Translation Validation of Code Motion Transformations Involving Loops

Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri

***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. We have also identified a scenario involving code motion across loops where the state-of-the-art translation validation method gives false positive results. Our method can prove the non-equivalence of the concerned finite state machines with data paths (FSMDs) in this scenario. We detected a bug in the HLS tool SPARK involving loop invariant code motion using our method. Experimental results demonstrate the usefulness of our method.**

***Index Terms*—Formal Verification, Translation Validation, Equivalence Checking, Code Motion, FSMDs Model.**

I. Introduction

VERIFICATION of code motion transformations has been an active research area for the last ten years [1]–[8]. The methods [1]–[6] fail to handle the case of code motion across loops and loop invariant code motion in nested loops. The technique presented in [8] handles code motion across loops but it requires additional information from the synthesis tool which is difficult to obtain in general. A Value Propagation based equivalence checking (VP) method was proposed in [7] which also handles code motion across loops. Unlike the technique presented in [8], the VP method does not require additional information from the HLS tool. 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.

The VP method handles scenario S_1 but it cannot handle scenarios S_2 and S_3. In addition, Example 1 given in Sec. III shows a case where the VP method [7] provides a *false positive* result for a scenario involving code motion across loops. Moreover, the VP method does not check whether a computation is a false computation 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 authors are with the Department of Computer Science and Engineering, Indian Institute of Technology, Guwahati 781039, India (e-mail: r.chouksey@iitg.ac.in; ckarfa@iitg.ac.in; pbhaduri@iitg.ac.in;).

In this paper, we present an equivalence checking method based on value propagation for code motion involving loops to overcome all the above limitations of existing works. Our method is capable of handling all the three scenarios, i.e., S_1, S_2 and S_3, mentioned above. Moreover, our method is able to prove non-equivalence for the case given in Example 1. Also, if the loop is executed at least once, then our method will ignore the false computation during equivalence checking. In particular, a bug in the HLS tool SPARK [9] involving loop invariant code motion is detected by our method.

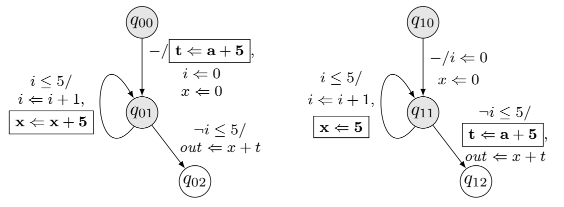
II. Value Propagation Based Equivalence of FSMDs

In this section, the FSMD model and the VP method presented in [7] are briefly explained. An FSMD M is defined as a 7-tuple $\langle Q, q_0, I, O, V, f, h\rangle$, where Q is the finite set of states, $q_0 \in Q$ is the reset (initial) state, I is the finite set of input variables, O is the finite set of output variables, V is the finite set of storage variables, $f : Q \times 2^S \rightarrow Q$ is the state transition function, $h : Q \times 2^S \rightarrow U$ is the update function. Here S represents the set of relations over arithmetic expressions and Boolean literals and U represents a set of storage and output assignments. An FSMD is an inherently deterministic model.

A *computation* of an FSMD is a finite walk from the reset state q_0 to itself, and q_0 should not occur in between. An FSMD may consist of an infinite number of computations because of the presence of loops. The paper [7] breaks down an FSMD into smaller segments by introducing cutpoints so that each loop in an FSMD is cut at at least one cutpoint. 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 *condition of execution* R_α of a path α is a logical expression over $I \cup V$, which must be satisfied by the initial data state in order to traverse the path α. The *data transformation* s_α of a path α is an updated variable vector. Two paths β and α are equivalent, denoted by $\beta \simeq \alpha$, if $R_\beta \equiv R_\alpha$ and $s_\beta = s_\alpha$.

