Formal Verification of Optimizing Transformations during High-level Synthesis

Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri
{r.chouksey,ckarfa,pbhaduri}@iitg.ac.in
Department of Computer Science & Engineering
Indian Institute of Technology Guwahati
Guwahati, Assam, India

ABSTRACT

Translation validation is the process of proving that the target code is a correct translation of the source program being compiled. In this work, we propose a translation validation method to verify code motion transformations involving loops applied during the scheduling phase of high-level synthesis (HLS). Our method is capable of ignoring false computations during translation validation. In this work, we show that how to generate a counter-trace (*cTrace*) using the internal information of verifier in the case of non-equivalence reported by a translation validation method. We also show how a Bounded Model Checker (CBMC) can be used to find a counter-example for a given *cTrace*. Experimental results demonstrate the usefulness of our method.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**.

KEYWORDS

Translation Validation, Equivalence Checking, Code Motion, Finite State Machine with Datapath (FSMD), CBMC, Counter-example Generation

1 INTRODUCTION

High-level synthesis (HLS) techniques translate high-level languages like C/C++ into register transfer level design [9]. Due to its complexity, proving the correctness of an HLS tool is prohibitively expensive. Code motion based optimizations are used in the scheduling phase of HLS tools to improve the quality of synthesis results. Such transformations move operations across the boundaries of basic blocks. They are widely used to improve the quality of synthesis results for designs with complex and nested conditionals and loops. Code motion techniques change the data-flow of a behavior considerably. Therefore, it is necessary to verify the semantic equivalence between the input behavior to HLS (i.e., source behavior) and the scheduled behavior generated by HLS (i.e., transformed behavior).

Translation validation is the process of verifying, for each translation that HLS tool performs, that the target code generated by the tool is a correct translation of the source code. This is proved by showing the equivalence between the source and transformed behaviors. Even if this approach does not guarantee that the HLS tool is bug-free, it guarantees that any error in translation will be caught when the tool runs. The present work is aimed at developing translation validation methodologies, specifically, for code motion transformations involving loops.

Many path-based equivalence checking approaches [11–13] have been proposed to handle code motion based transformations during HLS where behaviors are represented by a finite state machine with datapaths (FSMD). In general, path based approaches decompose each FSMD into a finite set of finite paths and the equivalence of FSMDs is established by showing path level equivalence between two FSMDs. However, all these methods cannot handle the transformations that result in code motion across loops. A Value Propagation based equivalence checking (VP) method was proposed in [1] which can handle code motion across loops. There are three possible scenarios during code motion transformations involving loops:

S_1 : Some code segment before a loop body is placed after the loop body or vice versa (i.e., code motion across loops).
S_2 : Some code segment is moved before the loop from inside the loop body.
S_3 : Some code segment is moved after the loop from inside the loop body.

We identify the following limitations of the VP method presented in [1].

- The VP method does not check whether a computation is a false computation [3] i.e., it never executes. As a result, it gives false negative results in the case of loop invariant code motion involving false computations.
- The VP method can handle scenario S_1 but it cannot handle scenarios S_2 and S_3.

In case of non-equivalence, these path-based equivalence checking approaches, in general, report that the behaviors "May Not be equivalent" and the path for which possible non-equivalence arises in the equivalence checker. This information is not sufficient for debugging the issue. A counter-example which will reproduce the non-equivalence between the source and the transformed behaviors will add significant value to the adoption of such path-based approaches.

1.1 Problem statements

The objective of this work to develop a translation validation methodologies, specifically, for code motion transformations involving loops that occur during the scheduling phase of HLS. Specifically, the following verification problems will be addressed:

(1) Ignoring false computations during translation validation.
(2) Translation validation of code motion transformation involving scenarios S_1, S_2 and S_3 (as discussed in Section 1).
(3) Generating a *cTrace* and how the CBMC [6] tool can be used to find a suitable counter-example for the given *cTrace* in

```
for ($i_1 = L_1; i_1 \leq H_1; i_1 += r_1$)
   for ($i_2 = L_2; i_2 \leq H_2; i_2 += r_2$)
      ⋮
      for ($i_n = L_n; i_n \leq H_n; i_n += r_n$)
          $S_n$ :   ...
```

Figure 1: Nested loop structure

case of non-equivalence reported by translation validation approaches.

