An Efficient Path-oriented Bitvector Encoding Width Computation Algorithm for Bit-precise Verification *

Nannan He and Michael S. Hsiao {nhe, mhsiao}@vt.edu
Department of Electrical and Computer Engineering, Virginia Tech Blacksburg, VA, 24061

Abstract

Bit-precise verification with variables modeled as bitvectors has recently drawn much interest. However, a huge search space usually results after bit-blasting. To accelerate the verification of bit-vector formulae, we propose an efficient algorithm to discover non-uniform encoding widths W_e of variables in the verification model, which may be smaller than their original modeling widths but sufficient to find a counterexample. Different from existing approaches, our algorithm is path-oriented, in that it takes advantage of the controllability and observability values in the structure of the model to guide the computation of the paths, their encoding widths and the effective adjustment of these widths in subsequent steps. For path selection, a subset of single-bit path-controlling variables is set to constant values. This can restrict the search from those paths deemed less favorable or have been checked in previous steps, thus simplifying the problem. Experiments show that our algorithm can significantly speed up the search by focusing first on those promising, easy paths for verifying those path-intensive models, with reduced, non-uniform bitwidth encoding.

1. Introduction

In formal verification, modeling data variables as *bitvectors* with a bounded width has shown some unique benefits. Bounded modeling is capable of accurately capturing the true semantics of the verification instances constrained by a physical word-size on a computer. Furthermore, with the advances in Boolean and bitvector arithmetic reasoning, SAT (or SMT) based formal verification has the potential to deal with large problems. Many existing software model checking tools (e.g., CBMC [3], SATABS [4], Saturn [14], F-SOFT [5]) and hardware design validation techniques (e.g., [9, 11]) have taken bitvector modeling.

However, with bitvectors, the search space can be huge after bit-blasting, especially when dealing with large hardware designs and/or software programs. For example, in a large instance, it is extremely challenging to find a satisfying assignment (counterexample) with a full-size encoding. One way to handle this problem is to reduce the encoded bitvector width of the variables, thereby restricting the searching space. Then, the verification is conducted on the restricted model instead of the original one. Several approaches have been proposed to compute the reduced bitvector width for enhancing the verification scalability.

In [10], an abstraction approach was introduced to scale down the data path for formal RTL property checking. Based on the static data dependency analysis and granularity analysis of bitvector equations, it computes an abstract model where the bitvector width of variables is reduced with respect to the property. To alleviate the state explosion in software model checking, the authors of [15] reduce the bitvector width of variables according to their lower and upper bounds determined by a symbolic value range analysis technique. Both approaches are applied as preprocessing steps that directly decrease the modeling width W of each variable and still preserve the verification property. However, they do not consider the dynamic information during verification.

Recently, a new approach was proposed in an under- and over-approximation based abstraction-refinement framework to iteratively learn the sufficient encoding width W_e of variables for verification, which is smaller than their individual modeling widths [2]. Starting with a small W_e for every free variable, their approach enlarges W_e of some variables in each refinement step by analyzing the abstract counterexample of the over-approximate abstraction. For refutable properties (where a counterexample exists), the refinement process continues until a SAT assignment is found (with a smaller W_e) or when the W_e of all variables have reached their original W. This approach dynamically computes small values for W_e during verification instead of scaling down the width beforehand using static analysis as in [10, 15]. Although it is flexible, its claimed efficiency is limited in scenarios where the SAT assignment can be represented with a smaller encoding width. Moreover, the values assigned in the abstract counterexample may be unnecessarily large derived from a large W_e and thus increases the verification difficulty.

In this paper, we present an efficient path-oriented bitvector encoding width computation algorithm to alleviate the above limitations. Similar to [2], our algorithm embeds the *dynamic* computation of W_e in the abstraction-refinement framework. However, it is distinguished by its path-oriented analysis with the guidance of *static* controllability metric (CM) and observability metric (OM) in three major ways. First, it computes the initial *non-uniform* W_e of variables on

*supported in part by NSF grants 0519959, 0524052, NIJ grant 2005-IJ-CX-K017.