An FSMD M_0 is contained in another FSMD M_1 ($M_0 \sqsubseteq M_1$), if there exists a path cover $P_0 = \{p_{00}, p_{01}, \cdots, p_{0k}\}$ of M_0 and $P_1 = \{p_{10}, p_{11}, \cdots, p_{1k}\}$ of M_1 such that $p_{0i} \simeq p_{1i}$ for all i, $0 \le i \le k$. Two FSMDs M_0 and M_1 are equivalent, denoted as $M_0 \equiv M_1$, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$.

The VP method of FSMDs [7] is based on propagating the values 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) M_0 (b) M_1

Fig. 1. An example where VP method gives false positive result.

a path β is an ordered pair $\langle R_\beta, s_\beta\rangle$, where the R_β is the condition of execution and the s_β is an updated variable vector representing the symbolic value[1] obtained by the variables at the end state of β. In Fig. 1(a), the propagated vector at the reset state q_{00} is $\vartheta_{00} = \langle \mathtt{T}, \langle a, i, out, x, t\rangle\rangle$ and the same at q_{01} is $\vartheta_{01} = \langle \mathtt{T}, \langle a, 0, out, 0, a+5\rangle\rangle$.

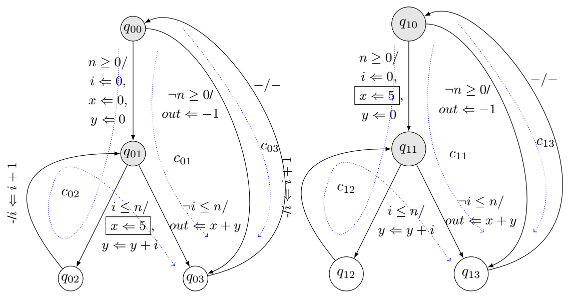
In the course of equivalence checking of two FSMDs, two paths, β and α say (one from each FSMD), are compared with respect to their corresponding propagated vectors for finding a path equivalence. If $R_\beta \equiv R_\alpha$ and $s_\beta = s_\alpha$, then these paths are declared as *unconditionally 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.

III. Motivational Examples

To detect valid code motion across a loop, the VP 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. A code motion across a loop is determined to be valid by the VP method iff (1) the values of marked variables are exactly the same after exiting the loop as before entering the loop in both behaviors and (2) the data transformations of unmarked variables, with respect to the propagated vector (stored before entering the loop) are exactly the same within the loop in both behaviors. It may be noted that after traversing the loop once, the VP method compares the unmarked variable values of each behavior. If the values are the same, then it declares that all the variables are identically defined. But this may not always be true as shown in Example 1. In a propagated vector, we use bold face to denote the marked variables.

Example 1. *For the path $q_{00} \Rightarrow q_{01}$ of M_0 in Fig. 1, the VP method finds the candidate C-equivalent path $q_{10} \Rightarrow q_{11}$ of M_1 since there is a mismatch in the values of t (i.e., t and a are marked variables). The propagated vectors at q_{01} and q_{11} are $\vartheta_{01} = \langle \mathtt{T}, \langle \mathbf{a}, 0, out, 0, \mathbf{a+5}\rangle\rangle$ and $\vartheta_{11} = \langle \mathtt{T}, \langle \mathbf{a}, 0, out, 0, \mathbf{t}\rangle\rangle$, respectively. After traversing the loop once, the propagated vectors at q_{01} and q_{11} will be $\vartheta'_{01} = \langle T, \langle \mathbf{a}, 1, out, 5, \mathbf{a+5}\rangle\rangle$ and $\vartheta'_{11} = \langle T, \langle \mathbf{a}, 1, out, 5, \mathbf{t}\rangle\rangle$ respectively. Here the marked variables t and a are not updated*

[1]The symbolic value of a variable x is 'x'.



