Vectorization of Verilog Designs and its Effects on Verification and Synthesis

Maria Fernanda Oliveira Guimarães
UFMG
Belo Horizonte, Brazil
maria.guimaraes@dcc.ufmg.br

Ulisses Drumond Souza Rosa
UFMG
Belo Horizonte, Brazil
ulissesrosa@dcc.ufmg.br

Ian Trudel
IanWorld
Montreal, Canada
ian.trudel@researchcenter.io

João Victor Amorim Vieira
Cadence
Belo Horizonte, Brazil
jvamorim@cadence.com

Augusto Mafra
Cadence
Belo Horizonte, Brazil
augusto@cadence.com

Mirlaine Crepalde
Cadence
Belo Horizonte, Brazil
mirlaine@cadence.com

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

***Abstract*—Vectorization is a compiler optimization that replaces multiple operations on scalar values with a single operation on vector values. Although common in traditional compilers such as rustc, clang, and gcc, vectorization is not common in the Verilog ecosystem. This happens because, even though Verilog supports vector notation, the language provides no semantic guarantee that a vectorized signal behaves as a word-level entity: synthesis tools still resolve multiple individual assignments and a single vector assignment into the same set of parallel wire connections. However, vectorization brings important benefits in other domains. In particular, it reduces symbolic complexity even when the underlying hardware remains unchanged. Formal verification tools such as Cadence® Jasper®, Synopsys VC Formal, and OneSpin operate at the symbolic level: they reason about Boolean functions, state transitions, and equivalence classes, rather than about individual wires or gates. When these tools can treat a bus as a single symbolic entity, they scale more efficiently. This paper supports this observation by introducing a Verilog vectorizer. The vectorizer, built on top of the CIRCT compilation infrastructure, recognizes several vectorization patterns, including inverted assignments, assignments involving complex expressions, and inter-module assignments. It has been integrated into production as part of the Jasper Formal Verification platform, where it improves verification time by 28.12% and reduces memory consumption by 51.30% on 1,157 designs from the ChiBench collection.**

***Index Terms*—verilog, vectorization, verification, optimization**

I. Introduction

Vectorization is a classic compiler optimization that replaces scalar operations with equivalent operations over wider data types when such widening preserves program semantics [1]. It is used in software compilers such as `gcc` and `clang`, particularly at the `-O2` and `-O3` optimization levels, to improve runtime performance and reduce code size [2].

In contrast, vectorization is rarely applied in compilers and tools that process Verilog designs. At the hardware level, vectors are ultimately realized as independent wires and bit-level logic, so synthesis tools lower vector signals into scalars early in the flow. This practice has created a common assumption in the EDA community that recovering vector structure offers little benefit. Consequently, mainstream Verilog processing frameworks such as CIRCT and Yosys do not provide analyses or transformations that group related scalar signals back into structured vectors.

However, the value of vectorization in the hardware context does not stem from making the resulting circuits "vectorial." Instead, vectorization reduces the symbolic complexity of the design representation, enabling the tool flow to operate on a more compact and structured intermediate form. This simplification accelerates elaboration, optimization, and formal reasoning, while also reducing memory consumption. In this context, this paper advances the following observation:

> *"Although the vectorization of Verilog designs does not change the hardware they describe, it reduces their symbolic complexity, enabling faster and more scalable analysis and verification."*

To support this observation, Section II presents three classes of vector transformations for Verilog: signal permutation, inter-module grouping, and structural vectorization. Section III then shows how to implement these transformations using static analyses and semantics-preserving compiler passes. Finally, Section IV describes a vectorizer implemented within the CIRCT infrastructure, as a pass over the Hardware dialect of the Multi-Level Intermediate Representation (MLIR) [7]. We have evaluated this implementation over a collection of 1,157 designs from the ChiBench [10] collection. Section IV-C shows that vectorization improves JasperGold's verification time by over 28% and reduces its memory consumption by more than 50%. Section IV-D shows that vectorization also reduces Genus's elaboration time by more than 5% on average, with much larger improvements on individual benchmarks. These benefits have led to the adoption of this transformation in Cadence Design Systems' internal code generation flows.