978-3-9810801-5-5/DATE09 © 2009 EDAA

different paths. By setting a bigger initial W_e for the variables on the easy-to-control paths while setting a smaller W_e for the other paths, our approach can greatly increase the chance of finding a SAT assignment in the restricted search space directly without the need to adjust W_e multiple times. Secondly, in the W_e adjustment steps (if necessary), our approach gives priority to enlarging the W_e of the easily-controllable variables first through the manipulating of the abstract counterexample generation guided by CM and OM. This helps to systematically search for the concrete counterexample with a reduced effort. Thirdly, it sets W_e to zero for some single-bit variables that determine the path(s) selection, thereby enforcing constant values on them to restrict choosing only a subset of paths. This can avoid searching those partitions that have been checked in previous steps, especially the ones on which the variables' W_e experienced no increase, thus simplifying the problem.

Outline The remainder of the paper is organized as follows. In Section 2, we will give some preliminaries related to our work. Our proposed encoding algorithm is presented in Section 3. We report our experimental results in Section 4 followed by the conclusion in Section 5.

2 Preliminaries

2.1 Bitvector Formula Encoding

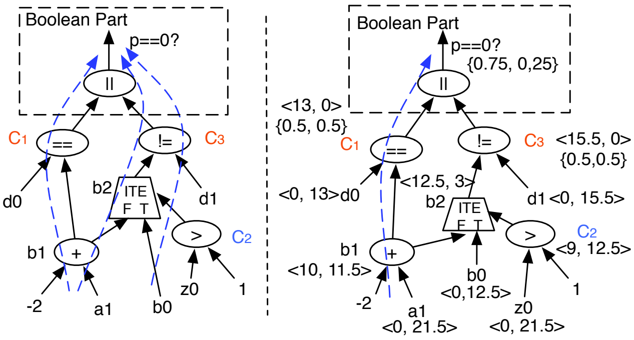
The bitvector arithmetic formula we focus on is a conjunction of terms, where each term is in the format (Identifier == Identifier [op Identifier]). Every Identifier represents a bitvector variable which is interpreted as a numeric value represented in two's complement form. According to whether the operator accepts single-bit inputs and whether the output is a single bit, we group operators into three categories:

- Bitwise operators: $\&$ (and), $|$ (or), $\otimes$ (xor), $\sim$ (not), ite $(if-then-else)$;
- Predicate operators: $==$, $\neq$, $>$, $<$, $\geq$, $\leq$;
- Non-Boolean operators: $+$, $-$, $\times$, $/$, $\%$, shift, type cast (Concatenation, Extraction and Extension), etc;

The resulting formula can be represented as a directed acyclic graph model. An example is shown in Fig. 1, where three possible paths (highlighted by dash lines) are possible to reach p from the inputs.

Definition 1. *Starting from the least significant bit, the encoding width $W_e(v)$ for a bitvector variable v is the number of consecutive bits in the vector whose values have not been assigned, $0 \leq W_e(v) \leq W(v)$. For each of the remaining $W(v) - W_e(v)$ bits, the value is set to be constant 0 (or 1).*

If the W_e of individual variables is smaller than their W, the search space can be restricted. For example, a variable v with $W = 32, W_e = 6$, the original value range is $[-2^{31}, 2^{31}-1]$ and the constrained value range is reduced to $[0, 63]$ by enforcing the 22 most significant bits to 0. When



(a) Graph Representation of (c) (b) Graph Model with CM/OM

F = (b1=a1+(-2)) ∧ (C1=(d0==b1)) ∧ (C2=(z0>1))
∧ (b2=C2 ite b0 b1) ∧ (C3=(d1!=b2))
∧ (p=(C1||C3)) ∧ p

(c) Example Bit-vector Arithmetic Formula

Figure 1. A Formula with its Graph Model

W_e is set equal to 0, the variable simply becomes a constant. We observe that the selector input of an ITE (if-then-else) variable (called $isel$) has a special property. (In Fig. 1, C_2 is an $isel$.) When constant 0 (1) is enforced to an $isel$ (setting its W_e to zero), the false (true) branch is always taken and some variables on the true (false) branch may become dangling variables. Thus, it is safe to slice away these dangling variables as they do not feed other portions of the code.

Definition 2. *A variable is a Boolean Frontier Variable (BFV) if it is the output of a predicate operator and all its fanouts are variables with Bitwise operators.*

In Fig. 1, C_1 and C_3 are BFVs. We consider them as pseudo inputs of the Boolean portion of the formula where every variable has $W = 1$.