(a) M_0 (b) M_1

Fig. 2. An example where the VP method provides false negative result.

```
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 ($i_n = L_n; i_n \leq H_n; i_n+ = r_n$)
      $S_n$ :   . . .
```

Fig. 3. Nested loop structure

in either of the loops (i.e., condition 1 is satisfied) and the unmarked variables x and i have the same transformation (the value of x is 5 and the value of i is 1) in both the loops (i.e., thus satisfy the condition 2) with respect to propagated vectors ϑ_{01} and ϑ_{11}. Therefore, the VP method says it is a valid case of code motion across a loop. Finally, $q_{01} \Rightarrow q_{02}$ and $q_{11} \Rightarrow q_{12}$ are designated as a U-equivalent, and the previously declared candidate C-equivalent path pairs are asserted to be C-equivalent. Hence, the VP method declares $M_0 \equiv M_1$. It may be noted that after exiting the loop the value of x at q_{01} will be 25 in M_0; while, it will be the value 5 at q_{11} in M_1. Clearly, these two behaviors are not equivalent. Hence, the VP method gives a false positive result in this case.

In the case of mismatch at the loop header, the VP method does not revert all the unmarked variables to their symbolic values and propagates their values along with the marked variables. It may cause the VP method to produce a false positive result in some scenarios as shown in Example 1. To avoid the false positive result the VP method should propagate only the marked variable values and all the unmarked variables should be reverted to their symbolic values. In Sec. IV-A, we propose an enhancement to address this issue. Example 2 below illustrates a case where the VP method provides false negative results due to the presence of a false computation.

Example 2. *The VP method in [7] does not check whether a computation is a false computation. It finds that the computation c_{02} and c_{03} of FSMD M_0 in Fig 2 are equivalent to the computation c_{12}, c_{13} of FSMD M_1, respectively. However, the VP method fails to find c_{11} as an equivalent computation of c_{01} in FSMD M_0, since they differ in the final value of the variable x. It may be noted that the loop will execute at least once for all possible $n \geq 0$ and $i = 0$. The computation c_{01} is, therefore, a false computation. The non-equivalence of FSMDs reported by the VP method is due to this false computation.*

$$\begin{tikzcd} \vartheta_{0i}: \langle\cdot\cdot x\cdot\cdot\rangle \arrow[d] & \vartheta_{1j}: \langle\cdot\cdot x\cdot\cdot\rangle \arrow[d] \\ \vartheta'_{0i}: \langle\cdot\cdot e_x\cdot\cdot\rangle & \vartheta'_{1j}: \langle\cdot\cdot e_x\cdot\cdot\rangle \end{tikzcd}$$

(a) Case 1.1

$$\begin{tikzcd} \vartheta_{0i}: \langle\cdot\cdot x\cdot\cdot\rangle \arrow[d] & \vartheta_{1j}: \langle\cdot\cdot x\cdot\cdot\rangle \arrow[d] \\ \vartheta'_{0i}: \langle\cdot\cdot e_{x_{0i}}\cdot\cdot\rangle & \vartheta'_{1j}: \langle\cdot\cdot e_{x_{1j}}\cdot\cdot\rangle \end{tikzcd}$$

(b) Case 1.2

$$\begin{tikzcd} \vartheta_{0i}: \langle\cdot\cdot x\cdot\cdot\rangle \arrow[d] & \vartheta_{1j}: \langle\cdot\cdot e_{x_{1j}}\cdot\cdot\rangle \arrow[d] \\ \vartheta'_{0i}: \langle\cdot\cdot e_{x_{1j}}\cdot\cdot\rangle & \vartheta'_{1j}: \langle\cdot\cdot e_{x_{1j}}\cdot\cdot\rangle \end{tikzcd}$$

(c) Case 2.1

$$\begin{tikzcd} \vartheta_{0i}: \langle\cdot\cdot e_{x_{0i}}\cdot\cdot\rangle \arrow[d] & \vartheta_{1j}: \langle\cdot\cdot e_{x_{1j}}\cdot\cdot\rangle \arrow[d] \\ \vartheta'_{0i}: \langle\cdot\cdot e_{x_{0i}}\cdot\cdot\rangle & \vartheta'_{1j}: \langle\cdot\cdot e_{x_{1j}}\cdot\cdot\rangle \end{tikzcd}$$

(d) Case 2.2

Fig. 4. 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

IV. Proposed Enhancements

We now propose solutions to prove the non-equivalence for the case given in Example 1 and to identify a false computation in an FSMD during equivalence checking. Further, we also provide a method to handle all the scenarios S_1, S_2 and S_3 during equivalence checking.