II. Overview

Vectorization of Verilog designs is the process of detecting groups of bit-level assignments or operations that behave collectively as word-level expressions and replacing them with

equivalent vector operations. The implementation of vectorization discussed in this paper handles any combination of three classes of transformations, which Figure 1 illustrates. Case (a) shows *general bit permutations*, where arbitrary reorderings of signals are compacted into a single vector assignment. Case (b) shows *intermodule vectorization*, in which structurally similar module instances are selectively inlined to expose vectorization opportunities across module boundaries. Case (c) shows *complex patterns* that combine logic or arithmetic operations on individual bits into a single vector-level expression. The proposed vectorizer handles any combination of these cases, including, for example, intermodule vectorization of permutations involving complex arithmetic and logical expressions. Section III shall explain how the patterns that enable each of these three cases can be identified, and how they can be transformed into vectorized assignments.

(**a**) General bit permutation

```
module bit_mixing_vectorization(output wire [3:0] out, input wire [3:0] in);
    assign out[3] = in[0];
    assign out[1] = in[2];
    assign out[2] = in[3];
    assign out[0] = in[1];
endmodule
```

```
module bit_mixing_vectorization(output wire [3:0] out, input wire [3:0] in);
    assign out = {in[0], in[3:1]};
endmodule
```

(**b**) Intermodule vectorization

```
module intermodule_vectorization(output wire [3:0] out, input wire [3:0] in);
    mybuf buf_inst1(out[3], in[3]);
    mybuf buf_inst2(out[2], in[2]);
    mybuf buf_inst3(out[1], in[1]);
    mybuf buf_inst4(out[0], in[0]);
endmodule
module mybuf(output wire o, input wire i);
    assign o = i;
endmodule
```

```
module intermodule_vectorization(output wire [3:0] out, input wire [3:0] in);
    assign out = in;
endmodule
```

(**c**) Complex patterns

```
module pattern_recognition(
    output wire [3:0] result, input wire [3:0] a, b, input wire sel
);
    assign result[3] = (a[3] & sel) | (b[3] & ~sel);
    assign result[2] = (a[2] & sel) | (b[2] & ~sel);
    assign result[1] = (a[1] & sel) | (b[1] & ~sel);
    assign result[0] = (a[0] & sel) | (b[0] & ~sel);
endmodule
```

```
module pattern_recognition(
    output wire [3:0] result, input wire [3:0] a, b, input wire sel);
    wire [3:0] _GEN = {4{sel}};
    assign result = a & _GEN | b & ~_GEN;
endmodule
```

Fig. 1. Examples of vectorization patterns. The upper, white box, shows Verilog syntax that enables vectorization. The lower gray box shows the syntax transformed after vectorization. (a) General bit permutation. (b) Intermodule vectorization. (c) Complex logic and arithmetic patterns.

III. Implementation

The bit-level vectorization of Verilog designs described in this paper is implemented on top of **CIRCT** (Circuit Intermediate Representation Compilers and Tools), and follows the flow seen in Figure 2. Notice that although the analyses and optimizations happen on the Hardware dialect of MLIR, users see them as a source-to-source transformation, as HW specifications can be translated back to Verilog. Thus, vectorization has the additional benefit of improving code readability. The rest of this section details the core components of that pipeline.

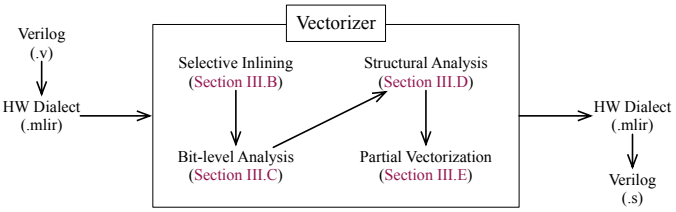

Fig. 2. The implementation of vectorization in the CIRCT Framework.