2.2 Controllability/Observability Metrics

CM and OM are two generic metrics widely used to evaluate the testability of a hardware design ([8, 13]) or software components [12]. In [7], CM and OM have been used to estimate the amount of influence that each bitvector variable has on the property verification. Specifically, the CM of a variable approximates the *difficulty* of setting a value along the paths reaching a target variable from the inputs. The *difficulty* is defined by two main factors: the lengths of the paths and the computation complexity along these paths. The OM approximates the amount of impact that a value-change on a variable has on the output. It is used to estimate the verification relevance of a variable to the target property of interest.

The CM/OM computation defined in [7] is adopted here. It uses pre-calculated controllability and observability coefficients (COC) of basic bitvector operators to represent the operators' computational efficiency. The COC approximates the different amounts of influence the operators have on the CM and OM. The details of COC values and the formulas of the CM and OM computation are omitted due to the space limit. We introduce two properties of our CM/OM computation relevant to this work:

- Larger values of the CM/OM indicate harder controllable/observable.
- On any path from primary inputs (PIs) to a primary output (PO), $CM(v_1) < CM(v_2)$ and $OM(v_1) > OM(v_2)$ always hold if v_1 is the predecessor of v_2.

However, in [7], the 1- and 0-state CM of the variables in the Boolean portion of the instance were not differentiated. Here, we first estimate the 0-state CM^0 and 1-state CM^1 of every BFV var according to four cases below:

1. $op(var) \in \{\geq, \leq, <, >\} : CM^0 = CM^1 = 0.5$.
2. $op(var) \in \{==, \neq\}$ and at least one argument of var is fully controllable like PI: $CM^0 = CM^1 = 0.5$.
3. $op(var) \in \{==\}$ and no arguments of var is fully controllable: $CM^0 = 1 - 2^{-W}, CM^1 = 2^{-W}$.
4. $op(var) \in \{\neq\}$ and no arguments of var is fully controllable: $CM^0 = 2^{-W}, CM^1 = 1 - 2^{-W}$.

We propagate the CM^0 and CM^1 of BFVs to all other variables in the Boolean portion according to the 1 and 0 probability measure of the corresponding bitwise operator. In Fig. 1(b), it shows CM and OM labeled as a $< CM, OM >$ pair next to each variable outside or at the boundary of the Boolean portion. It also gives the CM^1 and CM^0 next to each variable in the Boolean portion (enclosed by the brace). For example, CM^0 and CM^1 of variable C_1 are both 0.5 based on Case(2) since $d0$ is a PI. To verify the property $p == 0$, C_1 and C_3 have the same CM^1. But since CM of C_1 is smaller than that of C_3 which means C_1 is more easily controllable, the path following the dash line is considered as the easy-to-control path.

3 Our Proposed Algorithm

3.1 Overview of the Steps

To enhance the scalability of bit-precise verification, we propose an efficient path-oriented algorithm to compute a small but verification-sufficient encoding width W_e of individual variables in the instance. The algorithm exploits not only the dynamic information learned in the abstraction-refinement iterations, but also the high-level static structural information through the guidance of the controllability and observability metrics. We assume a verification instance formulated as a bitvector arithmetic formula whose satisfiability corresponds to the negation of a given property (i.e., SAT means the property is refuted). We also assume many paths exist in the instance, which is common in practical problems. Finally, our current work focuses on refuting properties.

The basic flow of our algorithm is illustrated in Fig. 2. We give an overview of each step in below. Three important steps enclosed with dash line borders will be presented in more details in following subsections.