2 CONTRIBUTIONS

In the following, we outline in brief the contributions of work on each of the objectives identified in Subsection 1.1.

2.1 The enhanced value propagation method

The EVP method of FSMDs described in [4] is based on propagating the mismatched values (as a propagated vector) of live variables through all the subsequent path segments until the values match or the final path segment ending in the reset state is reached. A *propagated vector* for a path β is an ordered pair $\langle R_\beta, s_\beta \rangle$, where the first element is the condition of execution (R_β) representing the condition that must be satisfied at the start state of β and the second element is an updated variable vector (s_β) representing the symbolic value obtained by the variables at the end state of β.

In the course of equivalence checking of two FSMDs, two paths, β and α say (one from each FSMD), are compared with respect to their corresponding propagated vectors for finding a path equivalence. If the conditions of execution and the data transformations of the two paths are equal, then they are declared as *unconditionally equivalent* (U-equivalent in short, denoted by $\beta \simeq \alpha$). If some mismatch in data transformation is detected, then they are declared to be conditionally equivalent (C-equivalent in short, denoted by $\beta \simeq_c \alpha$) provided their final state-pairs eventually lead to some U-equivalent paths; otherwise, these two paths and, therefore, two FSMDs are declared to be not equivalent. Once a C-equivalent path is identified, the VP method tries to find a U-equivalent path in a depth-first search (DFS) manner.

2.2 Handling false computation involving loops

The EVP method avoids the false computation by automatically extracting a formula that checks whether a loop will always execute at least once under a propagated condition. Let us consider the nested loop structure of depth n shown in Fig. 1. If formula 1 shown below is valid then the statement S_n, at the loop structure of nesting depth n, will always execute at least once.

$$C_p \implies \Bigg(\exists i_1, \exists i_2, \cdots, \exists i_{n-1}, \exists a_1, \exists a_2, \cdots, \exists a_{n-1} \\ \Big((L_n \leq H_n) \wedge \Big(\bigwedge_{x=1}^{n-1} f_x\Big)\Big)\Bigg) \quad (1)$$

where $f_x = \Big((L_x \leq i_x \leq H_x) \wedge (i_x = a_x r_x + L_x) \wedge (a_x \geq 0)\Big)$. Here C_p is the propagated condition before entering the nested loop of depth n. For checking the validity of this formula, EVP method uses the SMT solver Z3 [7] in the theory of linear integer arithmetic. This formula will guide the EVP method during equivalence checking to identify and ignore false computations. More importantly, EVP method can handle any level of loop nesting. The details can be found in [3].

2.3 Handling loop invariant Code motion

The EVP method is capable of handling all the three scenarios, i.e., S_1, S_2 and S_3. Only codes/operations that are invariant to a loop are allowed to move across/from loop (as discussed in scenarios S_1, S_2 and S_3 in Introduction). In the EVP method, we are essentially ensuring that any code motion involving loops is actually loop invariant. The overall approach is discussed below.

To detect the validity of code motion involving loops, the EVP method marks the live variables which exhibit a mismatch in the propagated vector. Those variables on which these marked variables depend are also marked in the propagated vector. The rest of the variables are denoted as unmarked variables. Let q_{0i} be the entry/exit state of a loop body in M_0 and its corresponding state q_{1j} be the entry/exit state of a loop body in M_1. The state q_{0i} has the propagated vector ϑ_{0i} before entering the loop and the propagated vector ϑ'_{0i} after traversal of one of the paths inside the loop leading to q_{0i}. Similarly, state q_{1j} has the propagated vector ϑ_{1j} before entering the loop and the propagated vector ϑ'_{1j} after traversal of one of the paths inside loop leading to q_{1j}. During code motion involving loops the following cases may arise:

Case 1 *Unmarked Variable*: There are two possibilities for an unmarked variable, say x. It may be noted that x has symbolic values in both ϑ_{0i} and ϑ_{1j}.