A. CIRCT: The Underlying Infrastructure

CIRCT is an open-source compiler infrastructure for hardware design that supports languages such as Verilog and Chisel. Built on MLIR [7] and LLVM [6], it enables the analysis, optimization, and transformation of hardware descriptions for synthesis, simulation, and other downstream tools. CIRCT comprises multiple domain-specific intermediate representations, each implemented as an MLIR dialect. Translators convert one dialect into another, allowing users to select the most suitable abstraction for each task. Vectorization takes place within the *Hardware (HW) dialect*.

The `hw` dialect provides the structural backbone of CIRCT, defining abstractions for hardware composition. It introduces data types such as fixed-width integers, arrays, and structs, and specifies module boundaries and ports while abstracting away internal logic. Combinational and sequential behavior are expressed through specialized dialects such as `comb` and `seq`. For example, `hw` defines modules (`hw.module`), instantiations (`hw.instance`), and wires (`hw.wire`), while `comb` provides logic operations such as `comb.and`, `comb.add`, and `comb.mux`.

Example 1: Figure 3(a) shows a simple Verilog assignment, and Figure 3(b) shows its equivalent specification in CIRCT. When lowered to the `hw` and `comb` dialects, the design is represented structurally as four independent bit-level connections. Each `comb.extract` operation isolates one bit from the input vector, and each bit is wired independently to the corresponding output position. The HW representation inherits some conventions from the LLVM ecosystem. For instance, the program in Figure 3(b) is in Static Single Assignment (SSA) form [3], meaning that each variable is defined exactly once. Moreover, every value is explicitly typed; for example, `i1` (a single bit) is the type of variable `%out_3`.

B. Selective Inlining

The analyses and transformations described in Sections III-C, III-D, and III-E are *intra-module*; that is, they operate within the boundaries of a single Verilog module. To enable *inter-module* vectorization, we employ *selective inlining* of submodules. Inlining consists of copying the body of a module into the point where it is instantiated. By "selective," we mean that a module may be inlined at some instantiation sites but not at others. This approach is necessary because full

```
module simple_ex(
  output wire [3:0] out,
  input wire [3:0] in
);
  assign out[3] = in[3];
  assign out[2] = in[2];
  assign out[1] = in[1];
  assign out[0] = in[0];
endmodule
```

(a)

```
hw.module @simple_ex(
  %in: i4,
  %out: !hw.output<i4>
) {
  %in_3 = comb.extract %in from 3: (i4)->i1
  %in_2 = comb.extract %in from 2: (i4)->i1
  %in_1 = comb.extract %in from 1: (i4)->i1
  %in_0 = comb.extract %in from 0: (i4)->i1

  %out_3 = hw.output %in_3: i1
  %out_2 = hw.output %in_2: i1
  %out_1 = hw.output %in_1: i1
  %out_0 = hw.output %in_0: i1
}
```

(b)

Fig. 3. (a) Simple Verilog design. (b) Equivalent specification in CIRCT.

inlining is impractical: it can cause code bloat and substantially increase compilation time.

Determining whether an inlining decision will contribute to vectorization is not possible without performing the inlining itself and rerunning the analyses from Sections III-C, III-D, and III-E. To avoid such iterations and keep compilation time low, inlining decisions are guided by two heuristics that estimate module regularity and recursive size:

- **Regularity:** The `isHighlyRegular` analysis determines whether the callee module consists solely of simple combinational logic (e.g., `comb.and`, `comb.or`, `comb.extract`, `comb.concat`).
- **Size:** The `getRecursiveSize` analysis computes the total number of operations in the callee, including those in any instantiated submodules. Inlining is permitted if the cumulative operation count remains below 150 (measured in CIRCT's `hw` dialect).

The emphasis on regularity arises because the analyses described in Sections III-C and III-D can only reason about modules composed of simple combinational operations.

C. Bit-Level Dataflow Analysis

This static analysis identifies groups of assignments that implement bit permutations, such as in Figure 1(a). It propagates bit origins through chains of assignments. The analysis succeeds if all bits of an output value can be traced back to a single source vector and form a valid one-to-one permutation.