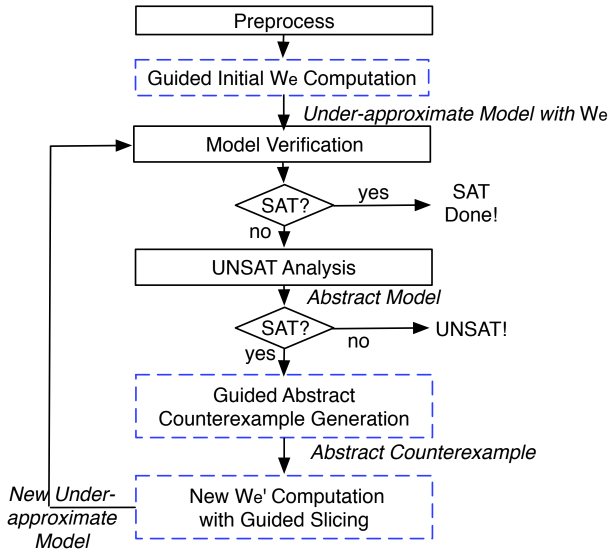


Figure 2. Basic Flow of Our Algorithm

1. *Preprocessing*: Given a bitvector formula, our algorithm first builds its graph model M, then computes *BFVs*, CM, and OM. Next, it selects some *isel*s to build the set IS. These *isel*s stored in IS will be used to conduct the path(s) oriented abstract counterexample generation in Step 5 and slicing in Step 6. We prefer selecting the $isel$s with a larger number of connections to the verification-relevant *ITE*s because they have a greater impact on reducing the partition size. The work proposed in [10, 15] can also be applied here to reduce the modeling width.
2. *Guided initial W_e computation*: Our algorithm computes a small initial W_e and determines the constant value to be placed for the bits outside of the encoding widths. The main idea is to give preference (larger initial W_e) to variables on easy-to-control paths so as to increase the chances of finding a SAT assignment fast. Moreover, it keeps the encoding size of a variable consistent with the input and output variables and a given operator type. The algorithm also considers the effect of constants in the encoding computation. It then applies the computed initial W_e of variables to build an initial under-approximate model M_u.
3. *Model verification*: Through bit-blasting, M_u is transformed to a Boolean model β_u. If β_u is SAT, we can conclude M is also SAT as the values enforced on M_u are achievable in M; otherwise, go to Step 4.
4. *UNSAT analysis*: Our algorithm adopts the method used in [6] to conduct the UNSAT analysis for refining the abstraction M_o. The basic idea is to identify *all* variables involved in the UNSAT proof of M_u and add them into M_o. Note that the constant values used outside the encoding widths and constant values for the $iSel$ variables in M_u are not carried over. Thus, all variables in

M_o have the original modeling width W. The refined M_o can refute all spurious counterexamples where the assignment of the variable is within the encoded value range of its W_e, so that it prevents repeated generation of the same spurious counterexample. A theorem on this can be found in [6]. This method is simpler than the technique in [2] whose proof-based abstraction also considers the special usage of Boolean nodes.

5. *Guided abstract counterexample generation*: Since M_o is small, it is easy to handle it and find an abstract counterexample γ with certain expectations. This is done by enforcing some extra constraints on M_o guided by CM and OM to steer the search. To avoid generating a new M_u that is similar to the previous M_u in which no SAT assignment was found, we prefer generating γ on M_o that can help enlarge those least frequently enlarged W_e. As some variables have values assigned in γ falling beyond the value ranges of their previous W_e values, it is also preferred that they are the easy-to-control ones.
6. *New W_e computation with guided slicing*: Given the abstract counterexample γ, we enlarge the W_e of some variables in the previous M_u so that the new value ranges can adequately cover their values assigned in γ. We update the W_e of all other data dependent variables in M and apply the new W_e on variables to building the new under-approximate model $M_u^{'}$. To avoid repeatedly searching the same space among the iterations, our algorithm enforces certain constant values on some $isel$s and applies model slicing to remove from the new under-approximate model those variables whose W_e were not enlarged and have thus become dangling variables. The process goes back to Step 3 to start a new iteration.

Due to the finite search space, the algorithm always terminates. In the worst case, the W_e of individual variables may need to be enlarged to their original W. However, experiments show that a SAT assignment with small value ranges or on the path with the small number of variables exists in most instances.

```
Initial_encoding_width (M, P, IS )
 /*phase 1. CM/OM guided We computation*/
Backtrace from P to BFVs on easy-to-control paths;
Extract variables on traced paths to set S;
for each isel in IS
     Remove hard-to-control branches from S guided by CM;
end for
Set We=min{max{celling(W/6), 6},W}&(Ce=0) for all PIs in S;
Set We=min{2,W}&(Ce=0) for all other PIs in M;
Propagate We for all other variables in M;
 /* phase 2.  We adjustment */
for each constant O connected with predicate OP
  if(O > 0)
      Set W_T=ceiling(log2(O), C_T=0;
  else
      Set W_T=ceiling(log2(-O), C_T=1;
  endif
  for each V related to O with predicate OP
     if((We of V < W_T) || (Ce!= C_T) )
        Adjust We=W_T and (Ce=C_T) for V;
        Adjust We for PIs that Vs depends;
     endif
  end for
end for
Propagate adjusted We for all other variables in M;
end
```

