Word-Level Counterexample Reduction Methods for Hardware Verification

Zhiyuan Yan and Hongce Zhang[†]
Microelectronics Thrust, The Hong Kong University of Science and Technology (Guangzhou)
[†]Corresponding author: hongcezh@ust.hk

***Abstract*—Hardware verification is crucial to ensure the correctness in the logic design of digital circuits. The purpose of verification is to either find bugs or show their absence. Prior works mostly focus on the bug-finding process and have proposed a range of verification algorithms and techniques to be faster to reach a bug or conclude with a proof of correctness. However, for a human verification engineer, it also matters how to better analyze the counterexamples trace to understand the root cause of bugs. This kind of technique remains absent in word-level circuit analysis. In this paper, we investigate the counterexample reduction method. Given the existing techniques for the bit-level circuit model, we first extend current semantic analysis methods to the word-level counterexample reduction and then develop a more efficient word-level structural analysis approach. We compare the effectiveness and overhead of these methods on the hardware model-checking problems and show the usefulness of such analysis in applications including pivot input analysis, word-level model-checking and counterexample-guided abstraction refinement.**

***Index Terms*—Hardware Model Checking, Counterexample Reduction, Counterexample Reduction Applications**

I. Introduction

Model checking is crucial in the formal verification of digital circuits. It ensures that a state transition system M complies with its expected property P [1], [2]. Upon property failures, the model checking algorithm will generate a counterexample trace spanning from the initial state to the property-violating state, containing assignments to all state and input variables in every transition (clock cycle) [3]. Presently, the SAT-based model checking is one common bug-finding method, which has been demonstrated to be highly effective in hardware verification by the Hardware Model Checking Competition (HWMCC) [4]. Berkeley-ABC [5] is one of the SAT-based model checkers that operate at the bit level. It considers M described by state bits and Boolean operators. The input format for Berkeley-ABC is the AIG format, which describes the circuits by only AND gates and inverters [6].

For a long counterexample trace, it could be difficult for a human engineer to understand the root cause of a bug from the sea of variable assignments. This has led to the development of techniques for counterexample reduction and generalization, which simplifies the trace to include only relevant variable assignments. For example, in a circuit with an AND gate: $c = a \wedge b$, if $b = 0$, the output is determined regardless of a's value. Thus, in a counterexample where $c = 0$, $a = 1$, and $b = 0$, the assignment to a can be safely removed. In practice, Berkeley-ABC [5] is equipped with backward analysis to reduce and generalize counterexamples [7]. Similarly, the IC3/PDR algorithm utilizes these techniques to enhance model-checking efficiency [8]. These strategies not only aid engineers in comprehending the bug in the hardware but also boost the overall efficiency of various formal verification applications.

However, as systems become larger and more complex, it becomes beneficial to raise the level of reasoning to a higher level. In response, there is the emergence of word-level model checkers, like Pono [9] and AVR [10]. These model checkers employ solvers for satisfiability modulo theories (SMT), which extend SAT solvers with capabilities of reasoning about arithmetics, and other first-order theories [11]–[14]. Furthermore, these checkers could leverage semantic information at the word level to enhance lemma learning, enabling word-level reasoning to outperform in various scenarios [4], [15].

Despite these advancements of word-level reasoning, there is a lack of accurate and efficient word-level counterexample reduction method. Existing bit-level counterexample reduction methods are not applicable at the word-level, because traditional bit-level counterexample reduction works on the single-bit logic operations (primarily AND operators for AIG as there is no reduction opportunities for inverters). Conversely, SMT contains a much broader spectrum of operators, including logical, bit-wise, and arithmetic functions. There lacks a set of rules to perform word-level counterexample reduction. Transforming word-level models to bit-level to apply existing techniques inevitably results in information loss. Consequently, there is a need to develop a "native" counterexample reduction method for word-level hardware models.

The paper primarily investigates the following three research questions:

- How to develop effective counterexample reduction and generalization methods for word-level model checking?
- How do these word-level techniques compare in performance to their bit-level counterparts?
- Do word-level counterexample reduction and generalization techniques genuinely enhance the efficacy of various formal verification applications?

To answer the above questions, our paper introduces two novel techniques for counterexample reduction and generalization at the word level: unsatisfiable core (UNSAT core) analysis and dynamic cone-of-influence (D-COI) analysis. For UNSAT core analysis, we extend it to word-level counterexample reduction by transforming the original formula into a UNSAT SMT formula and using UNSAT core to simplify the counterexample trace, while D-COI analysis defines a set of rules for SMT operators and employs backward tracing to identify property violation root causes. Our comparative study shows that these

This work is supported by the National Natural Science Foundation of China (grant no. 62304194) and Guangzhou-HKUST(GZ) Joint Funding Program (grant no. 2023A03J0013).

word-level techniques slightly outperform their bit-level counterparts, eliminating the need for model conversion to bit-level for counterexample processing. Finally, we showcase the utility of these techniques in several applications: identifying pivotal inputs in counterexamples, enhancing word-level IC3/PDR performance, and aiding counterexample-guided abstraction refinement (CEGAR) by generating more generalized counterexamples. These reduction techniques address the existing gap in word-level counterexample analysis and improve the efficacy of word-level model checking in complex digital systems.

