinational blocks, a pre-proved lemma states that a rectangle block of delays connected to the inputs of a multi-input-single-output circuit can be pushed to its output. This lemma converts expression (5) to (6). Covoh provides a simple environment for verification, with the rewrite rule `pushD` defined as,

```
pushD : forall (m : nat) (circuit : t A m -> A),
  mapn m (@D ^^ m);; circuit = circuit;; @D ^^ m
```

Once we have reached expression (5), the "`rewrite pushD`" tactic can be used for its transformation. The tactic matches the LHS of the rewrite rule (in red) and replaces it with the RHS (in blue).

## 4.3 FPGA Evaluation

Covoh can compile optimized designs into structural Verilog RTL, which can then be processed by other backend tools. This subsection evaluates the performance of the optimizations by generating Verilog codes using Covoh and synthesising them using the Xilinx Vivado 2021.02 tool. FPGA power consumption is also estimated by the Xilinx Vivado. The Xilinx ZC702 evaluation board [10] is used, which consists of a Zynq-7000 XC7Z020 SoC.

*4.3.1 Horner's Rule for partial pipelining.* An extension of Horner's Rule is partial pipelining of right reduction. The optimization (section 4.1) is slightly altered to provide a method for pipelining any reduction of 2-input-1-output blocks. Fig. 8 gives an example of different levels of pipelining for four blocks, where $k$ indicates the level of pipelining or the amount of clusters. $k = 1$ indicates no pipelining, and $k = n$ indicates full pipelining. Covoh proves the structural equivalence, including cycles of latency, of three designs in Fig. 8. We usually remove the D-chain at the output to achieve a lower latency for the less pipelined designs.

We evaluate the effect of pipelining by estimating clock speed, dynamic power consumption, and the amount of resources. We use the right reduction (`rdr` in Fig. 2) of 128 XOR gates as a starting point and apply partial pipelining to produce designs with k pipeline stages, $1 \le k \le 128$. Designs are repeated 20 times to get a more accurate estimation of power consumption. The major benefit of higher levels of pipelining is the increase of design throughput, as reflected by the clock speed (Fig. 9). Unlike the optimization in section 4.1, the increase in pipelining causes a significant increase

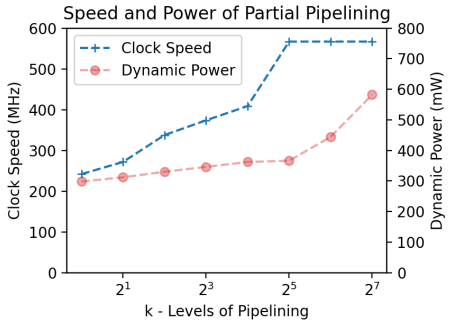


**Figure 9: Estimated clock speed and dynamic power consumption of right reduction of 128 XOR gates at different levels of pipelining**

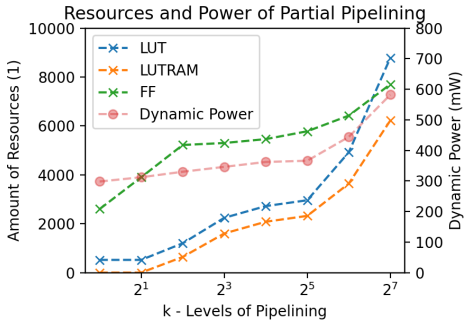


**Figure 10: Resources (FF/LUT/LUTRAM) used for FPGA synthesis and estimated dynamic power consumption of right reduction of 128 XOR gates at different levels of pipelining**

in resources (Fig. 10). There can be up to around $n^2/2$ delays (registers) for full pipelining ($k = n$), while no pipelining uses n delays. It is because Vivado automatically replaces the long chain of delays as a combination of registers, LUT, and LUTRAM. It almost flats the amount of FF for larger $k$ but significantly increases the amount of LUT/LUTRAM. Since the combinational blocks are becoming finer-grained, the opportunity of optimizing them before mapping into LUT resources has reduced, which leads to further increase in the amount of LUT/LUTRAM. Dynamic power consumption increases due to an increase in resources. However, since these values are estimated using Vivado tools, power consumption due to glitching may be underestimated. Previous work [23] [1] reports power reduction with increased pipelining, and the estimation of FPGA power consumption is known to be complicated [6]. Our purpose is to illustrate that our parametric descriptions cover a family of designs with different trade-offs. Varying a single parameter that controls the level of pipelining produces functionally correct designs with different trade-offs in throughput, latency, resource usage, power consumption, etc. The parametric tuning step (E) in Fig. 4 finds the most appropriate parameter value for a given application.