A. Showing the non-equivalence for false positive cases

The VP method propagates the values of live variables over the corresponding paths of the two behaviors as follows.

- If there is a mismatch in the propagated vector in a corresponding state pair, then it propagates not only the mismatched values (corresponding to marked variables), but also the matched values (corresponding to unmarked variables).
- If there is no mismatch in the propagated vector in a corresponding state pair then all variables are reverted back to their symbolic values.

In our method, we propagate the values of live variables over the corresponding paths of the two behaviors in the same way as mentioned above. *However, in case of a mismatch at the loop header, we propagate only the marked variable values and all the unmarked variables are reverted to their symbolic values.* This helps us to identify whether an unmarked variable is defined identically in both the loops. In Example 1, using this rule the propagated vector at q_{01} (via $q_{00} \Rightarrow q_{01}$ path) is $\vartheta_{01} = \langle \mathtt{T}, \langle \mathbf{a}, i, out, x, \mathbf{a+5}\rangle\rangle$ and the propagated vector at q_{11} is $\vartheta_{11} = \langle \mathtt{T}, \langle \mathbf{a}, i, out, x, \mathbf{t}\rangle\rangle$ (via $q_{10} \Rightarrow q_{11}$ path) before entering the loop. At the end of the loop the propagated vector at q_{01} will be $\vartheta'_{01} = \langle \mathtt{i} \le 5, \langle \mathbf{a}, i, out, x+5, \mathbf{a+5}\rangle\rangle$, and the propagated vector at q_{11} will be $\vartheta'_{11} = \langle \mathtt{i} \le 5, \langle \mathbf{a}, i, out, 5, \mathbf{t}\rangle\rangle$. The value for x (unmarked variable) is not the same in ϑ'_{01} and ϑ'_{11}. Hence, it is not a valid code motion and the two behaviors shown in Fig. 1 are not equivalent.

B. Handling False Computation Involving Loops

Let us consider the nested loop structure of depth n shown in Fig. 3. Each iterator i_x, $1 \le x \le n$, is initialized to L_x. Each iterator i_x reaches its upper limit H_x by incrementing a step constant r_x. The terms L_x and H_x, $x = 1, \ldots, n$, are assumed to be linear expressions over the input variables, constants or previous loop iterators $i_1 \cdots i_{x-1}$. These requirements on L_i, H_i, r_i and the increment statement restrict the kind of loops to which our method will apply. Conceptually, the propagated condition C_p in a state s is the condition of a path from the reset state of the behavior to the state s. In Fig. 2, for example, C_p is $n \ge 0$ at state q_{01}. 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.

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

where $f_x = \Big((L_x \le i_x \le H_x) \wedge (i_x = a_x r_x + L_x) \wedge (a_x \ge 0)\Big)$. Here C_p is the propagated condition before entering the nested loop of depth n. We use this formula to identify a false computation during equivalence checking. For checking the validity of this formula, we use the SMT solver Z3 [10] in the theory of linear integer arithmetic.

For example, in Fig. 2 to verify whether the loop $q_{01} \xRightarrow{i \le n} q_{01}$ will execute at least once, we should check the validity of the formula $n \ge 0 \implies 0 \le n$, which is valid. Thus, the loop will always execute at least once for all possible values of $n \ge 0$, and hence c_{01} is a false computation. By ignoring this false computation, our method shows the equivalence between the two behaviors shown in Fig. 2.