Overall, the main contributions in this paper are:

- We extend UNSAT core analysis and develop a new algorithm for word-level counterexample reduction and generalization in model checking. To our best knowledge, it is the first work of applying these techniques in word-level hardware verification.
- We conduct a comparative study on both the word-level and bit-level counterexample reduction techniques. The result shows that our word-level reduction techniques achieve competitive efficiency and a slightly better reduction rate, compared to the bit-level counterpart.
- We demonstrate the usefulness of the word-level counterexample reduction method in three applications. These applications not only help human engineers better pinpoint problems in the digital circuit, but also speed up the whole word-level verification process.

II. Problem Formulation

A digital circuit can be modeled using a finite state transition system, defined as follows:

Definition 1 (**Finite State Transition System**). *A state transition system M is represented by the tuple $\langle \vec{x}, Init(\vec{x}), Tr(\vec{x}, \vec{x}')\rangle$, which consists of state variables $\vec{x}$, the predicate $Init(\vec{x})$ that specifies the initial states, and $Tr(\vec{x}, \vec{x}')$ defining the transition relation, where the next state variables are denoted as $\vec{x}'$.*

As primary inputs of a circuit can be treated as free state variables that are not bound by the transition relation, we omit them in this formulation. The transition relation Tr is conceptualized as a word-level netlist or a directed acyclic graph where nodes correspond to logic gates. As the transition relation of circuits is functional, the next state is uniquely determined given the present state. State s_2 is considered a successor of state s_1 if and only if $(s_1, s_2) \in Tr$.

Given a safety property P, which is an SMT formula over state variables, a model checker's task is to verify if P holds for all states reachable from the initial state defined by $Init(\vec{x})$. If P is satisfied in all reachable states, the system is deemed safe. Otherwise, the system is considered unsafe, and a counterexample trace is produced.

Definition 2 (**Trace**). *A trace is a sequence of states of length k: $s_1, s_2, ..., s_k$, where each pair (s_i, s_{i+1}) satisfying the transition relation, namely $(s_i, s_{i+1}) \in Tr$ $(1 \le i \le k-1)$.*

Definition 3 (**Counterexample Trace**). *A counterexample trace CT is a trace starting from the initial state where the final state s_k does not satisfy the given property, denoted as $s_k \models \neg P$ and $s_1 \models Init$. This trace serves as a proof that P does not hold on M.*

Counterexample reduction and generalization is formally defined below:

Definition 4 (**Reduced Counterexample Trace**). *A reduced counterexample trace $RCT : S_1^{(m)}, S_2^{(m)}, ..., S_k^{(m)}$ of a counterexample trace $CT : s_1, s_2, ..., s_k$ is a sequence of sets of states with the same length k, where each state in CT is contained in the corresponding state set in RCT, namely $s_i \in S_i^{(m)}$ and the set of states in RCT also complies with the requirement of traces: all ending states violate the property: $\forall s_k^{(m)} \in S_k^{(m)}, s_k^{(m)} \models \neg P$, and the state sets are connected by transition relation: $\forall s_i^{(m)} \in S_i^{(m)}, \exists s_{i+1}^{(m)} \in S_{i+1}^{(m)}, \left(s_i^{(m)}, s_{i+1}^{(m)}\right) \models Tr$.*

Upon the first look, Definition 4 defines the generalization of a counterexample trace, because it generalizes a state in a trace to a set of states. But in fact, this is also a reduction of variable assignments in CT. For a state variable assignment $x_1 = v_1 \wedge x_2 = v_2 \wedge x_3 = v_3 \wedge ...$ that represents a single state, once we drop some variable assignments, it effectively becomes a description of a set of states. The goal of this work is to find the reduced (and also generalized) trace given a concrete one.

III. Methods for Word-Level Counterexample Reduction

This section introduces two different techniques for word-level counterexample trace reduction. The first one is based on the UNSAT core reduction, while the second one is the D-COI.

A. UNSAT Core Reduction

Given a state transition system M and a property P to check, for a CT across k states: $s_1, s_2, ..., s_k$, we use $F_1, F_2, ..., F_k$ to represent the formulas where state variables are constrained to the assigned values in the corresponding states, namely, $F_k \stackrel{\text{def}}{=} (\bigwedge_{i=1}^{n} x_i^k = v_i)$, where v_i is the assignment for the variable x_i in the k-th cycle. Now consider the following SMT formula:

$$\begin{aligned} Init(\vec{x}^1) \wedge Tr(\vec{x}^1, \vec{x}^2) \wedge F_1 \wedge Tr(\vec{x}^2, \vec{x}^3) \\ \wedge F_2 ... \wedge Tr(\vec{x}^k, \vec{x}^{k+1}) \wedge F_k \wedge P \end{aligned} \tag{1}$$

Theorem 1. *Formula (1) is unsatisfiable.*

Proof. Each F formula represents the assignments for all input and state variables in that cycle. Conjunction with these assignments effectively fixes the value of each variable in every cycle. As circuit models are functional, this conjunction inevitably results in the violation of property P at cycle k. Since P and $\neg P$ cannot hold at the same time, Formula (1) is unsatisfiable. □

Starting from this UNSAT formula, we may try to remove some variable assignments in the F formulas and see if the remaining formula is still unsatisfiable. This essentially is an UNSAT core reduction (minimization) problem [16].