*4.3.2 Resources of avionics monitoring.* The optimization of the avionics monitoring focuses on reducing the amount of resources quantified by the amount of LUT and FF (flip-flops). We increase the

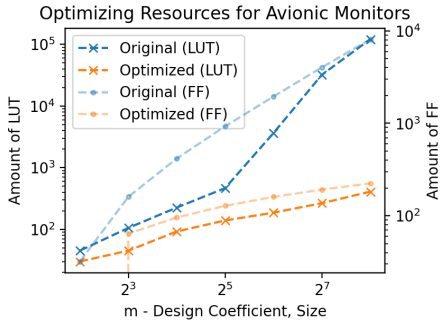


**Figure 11: Resources (LUT/FF) used for FPGA synthesis of the original and optimized designs for avionics monitoring**

**Table 1: Comparing Covoh with RubyZF**

| | Covoh | RubyZF |
|---|---|---|
| Series of parallel | 1[a] | 15 |
| Composition of parallel and `lsh` | 1 | 15 |
| Commutativity of `mapn` and `tri` | 10 | 19 |
| Simplified Horner's Rule | 9 | 16 |
| Honer's Rule [11] | 20 | 60 |
| Avionics monitoring [22] | 27 | N/A |

[a]Required steps (line of commands)

complexity by changing the datatype to 32-bit unsigned integers. $D$ is now an array of D-type flip-flops that support 32-bit input, and the associative operation $S$ is chosen to be a 32-bit multiplier. The user does not need to worry about the switching of datatype since all the Verilog codes are generated automatically by the Covoh compiler.

The experiment generates designs from expressions (3) and (4). It varies the value of $m$ as a power of two due to the exponential behaviours of the optimization. The amount of LUT and FF are plotted on a log-log scale (Fig. 11). Increasing $m$ from $2^2$ to $2^8$ increases the amount of LUT from around $10^2$ to $10^5$ in the original design, while the amount of LUT is well controlled at around $10^2$ in the optimized design. In theory, the amount of FF should be the same for both designs, however, the optimized design reports much fewer FF being used after synthesis. This is due to the Vivado's compile-time optimization for D-chain. Therefore, the amount of FF closely matches the amount of LUT in both cases. The amount of resources reported after synthesis confirms the optimization of avionics monitoring being $O(2^n)$ to $O(n)$.

## 4.4 Comparison with other formal tools

Let us compare Covoh with other tools. RubyZF [18] is the legacy solution for verifying Ruby hardware designs. One driving force of creating a new framework for Ruby-like design language is the difficulties of proving complex designs. Several optimizations have been confirmed in both Covoh and RubyZF with the required steps (line of commands), as shown in Table 1. For basic designs, where Covoh benefits from its automation tactics, it is up to 15 times simpler

**Table 2: Features of Hardware Verification System**

| Name of tool | Host system | Verification | Symbolic simulation | VHDL/Verilog/ floorplan |
|---|---|---|---|---|
| Covoh | Coq | ✓ | ✓ | ✓ |
| RubyZF [18] | Isabelle | ✓ | | |
| $\mu$FP [13] | Orwell | | ✓ | ✓ |
| Rebecca [8] | SML | | ✓ | ✓ |
| Fe-Si [4] | Coq | ✓ | | ✓ |
| Quartz [17] | Isabelle | ✓ | | ✓ |
| Silver Oak [21] | Coq | ✓ | | ✓ |

than RubyZF. Besides, Covoh can save more than 40 commands for more advantageous designs, which requires considerable human effort in any interactive proof assistant. We believe the successful verification of avionics monitoring [22] is the first machine-aided formal proof of this optimization. RubyZF has not been used in verifying such complicated designs.

We also compare the supported features for Covoh and other related work in Table 2, which shows the advantages of our approach. Covoh is capable of supporting all three methods of verification, while none of the other approaches support all three of them. The main strength of Covoh is its capability of supporting both instance and parametric verification.

## 4.5 Repository of verification strategies

Our repository (Fig. 5) facilitates verification of the above designs. First, many designs, like the two discussed in this section, follow a similar development flow. Our repository contains customizable templates that help proof development. Second, the repository contains generic rewrite rules to automate common proof steps. One example is a simplification rule based on composing rewrite tactics, in which the expression $R; R^{-1}$ will be removed if detected. Such generic rewrite rules reduce the amount of proof steps by half for the two case studies.

# 5 CONCLUSION

Covoh is designed to lower the barrier to verification of optimization for producing efficient acceleration by providing a modular and robust development system. Numerical and symbolic simulation can help debug at an early stage of design and can be used for instance verification. Repositories provide pre-proved lemmas, designs, and templates to reduce verification efforts. Compilation to structural Verilog ensures the practical implementations of verified circuits.