Case 1.1 If x has the same value in ϑ'_{0i} and ϑ'_{1j} as shown in Fig. 2(a) then after exiting the loop x is reverted to its symbolic value.

Case 1.2 If there is a mismatch for x in ϑ'_{0i} and ϑ'_{0j} then there is a possibility of scenario S_3. Let $e_{x_{0i}}$ and $e_{x_{1j}}$ represent the mismatched values in ϑ'_{0i} and ϑ'_{1j} respectively as shown in Fig. 2(b). To check the validity of the code motion, we do the following test.

(1) The expressions $e_{x_{0i}}$ and $e_{x_{1j}}$ should be invariant in their corresponding loops.
(2) The variable x is not used before being defined in both the loops.

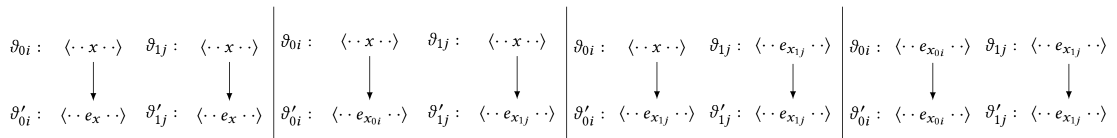
Case 2 *Marked Variable*: Marked variables arise in the case of S_1 and S_2. There are three possibilities for a marked variable.

Case 2.1 Suppose a marked variable, say x, has its symbolic value at ϑ_{0i} and $e_{x_{1j}}$ at ϑ_{1j}. If after executing the loop once the value of x matches in both the loops (i.e., x has the same value ($e_{x_{1j}}$) in ϑ'_{0i} and ϑ'_{1j}) as shown in Fig. 2(c), then scenario S_2 is possible. To check the validity of the code motion, we do the following test.

(1) The expression $e_{x_{1j}}$ should be invariant in both the loops.
(2) The variable x is not used before being defined in the loop at q_{0i}, and it has no definition in the loop at q_{1j}.

Case 2.2 Suppose x has its symbolic value at ϑ_{1j} and $e_{x_{0i}}$ at ϑ_{0i} and after executing the loop once the value of x matches in both the loops. This case can be handled in a manner similar to case 2.1.

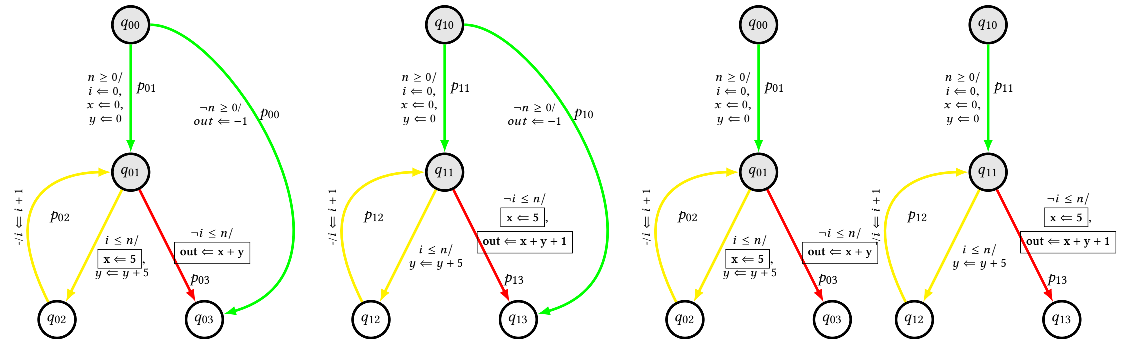
Case 2.3 In the remaining case, if before executing the loop and after exiting the loop the value of x remains the same in both the