However, one limitation of the UNSAT core minimization is its potential inefficiency, as shown in Section V-A4. To address this issue, we develop an alternative counterexample reduction method called dynamic cone-of-influence analysis.

B. Dynamic Cone-of-Influence

As the name suggests, dynamic cone-of-influence analysis (D-COI) looks in the cone-of-influence (COI) and tries to remove the signals that are out of COI and therefore, do not contribute to the violation of given properties. The analysis relies on not only the circuit structure but also the variable assignments in the CT generated by the solver. Thus, it is dynamic (namely, dependent on the model returned by the solver). It traverses backward from the property expression to the input and state variables in prior cycles.

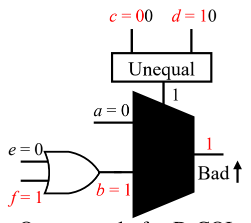

Fig. 1. One example for D-COI analysis.

Fig. 1 illustrates the idea of D-COI with an example circuit. The circuit contains a 2:1 multiplexer controlled by a comparator that outputs 1 when its two inputs c and d are not equal. In a CT generated from the model checker, all signals are already assigned by the solver, which are labeled in the figure. Suppose the property specifies that the output of the multiplexer should always be 0, while the model checker finds a trace that leads to output 1. D-COI analysis will identify the subset of signal assignments that sufficiently trigger this property violation. In this example, the output 1 of the multiplexer is generated by its input signal b as the controlling bit is 1. Therefore, signal a is regarded as irrelevant. The analysis will further backtrace from the controlling bit and signal b. As signal f is assigned the controlling value (1) of the OR-gate, the assignment of b can be solely attributed to signal f. Therefore, e is removed. Meanwhile, for the controlling bit, the inequality stems from the difference in the most significant bits of c and d. The algorithm then traces back from their most significant bits.

As circuits are represented by expressions in SMT [17] inside word-level model checkers, we specify backtracing rules for SMT operators to facilitate accurate D-COI tracking. Table I lists the backtracing rules. The notation "$t \triangleright [h, l]$" means bits from index h to index l of an SMT term t is within COI. Boolean variables are treated as one-bit-wide bit-vectors and use the range $[0, 0]$ if a Boolean variable is in COI. $x[i]$ refers to the i-th bit of term x. We denote $\mathsf{Model}(t)$ as the assignment of t in the CT and $W(t)$ indicates the bit width of term t. Below we further explain the rules for each type of SMT operator. For those without a specific backtracing rule, we will take the most conservative approach and backtrace to all subformulas.

Logic operators: for Boolean operators, D-COI detects if any input is equal to the controlling value. For example, if one input of $\texttt{And}(x, y)$ is 0, the output is unequivocal 0, eliminating the need to evaluate the other input. Conversely, when both inputs are 1, neither x nor y can be removed.

Bit-wise operators: for bit-wise operators over bit-vectors, D-COI scans from bit l to bit h to identify relevant bit-fields in COI following the same rule for each bit. For example, for the $r = \texttt{BVAnd}(x, y)$ operation, assume that in a specific CT, we have $\mathsf{Model}(r) = 00$, $\mathsf{Model}(x) = 00$, and $\mathsf{Model}(y) = 10$. D-COI scans each bit following the rule of a Boolean `And`. Because both $x[0]$ and $y[0]$ are taking the controlling value 0, we may retain only one assignment in COI. Then, D-COI checks the most significant bit, where $x[1] = 0$ and $y[1] = 1$. In this situation, $y[1]$ can be removed from COI.

Arithmetic operators: among the five arithmetic operators in the bit-vector theory, addition and multiplication allow finer-grain COI tracking. For example, the k-th bit of an addition is determined only by addend bits with index k or lower. Therefore, if bits in range $[h, l]$ are within COI, we only backtrace to range $[h, 0]$. The rule for `BVMul` is similar, with consideration of the controlling value 0.

Relational operators: when considering comparison operators, such as `BVUlt`, `BVUle`, `BVUgt`, `BVUge`, D-COI identifies the leftmost bit that makes a difference and track. For instance, for comparing x and y with the assignments $x = 0110$ and $y = 0000$, the values of $x[3..2]$ and $y[3..2]$ determine the relation of x and y. All other bits can be discarded from COI. As for operators like `BVComp` and `Equal`, and `Distinct`, it is adequate to identify and retain just a single differing bit between the two inputs. Conversely, if there is no such single differing bit, D-COI will keep all bits in COI.

Concatenation and bit-field operators: regarding the `Concat` operator and extension operators like `Zero_Extend`, D-COI will check the range $[h, l]$ and see how to proceed to the subformulas. Depending on the bit-width of subformulas and the range to track, D-COI may track both subformulas of $\texttt{Concat}(x, y)$ or only one and it will also map the range $[h, l]$ to the range on x or y. For extension, if only the extended bits are in COI, the subformula of `Zero_Extend` can be completely removed from COI because the extended bits are independent of the subformula, whereas for `Sign_Extend`, we still need to track the most significant bit as it is regarded as the sign bit in the extension. For the `Extract` operator, D-COI will map the tracking range to its subformula based on the extraction indices.