C. Handling Loop Invariant Code Motion

We consider marked and unmarked variables separately at the loop header to handle the scenarios S_2 and S_3. 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} then it indicates that x is defined identically in both the loops as shown in Fig. 4(a). 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. 4(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. The marked variables may have some mismatch in the corresponding propagated vectors ϑ_{0i} and ϑ_{1j}. 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. 4(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. However, this scenario is unlikely to occur in synthesis tools in practice.

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 loops as shown in Fig. 4(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.

V. Enhanced Value Propagation Based Equivalence Checking

In this section, we present our enhanced VP method (EVP). We use all the functions of the VP method as they are except the `correspondenceChecker` and `loopInvariant` functions. We have enhanced the correspondence checker so that our method can handle all the issues address in Sec. IV. The `loopInvariant` function is also enhanced to handle all the cases discussed in Sec. IV-C.

The behavior of the enhanced correspondence checker `ECC` function (Algorithm 1) is as follows. It takes as input a corresponding state pair [7] $\langle q_{0i}, q_{1j}\rangle$, a path covers P_0 (of M_0) and P_1 (of M_1), a corresponding state pair set W_{csp}, a set of U-equivalent path pairs E_u, a set C-equivalent path pairs E_c, and a *LIST* which maintains a candidate C-equivalent pairs of paths. It returns "*success*" if for every path emanating from q_{0i} an equivalent path originating from q_{1j} is found; otherwise, it returns "*failure*". 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. The function `checkFalseComputation` should be invoked once for all paths that terminate in the state q_{0i}. Moreover, a call to `checkFalseComputation` should be avoided if the state q_{0i} is reached through some back edge. To guarantee this, each loop header is associated with a flag *doLoopTest*. At each loop

Algorithm 1: $\texttt{ECC}(q_{0i}, q_{1j}, P_0, P_1, W_{csp}, E_u, E_c, LIST)$

```
/* If $q_{0i}$ is a loop header, then the paths from $q_{0i}$ are ordered such that
   the path exiting the loop body is considered first */
1  if $q_{0i}$ is a loop header and doLoopTest[$q_{0i}$] is TRUE then
2      doLoopTest[$q_{0i}$]=FALSE;
3      if checkFalseComputation($q_{0i}$) returns True then
4          avoidLoopExitPath[$q_{0i}$]=TRUE; /* Ignore False Computation */
5      end if
6  end if
7  foreach path $\beta : (q_{0i} \Rightarrow q_{0m})$ in $P_0$ do
8      if $q_{0i}$ is a loop header and avoidLoopExitPath[$q_{0i}$] is TRUE then
9          avoidLoopExitPath[$q_{0i}$]=FALSE;
10         continue;
11     end if
12     if Path $\beta$ is already present in the $LIST$ then
13         continue; /* prevent recursions which lead to an infinite loop */
14     end if
15     $(\beta, \alpha, \vartheta'_{0m}, \vartheta'_{\vartheta_{1n}}) \leftarrow$
         findEquivalentPath($\beta, \vartheta_{0i}, q_{1j}, \vartheta_{1j}, P_0, P_1$);
16     if path $\alpha : (q_{1j} \Rightarrow q_{1n})$ can be found in $P_1$ such that $\beta \simeq \alpha$ then
17         $E_u = E_u \cup \{(\beta, \alpha)\}$;                /* U-equivalence */
18         $W_{csp} = W_{csp} \cup \{(q_{0m}, q_{1n})\}$;
19     else if path $\alpha : (q_{1j} \Rightarrow q_{1n})$ can be found in $P_1$ such that $\beta \simeq_c \alpha$ then
20         if $q_{0m}$ or $q_{1n}$ is reset state then
21             return failure;    /* Reset state is reached with unresolved
                 mismatch */
22         else if $q_{0m}$ or $q_{1n}$ appears as the final state of some path already in
             $LIST \wedge$ loopInvariant($\beta, \alpha, \vartheta'_{0m}, \vartheta'_{1n}$) then
23             return failure;  /* Propagated values are not loop invariant
                 */
24         else
25             $\vartheta_{0m} \leftarrow \vartheta'_{0m}$; $\vartheta_{1n} \leftarrow \vartheta'_{1n}$;
26             Append $\langle \beta, \alpha \rangle$ to $LIST$
27             ECC($q_{0m}$,$q_{1n}$, $P_0, P_1, W_{csp}, E_u, E_c, LIST$);
28         end if
29     else
30         return failure;    /* Equivalent Path of $\beta$ may not be present in
             $P_1$ */
31     end if
32 end foreach
33 $E_c = E_c \cup$ {Last member of $LIST$};
34 $LIST \leftarrow LIST \backslash$ {Last member of $LIST$};
35 if $q_{0i}$ is a loop header then
36     doLoopTest[$q_{0i}$]=TRUE;
37 end if
38 return success;
```