The pass maintains a bit-to-source mapping structure (inspired by `BitArray`), which associates each result bit with a source value and source index. When successful, it applies the most efficient transformation depending on the permutation:

- For the identity permutation (`out[i] = in[i]`), the output vector is replaced directly by the input vector.
- For a full reversal (`out[i] = in[N-1-i]`), the result is replaced by a single `comb.ReverseOp`.
- For any other permutation (i.e., a "shuffle"), the vectorizer emits the minimal sequence of `comb.extract` and `comb.concat` operations, as seen in Figure 1 (a).

This analysis is fast but conservative: it does not trace through logical operations (e.g., `comb.and`, `comb.mux`). If it cannot prove that the bits originate from a single vector or pass only through supported operations, the vectorizer falls back to the structural analysis introduced in Section III-D.

D. Structural Analysis

A *logic cone* is the set of upstream operations that influence a single scalar output bit. Structural Analysis determines whether a set of bit-level operations can be safely collapsed into a single vector operation by verifying two conditions:

1) The cones of all output bits are *independent* (no cross-bit dependencies), and
2) The cones are *isomorphic* (identical structure with a uniform bit stride).

When these conditions hold, the vectorizer replaces N replicated scalar operations with a single vector-level operation that preserves the original hardware behavior. For a vector-typed output value with N bits, the analysis constructs N logic cones and performs the following checks:

- **Safety:** Cones must not overlap; shared intermediate values would indicate bit-to-bit dependencies (e.g., carry chains), which prevent vectorization.
- **Equivalence:** Cones must exhibit the same operator structure and consistent index progression across the input vectors (e.g., $a[i]$, $b[i]$, $c[i]$).

If the cones are structurally isomorphic and semantically independent, then they are merged and lowered to the most concise vector operation available in the target dialect.

Example 2: Each output bit in Figure 1(c) is computed using the same logical structure applied independently to the corresponding bits of two input buses. For a bit-vector output out, the *logic cone* of $out[i]$ is the complete set of upstream operations that may affect its value. Structural Analysis first ensures that cones are *independent*:

$$Cone(out[i]) \cap Cone(out[j]) = \emptyset \quad \text{for all } i \neq j.$$

Next, it checks that all cones are *isomorphic*:

$$Cone(out[0]) \cong Cone(out[1]) \cong \ldots \cong Cone(out[N-1]),$$

meaning they contain identical operator structure and width progression. This equivalence check performs a subgraph isomorphism test that verifies: (i) the same operators appear in each cone, (ii) with the same structural connectivity and operand ordering, and (iii) bit extraction follows a consistent index stride (e.g., $a[i]$, $b[i]$, control signals). In this case, all cones satisfy independence and isomorphism, enabling the replacement of N replicated bitwise expressions with a single vector-level operation (e.g., a conditional select).

E. Partial Vectorization

If the Bit-Level and Structural Analyses fail to vectorize the full output bus, the vectorizer attempts a best-effort approach. Instead of requiring the entire vector to follow a recognized pattern, this strategy discovers *contiguous sub-ranges* (hereafter, *chunks*) that can be vectorized independently.

The algorithm scans the output vector from the Most Significant Bit (MSB) to the Least Significant Bit (LSB) using a sliding window. For each bit position i, it searches for the largest suffix $out[i : j]$ that satisfies either the Bit-Level or Structural vectorization criteria. If such a chunk is

found, it is vectorized immediately, and the scan resumes at bit $j-1$. If no valid chunk is found, the window size is reduced and the search continues. This greedy segmentation ensures polynomial-time behavior while extracting the maximum vectorizable structure available. Its worst-case time complexity is $\mathcal{O}(N^2 \cdot T_{\text{check}}^{\max})$, which simplifies to $\Theta(N^3)$ when the vectorization predicate runs in linear time. However, in practice, incremental checking and greedy chunk discovery typically yield near-$\mathcal{O}(N^2)$ behavior.

Example 3: Even if a full 8-bit permutation does not match a known pattern, partial vectorization may still discover that: $out[7:4] \leftarrow \texttt{reverse}(in[3:0])$ and $out[3:0] \leftarrow \texttt{reverse}(in[7:4])$. In this case, two 4-bit chunks are vectorized independently, and the final result is reconstructed using a single `comb.ConcatOp`.