The ternary operator: the `Ite` operator takes the form "*cond ? te : fe*" which corresponds to a multiplexer in digital circuits — if the condition *cond* is true (false), then the expression *te* (*fe*) is the output. D-COI examines the assigned value of the condition, and if the condition is true, the fe becomes irrelevant. Conversely, if the condition is false, te is removed from COI.

Algorithm 1 outlines the process of backtracking in a CT that spans multiple clock cycles (transitions). The inputs to this algorithm include the transition relation Tr, the complete CT cex, and the property P. First, we create a set to gather COI variables and extract the trace length (Line 2-3). The set $vars\_in\_COI$ records the signals (variables) to recursively apply D-COI analysis in the prior cycle. Initially, this set is computed on the last cycle from the property expression (Line 4). The algorithm proceeds to traverse each variable in $vars\_in\_COI$ in the prior cycle (Line 8). If the variable is an input to the circuit, there is no need to further backtrack on this

TABLE I
THE RULES FOR D-COI BACKTRACE.

Type	Operator	Reduction Rule
Logic	And(x, y) $\triangleright$ $[0,0]$	$x \triangleright [0,0]$ **if** $\mathsf{Model}(x) = 0$, $y \triangleright [0,0]$ **if** $\mathsf{Model}(y) = 0$ $x \triangleright [0,0] \land y \triangleright [0,0]$ **Otherwise**
	Or(x, y) $\triangleright$ $[0,0]$	$x \triangleright [0,0]$ **if** $\mathsf{Model}(x) = 1$, $y \triangleright [0,0]$ **if** $Model(y) = 1$ $x \triangleright [0,0] \land y \triangleright [0,0]$ **Otherwise**
	Implies ($Ante$, $Conseq$) $\triangleright$ $[0,0]$	$Ante \triangleright [0,0]$ **if** $\mathsf{Model}(Ante) = 0$, $Conseq \triangleright [0,0]$ **if** $\mathsf{Model}(Conseq) = 1$ $Ante \triangleright [0,0] \land Conseq \triangleright [0,0]$ **Otherwise**
Bit-wise	BVAnd(x, y) $\triangleright[h,l]$, BVNand(x, y) $\triangleright[h,l]$	$x \ \triangleright [i,i], \forall i \in [h,l] \mathsf{Model}(x[i]) = 0 \land y \triangleright [j,j], \forall j \in [h,l]\ \mathsf{Model}(y[j]) = 0 \land$ $x \ \triangleright [k,k] \land y \triangleright [k,k]$, **for the rest** $k \in [h,l]$
	BVOr(x, y) $\triangleright[h,l]$, BVNor(x, y) $\triangleright[h,l]$	$x \triangleright [i,i], \forall i \in [h,l]\ \mathsf{Model}(x[i]) = 1 \land y \triangleright [j,j], \forall j \in [h,l]\ \mathsf{Model}(y[j]) = 1 \land$ $x \triangleright [k,k] \land y \triangleright [k,k]$, **for the rest** $k \in [h,l]$
Arithmetic	BVAdd(x, y) $\triangleright[h,l]$	$x \triangleright [h,0] \land y \triangleright [h,0]$
	BVMul(x, y) $\triangleright[h,l]$	$x \triangleright [W(x)-1,0]$ **if** $\mathsf{Model}(x) = 00...0$, $y \triangleright [W(y)-1,0]$ **if** $\mathsf{Model}(y) = 00..0$ $x \triangleright [W(x)-1,0] \land y \triangleright [W(y)-1,0]$ **Otherwise**
Relational	BVUlt(x, y) $\triangleright[0,0]$, BVUle(x, y) $\triangleright[0,0]$ BVUgt(x, y) $\triangleright[0,0]$, BVUge(x, y) $\triangleright[0,0]$	$x \triangleright [W(x)-1,i] \land y \triangleright [W(y)-1,i]$ **if** $\exists i\ \forall j > i\ \mathsf{Model}(x[j]) = \mathsf{Model}(y[j]) \land \mathsf{Model}(x[i]) \neq \mathsf{Model}(y[i])$ $x \triangleright [W(x)-1,0] \land y \triangleright [W(y)-1,0]$ **Otherwise**
	Equal(x, y) $\triangleright[0,0]$, BVComp(x, y) $\triangleright[0,0]$ Distinct(x, y) $\triangleright[0,0]$	$x \triangleright [i,i] \land y \triangleright [i,i]$ **if** $\exists i\, \mathsf{Model}(x[i]) \neq \mathsf{Model}(y[i])$ $x \triangleright [W(x)-1,0] \land y \triangleright [W(y)-1,0]$ **Otherwise**
Concatentation	Concat(x, y) $\triangleright[h,l]$	$x \triangleright [h-W(y), l-W(y)]$ **if** $l \geq W(y)$, $y \triangleright [h,l]$ **if** $l < W(y)$ $x \triangleright [h-W(y),0] \land y \triangleright [W(y)-1,l]$ **Otherwise**
Bit-field operations	Zero_Extend(x, c) $\triangleright[h,l]$ Sign_Extend(x, c) $\triangleright[h,l]$	$x \triangleright [W(x)-1,l]$ **if** $l < W(x) \land h \geq W(x)$, $x \triangleright [h,l]$ **if** $l < W(x) \land h < W(x)$ $x \triangleright [W(x)-1, W(x)-1]$ **Otherwise, for** Sign_Extend **x is irrelevant. Otherwise, for** Zero_Extend
	Extract(x, $[c_1, c_2]$) $\triangleright[h,l]$	$x \triangleright [c_2+h, c_2+l]$
Ternary	Ite($cond$, te,fe) $\triangleright[h,l]$	$cond \triangleright [0,0] \land te \triangleright [h,l]$ **if** $\mathsf{Model}(cond) = 1$, $cond \triangleright [0,0] \land fe \triangleright [h,l]$ **if** $\mathsf{Model}(cond) = 0$
Others		backtrace all sub-formulas