ϑ_{0i} : $\langle \cdots x \cdots \rangle$ ϑ_{1j} : $\langle \cdots x \cdots \rangle$
ϑ'_{0i} : $\langle \cdots e_x \cdots \rangle$ ϑ'_{1j} : $\langle \cdots e_x \cdots \rangle$
ϑ_{0i} : $\langle \cdots x \cdots \rangle$ ϑ_{1j} : $\langle \cdots x \cdots \rangle$
ϑ'_{0i} : $\langle \cdots e_{x_{0i}} \cdots \rangle$ ϑ'_{1j} : $\langle \cdots e_{x_{1j}} \cdots \rangle$
ϑ_{0i} : $\langle \cdots x \cdots \rangle$ ϑ_{1j} : $\langle \cdots e_{x_{1j}} \cdots \rangle$
ϑ'_{0i} : $\langle \cdots e_{x_{1j}} \cdots \rangle$ ϑ'_{1j} : $\langle \cdots e_{x_{1j}} \cdots \rangle$
ϑ_{0i} : $\langle \cdots e_{x_{0i}} \cdots \rangle$ ϑ_{1j} : $\langle \cdots e_{x_{1j}} \cdots \rangle$
ϑ'_{0i} : $\langle \cdots e_{x_{0i}} \cdots \rangle$ ϑ'_{1j} : $\langle \cdots e_{x_{1j}} \cdots \rangle$

(a) Case 1.1 (b) Case 1.2 (c) Case 2.1 (d) Case 2.2

Figure 2: A case (a) where unmarked variable x is defined identically in both the loops; (b) where unmarked variable x has some mismatch at the end of the loop; (c) where a marked variable x has the same value at the end of the loop; (d) where the values of the marked variable x do not update in both the loops



q_{00}
$n \geq 0/$ $i \Leftarrow 0,$ $x \Leftarrow 0,$ $y \Leftarrow 0$
p_{01}
$\neg n \geq 0/$ $out \Leftarrow -1$
p_{00}
q_{01}
$-/i \Leftarrow i+1$
p_{02}
$i \leq n/$ $\boxed{x \Leftarrow 5}$, $y \Leftarrow y+5$
$\neg i \leq n/$ $\boxed{out \Leftarrow x+y}$
p_{03}
q_{02}
q_{03}
q_{10}
p_{11}
p_{10}
q_{11}
p_{12}
$i \leq n/$ $y \Leftarrow y+5$
$\neg i \leq n/$ $\boxed{x \Leftarrow 5}$, $\boxed{out \Leftarrow x+y+1}$
p_{13}
q_{12}
q_{13}

(a) Source behavior M_0 (b) Transformed behavior M_1 (c) *cTrace* of M_0 (d) *cTrace* of M_1

Figure 3: Counter-trace Genegartion

loops as shown in Fig. 2(d) then scenario S_1 is possible. To check the validity of code motion, we do the following test.

(1) Variable x is not updated within the loop.
(2) All those variables on which the variable x depends should not be updated within the loop.

A code motion involving loop is a valid code motion if each marked and unmarked variable satisfies their respective cases as mentioned above. The details can be found in [4].

2.4 Counter-example generation using counter-trace

The translation validation approaches are proven to be sound but not complete. Therefore, all these approaches report the behaviors "May Not be equivalent" once they fail to prove the equivalence of source and transformed behaviors. During the course of equivalence checking the EVP method maintains two lists: `EQ_LIST` contains equivalent path pairs explored so far and `C_LIST` contains candidates for conditionally equivalent path pairs. In the paper [2, 5], we have shown how the equivalence information maintained in these lists by the EVP method can be used to find a *cTrace* in the case of non-equivalence reported by the EVP method. We have also shown in [5] how CBMC can be used to find suitable initialization values for input variables i.e., counter-example for a given *cTrace*. A counter-example demonstrates the non-equivalence between the source and transformed behaviors and the EVP method reports "Not equivalent" instead of "May Not be equivalent" in this case. Since the equivalence problem is not complete, CBMC may not always produce counter-example in case of non-equivalence reported by equivalence checker. This indicates a possible false negative results of the equivalence checker. It would be an interesting work to enhance the equivalence checker method to handle the false negative scenarios identified by our counter-example generation mechanism. In the following, working of our overall verification procedure is explained with Example 1.