header state q_{0i}, we also associated a flag *avoidLoopExitPath*. This flag is used to ensure that after avoiding the loop exit path once the loop exit path must be checked for subsequent calls of the function `ECC` for the state q_{0i}.

The function `ECC` invokes the function `findEquivalentPath` to find a U- or C-equivalent path $\alpha : (q_{1j} \Rightarrow q_{1n})$ in the transformed FSMD M_1 for each path $\beta : (q_{0i} \Rightarrow q_{0m})$ starting from state q_{0i} of the original FSMD M_0. The function `findEquivalentPath` returns a 4-tuple $\langle \beta, \alpha, \vartheta'_{0m}, \vartheta'_{1n}\rangle$ where β and α are corresponding paths as described above, ϑ'_{0m} is the propagated vector at the end state q_{0m} of β and ϑ'_{1n} is the propagated vector at the end state q_{1n} of α. If $\vartheta'_{0m} \equiv \vartheta'_{1n}$ then the path α is U-equivalent to path β. Consequently, the data structure W_{scp} gets updated (line 18). If `findEquivalentPath` does not find any path α in M_1 whose condition of execution R_α satisfies either $R_\beta \equiv R_\alpha$, or $R_\beta \implies R_\alpha$ or $R_\alpha \implies R_\beta$, then it returns $\alpha = \texttt{NULL}$ (i.e., M_0 and M_1 may not be equivalent, handled in line 30). If $\alpha \neq \texttt{NULL}$ and $\vartheta'_{0m} \not\equiv \vartheta'_{1n}$, then the path α is candidate C-equivalent to the path β and hence further value propagation is required. However, the following checks are carried out first and `ECC` reports "*failure*" in the following scenarios:

TABLE I
EXPERIMENTAL RESULTS ON THE BENCHMARKS PRESENTED IN [7]

Benchmarks	M_0		M_1		#Loop	VP	EVP
	#State	#Path	#State	#Path		Time(ms)	Time(ms)
PERFECT	6	7	4	6	1	24	40
GCD	8	11	14	8	1	56	116
MODN	8	9	9	9	1	92	176
LRU	33	39	32	39	8	364	1204
IEEE754	55	59	44	50	7	540	2080
BARCODE	32	55	24	57	15	482	3503

TABLE II
EXPERIMENTAL RESULTS ON TEST CASES WHERE THE VP METHOD FAILS

Benchmarks	VP		EVP	
	Equivalent	Time (ms)	Equivalent	Time (ms)
simple_types_ loop_invariant	No	4	Yes	12
mandel	No	4	Yes	16
mandel2	No	4	Yes	16
himenobmtxpa	No	4	Yes	20
Test 1	Yes	8	No	8
Test 2	Yes	8	No	8
Test 3	Yes	12	No	12
Test 4	Yes	16	No	16

1) if one of the state q_{0m} and q_{1n} is a reset state (line 21) it returns "*failure*";
2) if a loop has been crossed over then the function ECC invokes the function `loopInvariant`. The function `loopInvariant` checks for the loop invariance of the propagated vector ϑ'_{0m} and ϑ'_{1m}. The function `loopInvariant` returns `True` if each marked and unmarked variables satisfy their respective cases as mentioned in Sec. IV-C. If it returns `False` then the function ECC returns "*failure*".