Algorithm 1 D-COI on the counterexample trace

```
 1: procedure D_COI($Tr, cex, P$)
    Input: $Tr$: finite state transition system, $cex$: counterexample trace, $P$: property;
    Output: $COI$: the state variables and input variables in the Cone of Influence ;
 2:     $COI \leftarrow \{\}$        ▷ $COI$ is a set of tuples $(v,t)$, where $v$ is a variable and $t$ is the index in the trace
 3:     $k \leftarrow len(cex)$
 4:     $vars\_in\_COI \leftarrow$ get_COI($\neg P, k, cex$)        ▷ backtrack from $\neg P$ in the last cycle according to the rule in Table I
 5:     $k \leftarrow k - 1$
 6:     while $k > 0$ do
 7:         $vars\_to\_backtrack \leftarrow \{\}$        ▷ track variables to look at in the prior cycle
 8:         for each $var$ in $vars\_in\_COI$ do
 9:             $COI.insert(var, k)$
10:             if $var$ is bound by transition relation then
11:                 $update\_function \leftarrow$ get_update($Tr, var$)
12:                 $upd\_COI \leftarrow$ get_COI($update\_function, k, cex$)
13:                 $vars\_to\_backtrack.insert(upd\_COI)$
14:         $vars\_in\_COI \leftarrow vars\_to\_backtrack$
15:         $k \leftarrow k - 1$        ▷ go to the prior cycle
16:     for each $var$ in $vars\_in\_COI$ do
17:         $COI.insert(var, 0)$
18:     return $COI$
```

variable because it is not bound by transition relation and is irrelevant to the prior states. It will only be added to the COI. Otherwise, for state variables, their assignments are caused by the prior state and we will need to backtrack to the prior cycle following its state update function (Line 11-13). After processing all variables in the current cycle, the algorithm collects variables to track in the prior cycle ($vars\_to\_backtrack$) and proceeds to the next iteration (Lines 14-15). Upon completion, the algorithm retains assignments for initial state variables and input variables within the COI in the CT.

D-COI analysis is different from UNSAT core reduction in that the UNSAT core is in essence a semantic analysis which involves SMT solving, while the D-COI is mainly a syntactic analysis — it uses the abstract syntax tree of the state transition relation and the assignments in the CT.

```
always @(posedge clk) begin
  if (internal != 6)
    internal <= internal + 1;
  else if (in)
    internal <= internal + 1;
end
assert(internal < 10);
```

Fig. 2. One example to illustrate pivot input. Register `internal` is initialized to 0. For a counterexample of 10 cycles, only the input value at cycle 6 matters.

IV. APPLICATIONS OF COUNTEREXAMPLE REDUCTION

A. Pivot Input Analysis

Logic circuits are deterministic and functional, meaning that from a fixed starting state, future states are solely determined by circuit inputs. There could be certain circuit inputs at a specific time point causing the property violation. These are termed *pivot inputs*. Identifying pivot inputs aids engineers in understanding the root causes of bugs.

Fig. 2 shows such an example written in Verilog-HDL. The code snippet describes a 0-initialized counter `internal` that waits for an input signal `in` to continue counting when the count reaches 6 and there is an assertion that states the counter will not reach 10 or above. It can be seen that the value of in is critical to trigger the assertion violation, but only when the counter is at a specific value. As mentioned earlier, we call the input values that effectively steer the execution towards property violation as the *pivot input*. When using the Bounded Model Checking (BMC) engine to find assertion violations, it will return the shortest trace (in this specific example, a trace with a length of 10 clock cycles) and the `in` signal at the 6^{th} cycle is the pivot input for the counterexample trace.

Analyzing the pivot input is one application for our counterexample reduction techniques, both the D-COI and UNSAT-core-based methods can be leveraged to detect the pivot input for a circuit.