Figure 3. Alg. of Initial W_e Computation

3.2 Guided Initial W_e Computation

The initial W_e of variables including the constant value C_e chosen on bits outside of W_e are very important to the efficiency of our algorithm. We enforce the same constant value C_e to all bits beyond W_e starting from the most significant one of bitvector variables. The algorithm of the initial W_e computation with two phases is shown in Fig. 3. In phase 1, it identifies the PIs on the easy-to-control paths and sets a larger W_e to them; the other PIs are given a smaller W_e. Guided by CM and OM, our algorithm first backtraces from the target property along the easy paths in the Boolean portion of the instance to a set of BFVs, then it extracts all non-Boolean part variables in the cone of these BFVs and place them in a new set S. It removes from S any variable on the hard-to-control branches of ITEs connected with the $isel$s in the IS. We empirically set C_e as 0, choose 6 as the initial W_e for the PIs in S and 2 for the remaining PIs.

In phase 2, our algorithm adjusts the computed W_e and C_e considering the effects of constants in M. We observe from experiments that it is preferable to give the same negative or positive polarity for the variables with which the constant is computed. We set the W_e of variables that allow their encoded value range to cover the absolute value of the constant, especially for predicate operations. This is to avoid fixing the output value of such predicate operations. For example, consider the constraint $a > -4$, its value is not fixed only when setting W_e bigger than 2 and $C_e = 1$ for a. Finally, the adjusting of W_e and C_e on PIs are also propagated to all internal variables while considering computation consistency on the operators along the paths.

3.3 Abstract Counterexample Generation

In this step, our algorithm sequentially enforces two kinds of constant values on M_o to steer the search for an abstract counterexample as shown in Fig. 4. The set S_p which consists of some BFVs and $isel$s included in the M_o is constructed beforehand. A combination of constant values is first imposed to the variables in the set S_p so as to restrict searching on a subset of paths. The function $pathsSelGen$, which returns such constant assignments $\{C_1, .., C_K\}$, starts enumerating variables in S_p at the first iteration and stops until the verification is finished. Since no SAT assignment was found in M_u, we assume a low chance that a SAT one

```
//Choose K path selection variables from first Mo model to set Sp
 Cex_gen (Mo, Sp )
while (1)
  {C1,..,CK} = pathsSelGen(Sp);
  Enforce {C1,..,CK} assignment on Mo to get Mo*;
  if(Mo* is SAT)   break;
endwhile
Sort all pseudo PIs of Mo w.r.t. CM in increasing order;
Divide all sorted pseudo PIs into L groups;
Enforce current We for all pseudo PIs to get M**;
for i = 1 to L
  Refine M** by enlarging We for pseudo PIs in group i;
  if(Refined M** is SAT)
     CEXRET = SAT-SOL;
     break;
  endif
endfor
end
```

Figure 4. Abstract Counterexample Generation

exists on the partition of the instance similar to M_u, and we expect that the new M_u bears the least similarity to the previous M_u in the search space. So, our goal is to return a value combination with the least similarity to the last one and not found as infeasible so as to constrain the search in a different subset of paths. Once a value assignments is found satisfying M_o, we apply it to M_o to restrict the search space to ease the generation task. Next, small encoding widths W_es are applied to some pseudo PIs of M_o to further constrain the search. A greedy search starts from the most easily controllable PIs. The goal is that the value assignments on the harder-to-control pseudo PIs in the returned counterexample can still be covered by their present W_e.

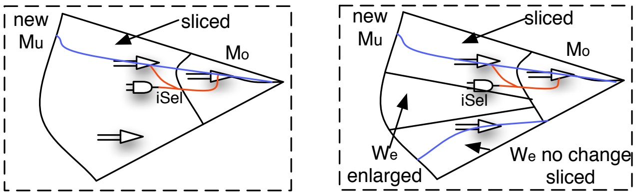
Theorem 1. *An infeasible assignment on M_o is also infeasible in M.*

Proof. Since the set of clauses β_o of M_o is a subset of CNF clauses β of M, if an assignment $v_0, .., v_n$ cannot satisfy β_o, this same assignment also cannot satisfy β because at least a subset (β_o) cannot be satisfied in β. □

With this theorem, we can safely check the invalidity of some assignments in the M_o with a low cost. It is especially efficient to identify a subset of infeasible paths using M_o.