Example 1. *Consider the input behavior M_0 and its transformed behavior M_1 shown in Fig. 3. The operation $x \Leftarrow 5$, a loop invariant for input behavior M_0, is placed after the loop body in the transformed behavior M_1 (scenario S_3). Note that the input behavior M_0 and the transformed behavior M_1, shown in Fig. 3, are not equivalent since there is mismatch in values of the variable out. In the course of equivalence checking, the EVP method declares that the path pairs (p_{00}, p_{10}) and (p_{01}, p_{11}) are U-equivalent and the path pairs are stored in the* `EQ_LIST`*. When the path pair (p_{02}, p_{12}) are compared, the values of x mismatches. Since, $x \Leftarrow 5$ is a loop invariant for loop body p_{02} and x is not used before defining it, it is a valid code motion (case 1.2). Hence, the path pair (p_{02}, p_{12}) declared to be candidate for C-equivalent and the path pair is stored in* `C_LIST`*. The EVP method finds that path pair (p_{03}, p_{13}) differs in the values of the variable out and reports a possible non-equivalence between these two behaviors. Since EVP method fails to show equivalence, counter-example generation mechanism uses* `EQ_LIST` *and* `C_LIST` *to find cTrace of M_0 (Fig. 3(c)) and cTrace of M_1 (Fig. 3(d)). We encodes the equivalence of these cTraces in CBMC. The CBMC produces a counter-example. It shows that for $n = 0$ the values of the variable 'out' differs thus two cTraces and hence the source and the transformed behaviors are not equivalent.*

Algorithm 1: containmentChecker(FSMD M_0, FSMD M_1)

```
1  Compute the path cover $P_0$ and $P_1$ of $M_0$, $M_1$, respectively; $W_{csp}$ is
     a set of corresponding state pairs and initially contains $(q_{00}, q_{10})$;
     EQ_LIST contains the pair of U(C)-equivalent pair; C_LIST
     contains the candidate of C-equivalent path pairs; initially
     EQ_LIST and C_LIST are empty;
2  foreach $(q_{0i}, q_{1j}) \in W_{csp}$ do
3      output ←
         ECC($M_0$, $M_1$, $q_{0i}$, $q_{1j}$, $P_0$, $P_1$, $W_{csp}$, EQ_LIST, C_LIST);
4      if output is "May Not be Equivalent" then
5          Report "unable to decide $M_0 \sqsubseteq M_1$" and exit;
6      else if output is "Not Equivalent" then
7          Report a counter-example and declare "$M_0 \not\equiv M_1$";
8      end if
9  end foreach
10 Report "$M_0 \sqsubseteq M_1$";
```

Algorithm 2: ECC(M_0, M_1, q_{0i}, q_{1j}, P_0, P_1, W_{csp}, EQ_LIST, C_LIST)

```
1  foreach path $\beta : (q_{0i} \Rightarrow q_{0m})$ in $P_0$ do
2      if $q_{0i}$ is a loop header and checkFalseComputation($q_{0i}$)
         returns True then
3          continue;
4      end if
5      if path $\alpha : (q_{1j} \Rightarrow q_{1n})$ can be found in $P_1$ such that $\beta \simeq \alpha$
         then
6          $W_{csp} = W_{csp} \cup \{(q_{0m}, q_{1n})\}$;
7          Insert $(\beta, \alpha)$ in EQ_LIST.
8      else if path $\alpha : (q_{1j} \Rightarrow q_{1n})$ can be found in $P_1$ such that
         $\beta \simeq_c \alpha$ then
9          if $q_{0m}$ or $q_{1n}$ is reset state then
10             goto step 18
11         else if $q_{0m}$ or $q_{1n}$ appears as the final state of some path
             already in C_LIST ∧ loopInvariant($\beta$, $\alpha$) then
12             goto step 18/* Propagated values are not loop
                 invariant                                        */
13         else
14             Insert $(\beta, \alpha)$ in C_LIST.
15             ECC($M_0$, $M_1$, $q_{0m}$, $q_{1n}$, $P_0$, $P_1$, $W_{csp}$, EQ_LIST, C_LIST);
16         end if
17     else
18         if counterExmapleGenerator($M_0$, $M_1$, $\beta$, $\alpha$,
             EQ_LIST, C_LIST) returns a counter-example then
19             return "Not equivalent";
20         else
21             return "May Not be Equivalent";
22         end if
23     end if
24 end foreach
25 EQ_LIST = EQ_LIST ∪ {Last member of C_LIST}
26 C_LIST = C_LIST \ {Last member of C_LIST}
27 return "success";
```

```
int main(){
 int x,i,n,z=0,out;
 x=0;
 for(i=4;i<n;i++){
  x = 5;
  z=z+x;}
 out=z+x;
 return out;}
```