B. Predecessor Generalization in Word-Level PDR Algorithms

Counterexample reduction has applications beyond facilitating human debugging. It is also valuable in algorithms involving counterexamples. For example, the IC3/PDR algorithm uses counterexamples-to-induction (CTIs) to refine the over-approximation of reachable states [5], [8]. While bit-level IC3/PDR typically uses ternary simulation for counterexample reduction [8], word-level implementations like `Pono` and AVR lack an equally effective word-level equivalent as it is not feasible to directly apply bit-level reduction on word-level models. Take `Pono` as an example, the existing counterexample generalization procedure is inaccurate as it does not differentiate individual bits in a word — even if only a single bit in a word contributes to the violation of the property, it will keep the whole word in the counterexample. On the other hand, our counterexample reduction and generalization analysis is more precise as it discerns specific bit-relevance in each word. Therefore, integrating our method should improve word-level model-checking algorithms, as show in Section V-B.

C. Counterexample-Guided Abstraction Refinement (CEGAR)

Another application of counterexample reduction is in the Counterexample-Guided Abstraction Refinement (CEGAR) [18] process. Abstraction in general could help scale up verification. However, over-approximation could incur spurious counterexamples. The CEGAR method iteratively refines the abstract model by eliminating these spurious counterexamples. By employing a better counterexample reduction approach, the spurious counterexample to eliminate can be more general, thereby improving the refinement process and decreasing the number of iterations needed for refinement.

One of the CEGAR processes in hardware verification is the synthesis of the symbolic starting state constraint. Verification from symbolic starting state is a common hardware verification technique in various applications [19]–[21] as it achieves better scalability for handling practical hardware designs. While, the symbolic starting state usually could not be just an arbitrary state as not every state is possibly reachable. Constraints for the symbolic state are often needed. Prior work [22] proposed to use a CEGAR approach to synthesize the symbolic starting state constraints, where the initial abstraction is the whole state space, which is then iteratively refined through blocking the non-reachable state that leads to the violation of specified properties. The counterexample generalization is helpful as it allows the later blocking step to remove more spurious counterexamples all at the same time, and therefore, reduces the number of CEGAR iterations.

V. Experimental Evaluation

In this section, we evaluate our word-level counterexample reduction methods, and the effectiveness of integrating our methods into model-checking applications. Our algorithm was implemented in C++, based on `Pono` and SMT-switch. All experiments were conducted on a server running Ubuntu 20.04 with a 2.9 GHz Intel Xeon Platinum 8375C CPU and 256 GB of RAM.

A. Pivot Input Analysis

1) Methods in Comparison: In this experiment, we use Berkeley-ABC's bit-level counterexample analysis as our baseline, which offers three reduction methods via the `write_cex` command with options `-o`, `-e`, `-u`. The `-o` option implements a method akin to D-COI but is at the bit-level, while `-e` and `-u` use SAT-based techniques. The `-e` takes more effort (i.e., more SAT queries) to try to obtain a more accurate result. These methods were documented by prior research [5], [7].

At the word level, we employ three methods: D-COI, UNSAT-core-based reduction, and an integrated approach combining initial D-COI reduction followed by UNSAT-core reduction. This experiment aims to compare the accuracy (i.e., reduction rate) and efficiency of bit-level and word-level reduction methods.

2) Test Cases: We use the unsafe instances in the 2020 HWMCC bit-vector track [4] to compare different methods. HWMCC contains model checking problems from both academia and industry. Among the 325 cases in HWMCC2020, only 20 of them lead to counterexample traces in all comparing tools. We then apply pivot input analysis to the traces.

3) Metric: In pivot input analysis, we only consider the reduction on the input variables. As a consequence, for a given unsafe test case, the reduction rate can be calculated as follows:

$$r_{pivot} = 1 - \frac{remaining\_assignment\_for\_input}{num\_inputvar \times trace\_length} \quad (2)$$

where $remaining\_assignment\_for\_input$ denotes the number of input variable assignments left after the application of the reduction technique, and $num\_inputvar$ denotes the number of input variables. $trace\_length$ represents the number of clock cycles (i.e., the bound) needed to reach a bad state in the trace.

4) Experimental Result: Table II presents the reduction rates and execution times for unsafe cases. Higher reduction rates indicate fewer remaining input assignments and better algorithm accuracy. The techniques in comparison include D-COI, UNSAT core, their integration (D-COI + UNSAT core), and three bit-level reduction techniques from Berkeley-ABC (`-o`, `-e`, `-u`). The UNSAT-core-based methods generally achieve higher accuracy, both in isolation and when integrated with D-COI. Comparing D-COI with its bit-level counterpart ABC_O, D-COI shows superior accuracy in 11 instances, equal performance in 4, and lower accuracy in 5. Overall, word-level techniques demonstrate comparable (in fact marginally better in most cases) accuracy to their bit-level counterparts.