3.4 New W_e with Guided Slicing

We focus on introducing the guided slicing process as illustrated in Fig. 5. First, we apply the assignments $\{C_1, .., C_K\}$ (that were obtained from M_o^*) to the new M_u and slice away any dangling variables. In Fig. 5(a), the $iSel$ whose value is enforced controls the *ITE*s of M both inside M_o and outside of M_o. So more branches can be sliced away of the new M_u compared to M_o. Next, the branches of the ITEs controlled by $isel$s in IS on which variables had no enlarged W_e are removed. In Fig. 5(b), the variables whose W_e need to be changed are in the center cone. The path branches shown in the bottom are outside of this cone and are removed from the new M_u.



(a) Step 1 (b) Step 2

Figure 5. Two-Steps Guided Slicing

4 Experimental Results

To validate the effectiveness of our approach, we implemented the proposed method in C++, which is called C2BIT-2, and applied it to the bitvector arithmetic benchmarks [1] in the following suites: $Spear$ and $TACAS07$. There are two main reasons that 14 benchmarks were chosen: Either 1) a benchmark is path intensive that has a large amount of $isel$s or BFVs, or 2) verification takes a long time by existing tools. They are all refutable properties (a satisfiable solution exists). C2BIT-2 uses Booleforce v1.0 to extract UNSAT core in the UNSAT case, and uses MINISAT v1.14 to generate abstract counterexamples and verify the satisfiability of the under-approximate model. The experiments were conducted on an Intel Xeon 2.8GHz processor with 2 GB RAM.

The results are reported in Table 1. First, for every benchmark, the characteristics of the original bitvector formula are given: Vars# reports the number of bitvector variables including constants; $iSel$# reports the number of $iSel$s both before and after pruning. After pruning the formula instance, we found that only about one-third of $isel$s are unique. For example, in log_vc331256, only 1019 out of 3110 $isel$s are unique. This implies that multiple ITE variables may share the same $isel$. Furthermore, these $isel$s have no dependency to each other and their inputs are close to PIs, thus they are very controllable. For the last benchmark, $isel\#$ is zero which means no ITE variables but the number of BFVs is big shown in parentheses. Next, the runtime of Spear v2.6 is reported. We apply three methods on the benchmarks: 1) uniform initial W_e with the proof-based refinement as in [2]; 2) uniform initial W_e with CM/OM guided refinement; 3) non-uniform initial $W_e < max, min >$ with CM/OM guided refinement (C2BIT-2). For each benchmark, the initial/final *modal* value of encoding widths after refinement are reported, followed by the number of refinement iterations and total runtime (including pruning, encoding, UNSAT core extraction and solving time) of these three methods. For instance, in innd_33359, the initial uniform W_e is 8 and the final *mode* of W_e after two iterations is 32 for method 1 and 12 for method 2. With method 3) the non-uniform initial W_e is $< 12, 2 >$ (variables on easy-to-control paths have width of 12 and the rest have widths of 2), and the final *mode* of W_e is still 12 as only few variables have the enlarged W_e after refinement. All methods need 2 iterations to find the SAT solution for this particular instance but C2BIT-2 is the fastest.

Table 1. Results Comparison (Benchmarks from Spear, TACAS07)