IV. Evaluation

This section evaluates the following research questions:

- **RQ1:** What is the impact of vectorization on the instruction count of the CHIBENCH benchmarks?
- **RQ2:** What are the absolute running time and asymptotic behavior of automatic vectorization?
- **RQ3:** What is the impact of vectorization on Jasper's running time and memory consumption?
- **RQ4:** What is the impact of vectorization on Genus's running time and memory consumption?

Before addressing each research question, we describe the experimental setup below.

Hardware: All experiments were conducted on an Intel Core i7-6700T processor with 8 GB of RAM, two L1 caches (128 KB each), one L2 cache (1 MB), and one L3 cache (8 MB), running Ubuntu Linux 22.04.1 LTS.

Software: Vectorization was implemented on top of CIRCT Release 192 (Oct-25). Section IV-C uses the Jasper Formal Verification Platform release 2023.12p002, and Section IV-D uses the Genus Synthesis Solution version 21.19-s055_1.

Benchmarks: Experiments use the CHIBENCH collection of Verilog designs [10]. CHIBENCH contains 49,599 designs mined from open-source repositories under permissible licenses. Among these, 1,157 designs exhibited vectorization opportunities. Unless stated otherwise, the numbers reported in this section refer to this subset of 1,157 Verilog files.

Correctness: JasperGold has been used to demonstrate semantic equivalence between each one of the 1,157 vectorized and original designs.

A. RQ1: Instruction Count

The size of Verilog designs in CIRCT's HW dialect can be measured by the number of MLIR instructions necessary to represent them. One of the effects of vectorization is that multiple scalar instructions may be replaced by a single vectorized instruction. Therefore, the more extensive the vectorization, the greater the expected reduction in instruction count. This subsection investigates this effect.

Discussion: Figure 4 compares the number of instructions before and after vectorization. Among the 1,157 designs that presented vectorization opportunities, 904 experienced code-size reduction, 8 showed an increase in instruction count, and 245 remained unchanged, even though they were transformed. Not every application of vectorization leads to a smaller design, because maintaining semantic equivalence can requires the insertion of auxiliary operations or intermediate wires. For instance, when signals are assigned in a non-contiguous or mixed order, the vectorizer must generate concatenation or extraction operations that may offset the instruction savings.

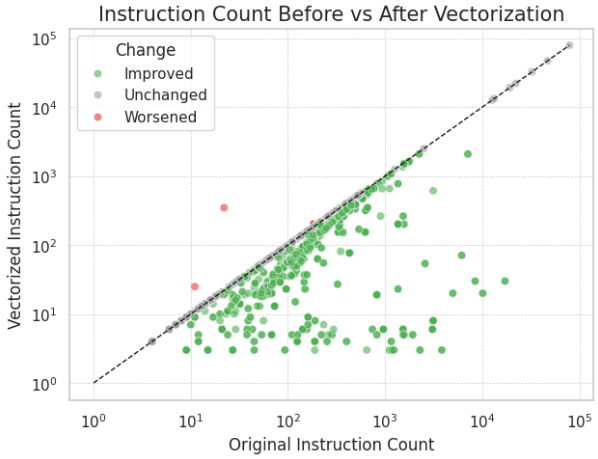

Fig. 4. Scatter plot comparing instruction counts before and after vectorization. Points below the diagonal indicate reductions, points above indicate increases, and points on the diagonal indicate no change.

Figure 5 shows the distribution of instruction-count reductions among the 904 designs that improved. Most reductions range between 40% and 50%, but some designs achieved nearly 100% reduction. The four most dramatic improvements were observed in the following benchmarks: `Chi_11888`: 3,837 instructions reduced to 3 (99.92%); `Chi_11336`: 2,283 reduced to 3 (99.87%); `Chi_1203`: 17,149 reduced to 30 (99.82%); and `Chi_11887`: 3,069 reduced to 6 (99.80%). These are extreme cases; overall, we expect vectorization to yield more moderate gains.