The efficiency (i.e., execution time) comparison reveals that firstly, D-COI and ABC_O are the fastest in identifying pivotal inputs, with D-COI outperforming in 15 instances and ABC_O in 3. Secondly, D-COI slightly outperforms ABC_O, despite the marginal difference. This is likely due to word-level

TABLE II

REDUCTION RATE AND EXECUTION TIME (SECONDS) FOR DIFFERENT PIVOT INPUT EXPLORATION TECHNIQUES

	Reduction						Execution Time					
	D-COI	**UNSAT core**	**D-COI + UNSAT core**	**ABC_O**	**ABC_E**	**ABC_U**	**D-COI**	**UNSAT core**	**D-COI + UNSAT core**	**ABC_O**	**ABC_E**	**ABC_U**
shift_register_top_w16_d8_e0	89.63%	**94.89%**	**94.89%**	91.02%	91.80%	90.71%	**0.02**	0.13	0.12	0.06	0.20	0.13
arbitrated_top_n2_w8_d16_e0	88.58%	**95.12%**	**95.12%**	87.42%	90.89%	87.42%	**0.06**	0.24	0.27	0.08	0.34	0.16
circular_pointer_top_w8_d16_e0	82.05%	**91.36%**	**91.36%**	78.86%	82.27%	82.05%	**0.03**	0.15	0.16	0.08	0.23	0.13
circular_pointer_top_w32_d16_e0	94.36%	**97.29%**	**97.29%**	94.43%	94.43%	94.36%	**0.03**	0.41	0.33	0.15	0.55	0.26
shift_register_top_w64_d8_e0	97.10%	**98.55%**	**98.55%**	97.10%	97.67%	97.37%	**0.02**	0.46	0.32	0.16	0.68	0.38
arbitrated_top_n4_w16_d16_e0	95.11%	**98.53%**	**98.53%**	94.62%	96.35%	94.62%	**0.15**	1.08	0.99	0.21	1.31	0.33
circular_pointer_top_w128_d8_e0	98.54%	**99.30%**	**99.30%**	98.25%	98.54%	98.51%	**0.01**	0.41	0.30	0.19	0.64	0.36
arbitrated_top_n5_w64_d16_e0	98.66%	**99.68%**	**99.68%**	98.55%	99.21%	98.57%	**0.22**	5.00	3.00	0.30	2.27	0.53
shift_register_top_w32_d8_e0	94.37%	**97.23%**	**97.23%**	95.04%	95.55%	94.96%	**0.02**	0.22	0.18	0.08	0.32	0.20
arbitrated_top_n3_w32_d16_e0	96.86%	**98.99%**	**98.99%**	96.71%	97.93%	96.76%	**0.09**	1.32	1.10	0.12	1.05	0.24
arbitrated_top_n5_w128_d8_e0	99.29%	**99.83%**	**99.83%**	99.25%	99.57%	99.31%	**0.12**	3.4	1.23	0.26	0.78	0.37
circular_pointer_top_w64_d8_e0	97.08%	**98.63%**	**98.63%**	96.64%	97.14%	97.08%	**0.01**	0.23	0.14	0.11	0.32	0.19
arbitrated_top_n3_w8_d16_e0	89.12%	**96.52%**	**96.52%**	89.30%	92.69%	89.21%	**0.09**	0.35	0.44	**0.09**	0.87	0.19
anderson.3.prop1-back-serstep	**30.06%**	**30.06%**	**30.06%**	**30.06%**	**30.06%**	**30.06%**	**0.01**	0.30	0.25	0.06	0.16	0.13
at.6.prop1-back-serstep	15.14%	**15.19%**	**15.19%**	14.17%	**15.19%**	**15.19%**	0.13	6.00	5.87	**0.07**	0.50	0.11
arbitrated_top_n4_w128_d16_e0	99.33%	**99.80%**	**99.80%**	99.27%	99.51%	99.27%	**0.15**	12.49	3.98	0.47	2.98	0.72
brp2.3.prop1-back-serstep	2.967%	2.967%	2.967%	**2.970%**	**2.970%**	**2.970%**	0.47	302.40	311.30	**0.10**	3.94	0.53
picorv32_mutAY_nomem-p4	97.55%	**97.67%**	**97.67%**	97.48%	97.59%	97.40%	2.46	16.05	16.15	**1.57**	12.34	3.41
vis_arrays_buf_bug	79.19%	90.91%	90.91%	87.08%	**91.15%**	88.52%	0.65	**0.11**	0.89	0.66	0.40	0.70
mul7	**67.53%**	67.41%	**67.53%**	**67.53%**	**67.53%**	**67.53%**	**0.001**	23.38	17.40	0.59	4.65	1.83

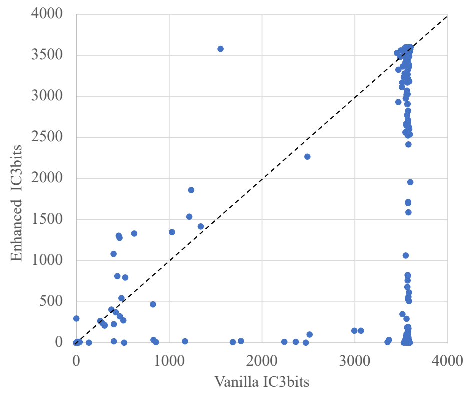

Fig. 3. Comparison of wall-clock time (seconds) between enhanced and vanilla IC3bits engine in Pono.

expressions containing complex operations, which result in deeper AIG circuits when converted to bit-level, thus increasing traversal time. Finally, UNSAT core-based techniques are more time-consuming than both D-COI and ABC_O. When D-COI is used in combination with UNSAT core method, it could save some of the efforts of the later, but still takes longer time than D-COI alone.