(a) Input Behavior

```
int main(void){
 int x,i,n,z,out,sT0_5;
 int returnVar_main;
 z = 0;x = 0;i = 4;x = 5;
 do{
  sT0_5 = (i < n);
  if (sT0_5){
   z = (z + x);
   i = (i + 1);}
  else break;
 }while (1);
 out = (z + x);
 returnVar_main = out;
 return returnVar_main;}
```

(b) Transformed Behavior

Figure 4: A bug in SPARK

3 OVERALL VERIFICATION METHOD

In this section we present an overall verification method. We begin the procedure of equivalence checking by invoking the function `containmentChecker` (Algorithm 1). To obtain a path cover, the function `containmentChecker` breaks down an FSMD into smaller segments by introducing cutpoints so that each loop in the FSMD is cut by at least one cutpoint. This is based on the Flyod-Hoare method of program verification [8]. The set of all paths from a cutpoint to another cutpoint without any intermediate occurrence of a cutpoint is a path cover of the FSMD. The function `containmentChecker` invokes enhanced correspondence checker (`ECC`) function (Algorithm 2) for each corresponding state pairs, one by one (in step 3). Depending on the output returned by `ECC`, `containmentChecker` outputs the decision whether the original FSMD is contained in the transformed FSMD or not. The function `ECC` (Algorithm 2) is the key function of our verification method. The function `ECC` returns "success" if for every path emanating from q_{0i} has an equivalent path originating from q_{1j} is found (in step 27 of Algorithm 2); otherwise, it returns "failure". In case of failure, function `ECC` invokes the function `counterExmapleGenerator` (in line 18). If `counterExmapleGenerator` returns a counter-example then the function `ECC` returns "Not equivalent" i.e., the two FSMDs are not equivalent (in step 19) otherwise the function `ECC` returns "May Not be Equivalent" (in step 21). To avoid the false computations at the loop header `ECC` invokes the function `checkFalseComputation` (in step 2). The function `checkFalseComputation` returns `True` if the loop at q_{0i} under the propagated condition will execute at least once, over all possible inputs in M_0. It returns `False` otherwise. If a loop has been crossed over then the function `ECC` invokes the function `loopInvariant` (in step 11). The function `loopInvariant` checks the validity of code motion involving loops. The function `loopInvariant` returns `TRUE` if each marked and unmarked variables satisfy their respective cases as mentioned in Subsection 2.3. If it returns `FALSE` then the function `ECC` returns "failure".

4 EXPERIMENTAL RESULTS

We have enhanced the VP method [1] by incorporating all proposed solutions discussed in Section 2. We have used HLS tool SPARK [10] to generate the test cases. All the experiments have been conducted

Table 1: Experimental results on test cases where the VP method fails

Benchmarks	VP		EVP	
	Decision	Time (ms)	Decision	Time (ms)
simple_types_ loop_invariant	MNEq	4	Eq	12
mandel	MNEq	4	Eq	16
mandel2	MNEq	4	Eq	16
himenobmtxpa	MNEq	4	Eq	20

on a laptop with 2 GHz Intel Core 2 Duo processor with 3 GB of RAM.

In our first experiment, we take some of the test-suite distributed with LLVM [14]. These benchmarks contain some loop invariant operations. We forced SPARK to apply loop invariant code motion (LICM) transformation to obtain the transformed behavior. These test cases represent the scenarios S_2 and S_3 as described in Section 1. The results of these experiments are tabulated in Table 1. It is evident from Table 1, that our EVP method can correctly identify the equivalence even when some loop invariant operation op is moved before (after) the loop from inside it. This is reported as 'Eq' in the Table 1. However, the VP method reports "May Not be equivalent" in these cases and denoted as 'MNEq' in the Table 1. It may be noted that our EVP method additionally can handle the scenarios where the existing method gives false negative results.