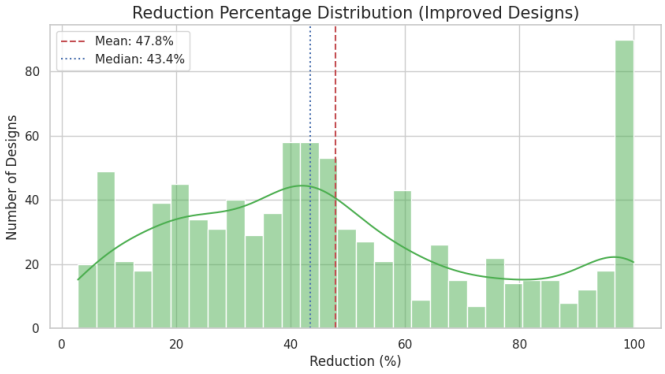

Fig. 5. Histogram of instruction-count reductions across designs that improved. Dashed and dotted lines indicate the mean (47.8%) and median (43.4%) reductions, respectively.

B. RQ2: Compilation Time

This section investigates the running time of the vectorization pass and its asymptotic behavior as a function of design size. This evaluation will demonstrate that the proposed transformation is sufficiently practical to be used in production.

Discussion: Figure 6 shows the asymptotic behavior of the vectorization pass by plotting its runtime (in seconds) as a function of the number of vectorized HW instructions on a log-log scale. The results show a linear trend, with a coefficient of determination $R^2 = 0.982$, confirming that runtime grows proportionally to design size. In absolute terms, vectorization is very fast, often orders of magnitude faster than Jasper's verification phase or Genus' synthesis step.

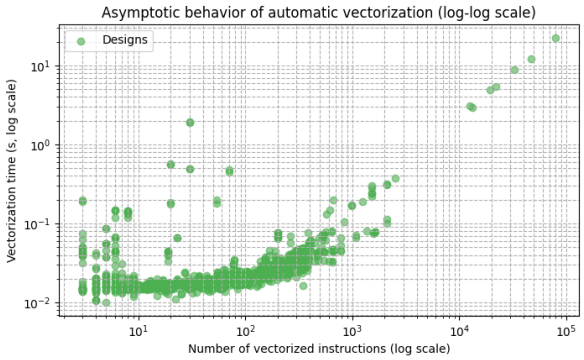

Fig. 6. Asymptotic behavior of the vectorization pass in a log-log scale. The data points form an approximately straight line with slope ≈ 1, further supporting the linear $O(n)$ time complexity.

C. RQ3: Impact on Jasper's Running Time and Space

The hypothesis of this paper is that vectorization, while not changing the underlying hardware, reduces symbolic complexity, which in turn enables faster and more scalable formal verification. This subsection evaluates this claim by measuring the impact of vectorization on the runtime and memory consumption of the HDL elaboration in Jasper.

Discussion: Figure 7 shows the impact of vectorization on Jasper's memory usage in the HDL elaboration step across the 1,157 benchmark designs. While the Original designs exhibit a wide distribution with heavy-tailed outliers, the Vectorized versions are tightly clustered around a much lower median memory footprint.

A similar pattern is observed for HDL elaboration runtime, as shown in Figure 8. It demonstrates that vectorization eliminates the high-runtime outliers present in the original designs. On average, vectorization reduced elaboration's memory consumption by 51.30% and verification runtime by 28.12%.

D. RQ4: Impact on Genus's Running Time and Memory Usage

The Cadence Genus Synthesis Solution is a physically aware RTL-to-gate synthesis tool that optimizes digital designs for power, performance, and area while ensuring timing closure. It processes a Verilog design through a two-stage pipeline:

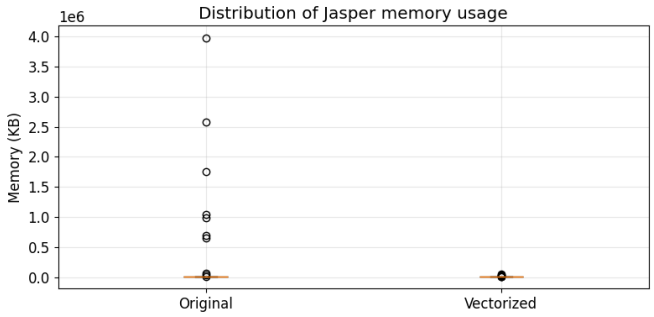

Fig. 7. Distribution of Jasper HDL elaboration memory usage (KB) for the original and vectorized versions of 1,157 Verilog designs.