If $\vartheta'_{0m} \not\equiv \vartheta'_{1n}$ and the above two cases do not occur, then $\langle \beta, \alpha \rangle$ is appended to $LIST$ and the propagated vector at q_{0m} and q_{1n}, are updated and ECC calls itself recursively (line 27). It may be noted that while updating the propagated vector (line 25), we update only mismatched variable values and reset the other variable to their symbolic values if the state is the loop header; otherwise, we update all the variable values. When ECC reaches line 38, it implies that for every chain of paths emanating from the state q_{0i}, there exists a corresponding chain of paths emanating from q_{1j} such that their final paths are U-equivalent.

VI. EXPERIMENTAL RESULTS

Our equivalence checking algorithm has been implemented in C and all the experiments have been conducted on a laptop with 2 GHz Intel Core 2 Duo processor with 3 GB of RAM. In our first experiment all the benchmarks listed in Table I are taken from [7]. Our method is able to establish the equivalence in all the benchmarks. Our method needs more time since at each loop header we invoke the SMT solver Z3 to identify false computations. In our second experiment, we take some of the test-suite distributed with LLVM [11]. These benchmarks contain some loop invariant operations. We forced SPARK to apply loop invariant code motion (LICM) transformation

```
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

Fig. 5. A bug in SPARK

to obtain the transformed behavior. The results of these experiments are tabulated in row 1–4 of Table II. It is evident from Table II, that our proposed method can correctly identify the equivalences. However, the VP method reports "may not be equivalent" in these cases. In our third experiment, we have created some test cases where the VP method provides a *false positive* result, but our EVP method can prove the non-equivalence. The benchmarks tabulated in row 5–8 of Table II are manually scheduled. The result of this experiment confirms that the VP method incorrectly reports equivalence for these test cases while our EVP method correctly proves the non-equivalence for these test cases. During our experimentation, we found a bug in the implementation of the LICM algorithm in the SPARK tool as shown in Fig. 5. 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.

REFERENCES

[1] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar, "An equivalence-checking method for scheduling verification in high-level synthesis," *IEEE TCAD*, vol. 27, no. 3, pp. 556–569, 2008.

[2] S. Kundu, S. Lerner, and R. K. Gupta, "Translation validation of high-level synthesis," *IEEE TCAD*, vol. 29, no. 4, pp. 566–579, 2010.

[3] T. Li, J. Hu, Y. Guo, S. Li, and Q. Tan, "Equivalence checking of scheduling in high-level synthesis," in *Sixteenth International Symposium on Quality Electronic Design*. IEEE, 2015, pp. 257–262.

[4] Y. Kim and N. Mansouri, "Automated formal verification of scheduling with speculative code motions," in *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*. ACM, 2008, pp. 95–100.

[5] C. Lee, C. Shih, J. Huang, and J. Jou, "Equivalence checking of scheduling with speculative code transformations in high-level synthesis," in *Proceedings, ASP-DAC'11*. IEEE, 2011, pp. 497–502.

[6] C. Karfa, C. A. Mandal, and D. Sarkar, "Formal verification of code motion techniques using data-flow-driven equivalence checking," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 3, pp. 30:1–30:37, 2012.

[7] K. Banerjee, C. Karfa, D. Sarkar, and C. A. Mandal, "Verification of code motion techniques using value propagation," *IEEE TCAD*, vol. 33, no. 8, pp. 1180–1193, 2014.

[8] J.-B. Tristan and X. Leroy, "Verified validation of lazy code motion," in *Proceedings, PLDI '09*. ACM, 2009, pp. 316–326.

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

[10] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS'08*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on CGO*. IEEE, 2004, pp. 129–142.