Further work on Covoh includes the following. First, extend Covoh repositories with further designs and verification strategies for a variety of applications. Second, explore how Covoh can be interfaced with other verification systems, such as those based on SMT solvers [12], which can enhance the automatic verification and error detection capabilities of Covoh. Third, study how Covoh can be integrated with other synthesis and visualization tools to provide an end-to-end hardware development system for teaching and research.

# ACKNOWLEDGMENTS

The support of the UK EPSRC (No. EP/N031768/1, EP/P010040/1, EP/V028251/1 and EP/S030069/1), Intel and Xilinx is gratefully acknowledged.

# REFERENCES

[1] Steve Bard and Nader I. Rafla. 2008. Reducing power consumption in FPGAs by pipelining. In *2008 51st Midwest Symposium on Circuits and Systems*. 173–176. https://doi.org/10.1109/MWSCAS.2008.4616764
[2] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.
[3] Thomas Braibant. 2011. Coquet: a Coq library for verifying hardware. (2011). https://hal.inria.fr/inria-00611757 working paper or preprint.
[4] Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) *(CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 213–228.
[5] Edmund M. Clarke and E. Allen Emerson. 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–71. https://doi.org/10.1007/BFb0025774
[6] Intel Corporation. 2017. *Understanding and Meeting FPGA Power Requirements White Paper*. Technical Report.
[7] Michael J. C. Gordon. 1985. Why higher-order logic is a good formalism for specifying and verifying hardware.
[8] Shaori Guo and Wayne Luk. 2001. An Integrated System for Developing Regular Array Designs. *J. Syst. Archit.* 47, 3–4 (apr 2001), 315–337. https://doi.org/10.1016/S1383-7621(00)00052-7
[9] Warren Hunt, Matt Kaufmann, J Moore, and Anna Slobodova. 2017. Industrial hardware and software verification with ACL2. *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences* 375 (10 2017), 20150399. https://doi.org/10.1098/rsta.2015.0399
[10] Xilinx Inc. 2019. *ZC702 Evaluation Boardfor the Zynq-7000 XC7Z020 SoC - User Guide*. Technical Report.
[11] Geraint Jones and Mary Sheeran. 1990. *Circuit design in Ruby*. North-Holland, 13–70.
[12] Florian Lonsing, Karthik Ganesan, Makai Mann, Srinivasa Shashank Nuthakki, Eshan Singh, Mario Srouji, Yahan Yang, Subhasish Mitra, and Clark Barrett. 2019. Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED: Invited Paper. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942096
[13] Wayne Luk, Geraint Jones, and Mary Sheeran. 1990. *Computer-Based Tools for Regular Array Design*. Prentice-Hall, Inc., USA, 589–598.
[14] Simon Marlow. 2010. *Haskell 2010 language report*. Technical Report.
[15] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G. Daly, Dillon Huff, and Pat Hanrahan. 2018. CoSA: Integrated Verification for Agile Hardware Design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–5. https://doi.org/10.23919/FMCAD.2018.8603014
[16] Oliver Pell. 2006. Verification of FPGA Layout Generators in Higher-Order Logic. *J. Autom. Reason.* 37, 1–2 (aug 2006), 117–152. https://doi.org/10.1007/s10817-006-9039-9
[17] Oliver Pell and Wayne Luk. 2005. Quartz: a framework for correct and efficient reconfigurable design. In *2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*. 8 pp.–14. https://doi.org/10.1109/RECONFIG.2005.32
[18] Ole Rasmussen. 1996. Ensuring Correctness of Ruby Transformations. In *Proceedings of the 3rd Workshop on Designing Correct Circuits*. https://doi.org/10.14236/ewic/DCC1996.11
[19] Dustin Richmond, Alric Althoff, and Ryan Kastner. 2018. Synthesizable Higher-Order Functions for C++. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2835–2844. https://doi.org/10.1109/TCAD.2018.2857259
[20] Mary Sheeran. 1984. MuFP, a Language for VLSI Design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 104–112. https://doi.org/10.1145/800055.802026
[21] Satnam Singh. 2021. *Silver Oak*. https://github.com/project-oak/silveroak
[22] Tim Todman, Stephan Stilkerich, and Wayne Luk. 2015. In-Circuit Temporal Monitors for Runtime Verification of Reconfigurable Designs *(DAC '15)*. Association for Computing Machinery, New York, NY, USA, Article 50, 6 pages. https://doi.org/10.1145/2744769.2744856
[23] Steven J. E. Wilton, Su-Shin Ang, and Wayne Luk. 2004. The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays. In *Field Programmable Logic and Application*, Jürgen Becker, Marco Platzner, and Serge Vernalde (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 719–728.