Benchmark	Vars#	isels#	Spear	Proof-Based			CM/OM Guided			C2BIT-2		
			(s)	W_e	Iter.	T(s)	W_e	Iter.	T(s)	$W_e <, >$	Iter.	T(s)
log_331256	27773	3110/1019	314.42	$4/12$	2	117.71	$4/6$	2	43.67	$< 6, 2 >$	1	10.21
log_331257	27369	3149/994	437.13	4/18	2	76.59	4/12	2	47.27	$< 6, 2 >$	1	9.38
log_331190	23188	2656/852	632.33	4/18	2	46.96	4/12	2	34.36	$< 6, 2 >$	1	5.34
log_331211	23554	2681/877	515.31	4/12	2	91.20	4/12	2	33.32	$< 6, 2 >$	1	6.47
innd_33359	1368	77/33	49.66	8/32	2	41.28	8/12	2	37.90	$< 12, 2 > /12$	2	30.29
innd_33725	1025	52/22	55.26	8/32	2	43.32	8/12	2	37.16	$< 12, 2 > /12$	2	33.71
nnrpd_21453	1330	76/32	20.94	8/18	2	47.21	8/18	3	50.76	$< 12, 2 > /12$	2	34.39
wget_17909	1042	37/32	380.74	8/18	2	520.3	8/18	2	148.97	$< 12, 2 > /12$	2	50.38
wget_18506	1062	38/30	39.86	8/18	2	30.26	8/18	2	29.14	$< 12, 2 >$	1	15.34
cli_1225314	17481	1991/604	40.45	4/12	2	33.52	4/12	2	27.70	$< 6, 2 >$	1	5.2
cli_1225757	18171	2090/616	25.02	4/12	2	36.79	4/12	2	25.84	$< 6, 2 >$	1	6.21
cli_1225783	18766	2159/637	51.06	4/6	2	57.80	4/6	2	28.19	$< 6, 2 >$	1	4.53
cli_1225832	19867	2273/665	65.25	4/12	2	75.28	4/12	2	29.67	$< 6, 2 >$	1	6.31
S-40-50	1321	0(156)/0	14.36	4/32	4	98.47	4/16	2	30.29	$< 8, 2 >$	1	12.31

Compared with Spear, C2BIT-2 has achieved significant speedups. Some were even greater than $10\times$. Note that the assignment found for each benchmark was validated using the CNF file and the variable mapping file generated by Spear. We also observe that the maximum W_e computed by two refinement methods are the same or similar for many benchmarks. One reason is that modern SAT solvers typically return a ‘maximally-false’ solution that contains as many false bits as possible that can produce small value assignments. However, with the CM/OM guidance, the enlargement of variables’ encoding widths focuses on the subset of paths so that the slicing can be conducted to reduce the model size and solving time. With C2BIT-2, the SAT assignment for 10 out of 14 benchmarks can be obtained in just one iteration with our initial non-uniform bitwidth encoding. It shows that this encoding effectively increases the chances of finding a SAT solution on the easy paths and invokes the little effort of searching on hard-to-control paths.

5 Conclusion

We have presented an efficient algorithm of computing small encoding widths for bitvectors by utilizing a path-oriented abstraction-refinement framework. This algorithm exploits both the high-level structure and dynamic verification knowledge to effectively steer the search. It takes advantage of the controllability and observability metrics to guide three major steps: initial encoding width computation, abstract counterexample generation, and under-approximate model slicing. Experiments show that our proposed algorithm can reduce the solving time significantly, especially in verifying the paths-intensive designs.

References

[1] D. Babi’c. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[2] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proc. of TACAS*, 2007.

[3] E. Clarke and D. Kroening. A tool for checking ansi-c programs. In *Proc. of TACAS*, pages 168–176, 2004.

[4] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. of TACAS*, pages 570–574, 2005.

[5] F.Ivancic, I. Shlyakhter, M. Ganai, and A. Gupta. Model checking c programs using f-soft. In *Proc. of ICCD*, 2005.

[6] N. He and M. Hsiao. Bounded model checking of embedded software in wireless cognitive radio systems. In *Proc. of ICCD*, 2007.

[7] N. He and M. Hsiao. A new testability guided abstraction to solving bit-vector formula. In *International Workshop on Bit-Precise Reasoning*, 2008.

[8] F. F. Hsu and J. H. Patel. High-level variable selection for partial-scan implementation. In *Proc. of ICCAD*, 1998.

[9] C. Y. Huang and K. T. Cheng. Assertion checking by combined word-level atpg and modular arithmetic constraint-solving techniques. In *Proc. of DAC*, 2000.

[10] P. Johannsen and R. Drechsler. Formal verification on the rt level computing one-to-one design abstractions by signal width reduction. In *Proc. IFIP International Conference on Very Large Scale Integration*, 2001.

[11] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for rtl-level verification. In *Proc. of ICCAD*, pages 786–793, 2006.

[12] T. Nguyen, M. Delaunay, and C. Robach. Testability analysis for software components. In *Proc. of ICSM*, 2002.

[13] K. Thearling and J. Abraham. An easily computed functional level testability measure. In *Proc. of ITC*, 1989.

[14] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proc. of TOPLAS*, 2005.

[15] A. Zaks, Z. Yang, I. Shlyakhter, and F. I. etl. Bitwidth reduction via symbolic interval analysis for software model checking. In *TCAD*, pages 1513–1517, 2008.