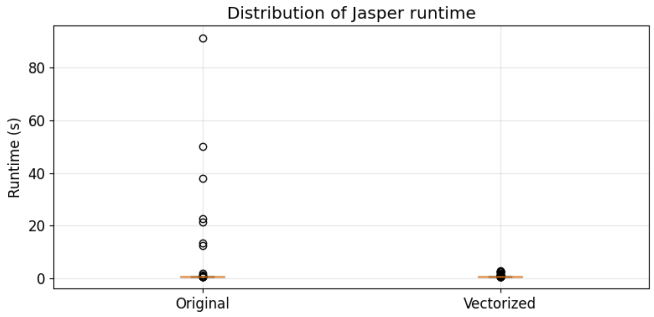

Fig. 8. Distribution of Jasper HDL elaboration runtime (s) for the original and vectorized versions of 1,157 Verilog designs.

- **Elaboration**: This initial phase parses the Register-Transfer Level (RTL) code, resolves design hierarchy and constructs, and translates the high-level description into a gate-level netlist.
- **Synthesis**: This phase performs physically aware optimizations, timing closure, and Design-for-Test (DFT) insertion to generate a gate-level netlist optimized for power, performance, and area.

Vectorization improves both phases in terms of memory usage and runtime. Memory is optimized because vectorized assignments are represented as single objects rather than as potentially large sequences of bit-level operations. Reducing the number of assignments also accelerates processing. For example, when optimizing data paths, Genus can operate on entire vectors instead of individual bits. This section evaluates the impact of these improvements.

Discussion: Vectorization brings modest benefit to Genus. On average, across 1,157 designs, Genus consumes 381 MB of memory per design (maximum resident memory). The average elaboration time is 0.094 seconds, and the average synthesis time is 1.351 seconds. We observe a geometric-mean improvement in memory consumption of 0.12%, elaboration time improves by 5.49%, and no statistically significant improvement is observed for synthesis time.

However, individual benchmarks reveal large speedups. Figure 9 shows the 20 largest improvements in Genus memory consumption, elaboration time, and synthesis time. In one

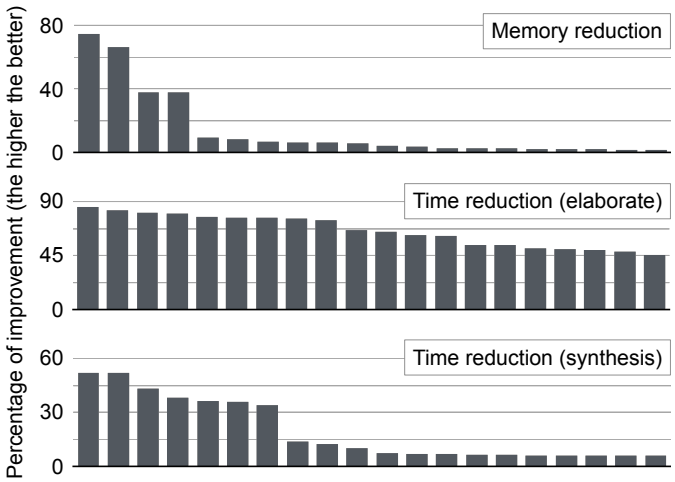

Fig. 9. The 20 largest observed improvements across three phases of the Genus pipeline.

ChiBench design (from https://github.com/awai54st/LUTNet), memory consumption dropped by 75%. In another design (from https://github.com/ehw-fit/evoapproxlib), synthesis time improved by 52%. Improvements in elaboration time are more consistent across benchmarks: the 20 best improvements range from 45% to 85%, with a geometric mean of 64.5%.

V. Related Work

To the best of our knowledge, no commercial or open-source RTL toolchain, including Yosys, CIRCT, JasperGold®, or Genus®, performs vectorization as a semantics-preserving RTL-to-RTL transformation. This type of transformation is also absent in HLS compilers such as LegUp, Bambu, and Xilinx Vivado, which instead apply vectorization at the software level to improve hardware generation. Nevertheless, our work connects to several research areas, including word-level reasoning for bit-vectors, recovery of word-level structure from bit-level/netlist representations, and algebraic rewriting techniques for structural circuit simplification.