Based on these findings, we choose to use D-COI in subsequent experiments, as it balances between efficiency and the reduction rate.

B. *The Effectiveness of Enhancing Word-Level Model Checking*

Furthermore, we test the usefulness of counterexample reduction by conducting an experiment that uses the D-COI analysis to improve the predecessor generalization procedure in word-level model checking. We base our experiment on a model checking engine called IC3bits, which is integrated in `Pono`. The experiment tests its performance on the bit-vector track of the 2020's HWMCC benchmarks before and after integrating D-COI analysis, with a 3600-second time limit. Fig. 3 shows the overall result. With our reduction method, the enhanced IC3bits engine demonstrates a notable improvement over the vanilla engine. It exceeds the vanilla engine over 103 instances with 46 exclusively solved instances.

C. *Effectiveness of Counterexample Reduction in CEGAR*

To demonstrate the efficacy of our counterexample reduction method in a CEGAR approach, we applied it to the task of symbolic starting state constraint synthesis. Note that it is preferable to perform word-level synthesis as word-level constraints are more succinct. The word-level counterexample reduction method saves the overhead of bit-level conversion in each CEGAR iteration. We use hardware designs from the previous work [22]. These designs differ in circuit structure, number of state bits (#. state-bits) and word-level state variables (#. word-state-vars), indicating varying levels of difficulty in solving. We set a time limit of 7200 seconds for the running time.

TABLE III

EXPERIMENTS OF SYMBOLIC STARTING STATE CONSTRAINT SYNTHESIS

Design Statistics			**w. D-COI**		**w.o. D-COI**	
Name	#. state-bits	#. word-state-vars	# iter.	T_{solve}(s)	# iter.	T_{solve}(s)
RC	8	2	3	**2.0**	3	**2.0**
SP	72	16	15	**12.2**	>2892	T.O.
PICO	1817	149	32	**1652.9**	>3	T.O.

The results, presented in Table III, include the number of CEGAR iterations (# iter.) and the total solving time (T_{solve}). Overall, D-COI enhances the generalization of the counterexamples thus allowing the CEGAR loop to terminate earlier, especially for larger designs like SP and PICO. Counterexample reduction not just helps to reduce the number of iterations, it also makes each iteration faster as the later blocking step now has a simpler counterexample. This explains that for PICO, CEGAR without D-COI can only finish 3 iterations, which is far from synthesizing the full constraints.

VI. CONCLUSION

This paper extends the UNSAT core and introduces the D-COI technique for word-level counterexample reduction, which are helpful for human verification engineers in analyzing the root cause of hardware bugs and automated hardware verification. We use three applications to demonstrate the usefulness of word-level counterexample reduction techniques and show that word-level model checking and CEGAR for initial state constraint synthesis exhibit notable improvements when integrated with the counterexample reduction method. We also compare our techniques with the bit-level counterparts and show that our word-level reduction methods achieve better accuracy and efficiency.

References

[1] A. Bernardini, W. Ecker, and U. Schlichtmann, "Where formal verification can help in functional safety analysis," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8.

[2] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–54, 2009.

[3] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, "Verifying safety properties of a powerpc- microprocessor using symbolic model checking without BDDs," in *Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11*. Springer, 1999, pp. 60–71.

[4] M. Preiner, A. Biere, and N. Froleyks, "Hardware model checking competition 2020," 2020.

[5] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.

[6] A. Biere, "The AIGER and-inverter graph (AIG) format version 20071012," 2007.

[7] A. Mishchenko, N. Een, and R. Brayton, "A toolbox for counter-example analysis and optimization," in *Proc. of IWLS*, vol. 13, 2013.

[8] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011, pp. 125–134.

[9] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett, "Pono: a flexible and extensible smt-based model checker," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer, 2021, pp. 461–474.

[10] A. Goel and K. Sakallah, "AVR: abstractly verifying reachability," in *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part I 26*. Springer, 2020, pp. 413–422.

[11] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, no. 1, pp. 53–58, 2014.

[12] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "cvc4," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 2011, pp. 171–177.

[13] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[14] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[15] A. Goel and K. Sakallah, "Empirical evaluation of ic3-based model checking techniques on verilog rtl designs," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 618–621.

[16] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable boolean formula," *SAT*, vol. 3, 2003.

[17] C. Barrett and C. Tinelli, *Satisfiability modulo theories*. Springer, 2018.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 2000, pp. 154–169.

[19] K. Ganesan, F. Lonsing, S. S. Nuthakki, E. Singh, M. R. Fadiheh, W. Kunz, D. Stoffel, C. Barrett, and S. Mitra, "Effective pre-silicon verification of processor cores by breaking the bounds of symbolic quick error detection," *arXiv preprint arXiv:2106.10392*, 2021.

[20] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 55–60.

[21] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification: 6th International Conference, CAV'94 Stanford, California, USA, June 21–23, 1994 Proceedings 6*. Springer, 1994, pp. 68–80.

[22] H. Zhang, W. Yang, G. Fedyukovich, A. Gupta, and S. Malik, "Synthesizing environment invariants for modular hardware verification," in *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings 21*. Springer, 2020, pp. 202–225.