During this experimentation, we found a bug in the implementation of the LICM algorithm in the SPARK tool as shown in Fig. 4. Here the operation $x = 5$ is moved before the loop body in the transformed behavior. The output of these behaviors will not be the same for any input $n \leq 4$. This behavior is proved to be non-equivalent by our EVP method. Thus, our method finds a previously unknown bug in a widely used HLS framework.

In our second experiment, the benchmarks are taken from [1]. We have manually introduced few changes like addition, multiplication or subtraction of a constant to some of the variables in the benchmarks so that source and transformed behaviors become non-equivalent. The results of our experiment are tabulated in Table 2. In the benchmarks listed in Table 2, the VP method fails to prove the equivalence of source and transformed behaviors. It reports the behaviors "May Not be equivalent". In all these cases, CBMC finds a mismatch in an output variable and generates a suitable counter-example. Hence, our method concludes that the behaviors are "Not equivalent" which is denoted as 'NEq' in the table. The time required by our method is little high compared to VP method in the case of non-equivalence as our method needs need to run CBMC on the *cTrace*. This experiment shows that with the help of our counter-example generation scheme our method can take strong decision about non-equivalence of behaviors. Moreover, counter-example provided by the our method will help the user to debug the root-cause of non-equivalence.

5 FUTURE PLANS

- We intend to enhance the EVP method to validate transformations such as (i) two or more consecutive if-else blocks are merged into one and (ii) multiple branches of an if-else block are merged into one.
- We plan to enhance the EVP method to handle the false negative scenarios identified by our counter-example generation mechanism.

Table 2: Experimental Results

Bench-marks	#Path	#State		Decision		Time (ms)		#Line in C prog. (CBMC input)
		M_0	M_1	VP	Our	VP	Our	
DCT	1	8	16	MNEq	NEq	85	766	185
PERFECT	7	6	4	MNEq	NEq	56	227	74
MODN	9	8	9	MNEq	NEq	66	890	137
GCD	11	8	4	MNEq	NEq	31	100	97

REFERENCES

[1] Kunal Banerjee, Chandan Karfa, Dipankar Sarkar, and Chittaranjan A. Mandal. 2014. Verification of Code Motion Techniques Using Value Propagation. *IEEE TCAD* 33, 8 (Aug 2014), 1180–1193.

[2] Ramanuj Chouksey, Chandan Karfa Kunal Banerjee, Pankj Kalita, and Purandar Bhaduri. forthcoming. A Counter-Example Generation Procedure for Path based Equivalence Checkers. *IET Softeware* (forthcoming).

[3] Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri. 2017. Translation Validation of Loop Invariant Code Optimizations Involving False Computations. In *VDAT*. 767–778.

[4] Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri. 2018. Translation Validation of Code Motion Transformations Involving Loops. *IEEE TCAD* (2018). https://doi.org/10.1109/TCAD.2018.2846654

[5] Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri. forthcoming. Improving Performance of a Path-Based Equivalence Checker using Counter-Examples. In *VLSID*.

[6] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 168–176.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008 (LNCS)*, Vol. 4963. Springer, Berlin, Heidelberg, 337–340.

[8] Robert W Floyd. 1967. Assigning meanings to programs. *Mathematical aspects of computer science* 19, 1 (1967), 19–32.

[9] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. 1992. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, USA.

[10] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. 2003. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings, VLSID*. IEEE, 461–466.

[11] Chandan Karfa, Chittaranjan A. Mandal, and Dipankar Sarkar. 2012. Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM TODAES* 17, 3 (Jul 2012), 30.

[12] Chandan Karfa, Dipankar Sarkar, Chitta Mandal, and P. Kumar. 2008. An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis. *IEEE TCAD* 27, 3 (Mar 2008), 556–569.

[13] Youngsik Kim, Shekhar Kopuri, and Nazanin Mansouri. 2004. Automated Formal Verification of Scheduling Process Using Finite State Machines with Datapath (FSMD). In *ISQED 2004*. IEEE Computer Society, 110–115.

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO 2004*. IEEE, 129–142.