Word-level reasoning and bit-vector solvers. Reasoning at the word (bit-vector) level can be far more efficient than bit-blasting in many verification tasks, leading to a long line of solvers and interpolation techniques that preserve word-level operators [4], [11]. These works optimize the internal representations used by SMT and equivalence checking engines, whereas our method transforms Verilog source to *expose* word-level structure before those engines operate.

Recovering word-level datapaths from bit-level nets. Several efforts address the inverse problem: given a low-level Boolean network (e.g., LUT or AIG netlist), reconstruct high-level words, carry chains, and datapath operators [8], [9]. These "word recovery" and reverse-engineering techniques rely on pattern matching and structural heuristics to detect specific arithmetic or multiplexer patterns. While conceptually related to the recognition component of our vectorizer, they are typically less general, operate after bit-blasting, and do not provide a semantics-preserving RTL-to-RTL rewrite.

Algebraic rewriting, AIG/FRAIG techniques, and structural compression. Finally, there is extensive work on algebraic rewriting and functional reduction using AIG-based representations [5], [12]. Techniques such as cut rewriting, FRAIGs, and structural hashing reduce circuit size by optimizing Boolean structure. These approaches are complementary to ours: while AIG rewriting simplifies logic after lowering to bit-level form, our vectorizer restores and encodes word-level behavior early in the compilation flow, enabling more scalable reasoning by downstream verification and synthesis tools.

VI. Conclusion

This paper introduced a compiler transformation that performs vectorization of Verilog designs. Although vectorization does not alter the synthesized hardware, it reduces the symbolic complexity of the underlying specification. Our experimental results demonstrate that this reduction improves both formal verification performance and synthesis optimization, as observed using the Jasper Formal Verification Platform® and the Genus Synthesis Solution®. Moreover, because the technique operates as a source-to-source transformation, it also enhances the readability and maintainability of Verilog designs, providing practical benefits beyond tool performance.

References

[1] R. Allen and S. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In *PLDI*, PLDI '88, page 241–249, New York, NY, USA, 1988. Association for Computing Machinery.

[2] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. Runtime pointer disambiguation. In *OOPSLA*, OOPSLA 2015, page 589–606, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] Alberto Griggio. Effective word-level interpolation for software verification. In *FMCAD*, page 28–36, Austin, Texas, 2011. FMCAD Inc.

[5] Daniela Kaufmann, Armin Biere, and Manuel Kauers. Verifying large multipliers by combining sat and computer algebra. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 28–36, 2019.

[6] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, page 75, USA, 2004. IEEE Computer Society.

[7] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: scaling compiler infrastructure for domain specific computation. In *CGO*, page 2–14. IEEE Press, 2021.

[8] Wenchao Li, Adria Gascon, Pramod Subramanyan, Wei Yang Tan, Ashish Tiwari, Sharad Malik, Natarajan Shankar, and Sanjit A. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *HOST*, pages 67–74, 2013.

[9] Ram Venkat Narayanan, Aparajithan Nathamuni Venkatesan, Kishore Pula, Sundarakumar Muthukumaran, and Ranga Vemuri. Reverse engineering word-level models from look-up table netlists, 2023.

[10] Rafael Sumitani, João Victor Amorim, Augusto Mafra, Mirlaine Crepalde, and Fernando Magno Quintão Pereira. Chibench: a benchmark suite for testing electronic design automation tools, 2024. https://arxiv.org/abs/2406.06550.

[11] Wenxi Wang, Harald Søndergaard, and Peter J. Stuckey. Wombit: A portfolio bit-vector solver using word-level propagation. *J. Autom. Reason.*, 63(3):723–762, October 2019.

[12] Cunxi Yu, Maciej Ciesielski, and Alan Mishchenko. Fast algebraic rewriting based on and-inverter graphs. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 37(9):1907–1911, September 2018.