Translation Validation of Loop Invariant Code Optimizations Involving False Computations

Ramanuj Chouksey, Chandan Karfa, and Purandar Bhaduri

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati 781039, India
{r.chouksey,ckarfa,pbhaduri}@iitg.ernet.in

Abstract. Code motion based optimizations are used quite often in electronic design automation (EDA) tools to improve the quality of synthesis results. Ensuring the correctness of such transformation is necessary for reliability of EDA tools. A value propagation (VP) based equivalence checking method of finite state machine with datapaths (FSMD) was proposed in [1] to specifically verify code motion across loops. In this work, we identify some scenarios involving loop invariant code motion where the VP based equivalence checking method fails to establish the equivalence between two actually equivalent FSMDs. We propose an enhancement over the VP based equivalence checking method [1] to overcome this limitation. Experimental results demonstrate that our method can handle the scenario where the VP based equivalence checking method fails.

Keywords: Formal Verification, Translation Validation, Code Motion, Equivalence Checking, Loop Invariant, FSMDs Model.

1 Introduction

Code motion based transformations move operations across the boundaries of basic blocks [6]. They are widely used to improve the quality of synthesis results for designs with complex and nested conditionals and loops. The objectives of code motion are reducing the number of computations at run time and improving register utilization by reducing the lifetime of temporary variables. Code motion techniques [10, 17, 6] change the data-flow of a program considerably. Therefore, it is necessary to verify the semantic equivalence between the original and the transformed program.

The bisimulation approach presented in [11, 12] has been applied successfully to verify structure-preserving code motions. This method fails when the control structure of input behavior is modified by a path based scheduler [2, 16]. To overcome this limitation, a path extension based equivalence checking method of FSMDs was first proposed in [9]. This work was later enhanced in [8, 13] to handle uniform and non-uniform code motion based optimization techniques. All these methods fail to handle the case of code motion across loops, and loop

invariant code motion in nested loops, since a path can't be extended beyond a loop by the definition of a path cover. A VP based equivalence checking method was proposed in [1], which can additionally handle code motion across loops.

In this paper, we identify some limitations of the VP based equivalence checking method [1]. Specifically, we show that the VP based equivalence checking method gives *false negative* results when some loop invariant operation `op` is moved before the loop from inside it, some operation after the loop depends on `op` and there is a guarantee the loop will execute at least once. As a result of the transformation, a false computation will arise. A computation of an FSMD is called false computation if it never executes. Since the method in [1] cannot identify false computations, it reports a possible non-equivalence of FSMDs (which are actually equivalent). We propose an enhancement to the value propagation based equivalence checking method to handle the above scenario. In particular, at the loop header, we automatically extract a formula that checks whether a loop will always execute at least once under a propagated condition. We check the validity of this formula using the SMT solver Z3 [18] in the theory of linear integer arithmetic. This formula will guide the VP based method during equivalence checking to identify and ignore false computations. More importantly, our method can handle any level of loop nesting. We implement the enhanced VP based equivalence checking method. We identify various test cases where VP based equivalence checking method fails to establish the equivalence, but our method is able to show the equivalence. Experimental results demonstrate the contribution of the paper.

The rest of this paper is organized as follows. A scenario where VP based equivalence checking method presented in [1] gives a false negative result is illustrated in Section 2. The FSMD model and the VP based equivalence checking method are briefly explained in Section 3. A solution to identify a false computation of an FSMD during equivalence checking is presented in Section 4. The enhanced VP based equivalence checking method is presented in Section 5. Experimental results are given in Section 6. Section 7 concludes the paper.

2 Motivational Example

Loop invariant code inside a loop body consists of statements or expressions which produce the same result each time the loop is executed. In other words, these statements are not dependent on loop iterations. This code can be moved outside the loop body without changing the program semantics. Loop invariant code motion improves overall program execution time by reducing the number of times loop invariant expressions are executed by a factor equal to the loop size.

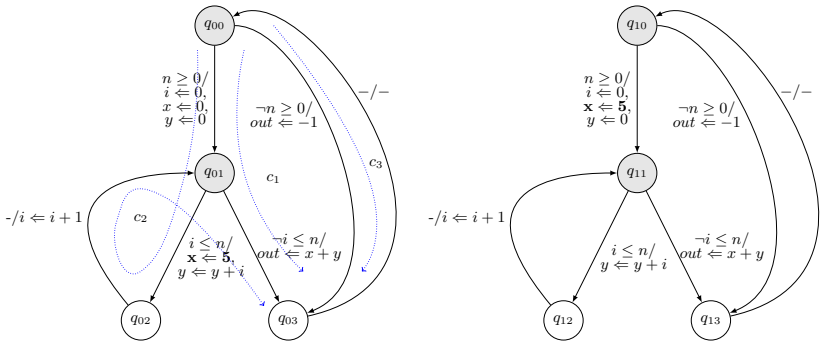
Let us consider the behavior in Fig. 1 and its corresponding FSMDs in Fig. 2. In this example, the operation $x \Leftarrow 5$ is a loop invariant for FSMD M_0 in Fig. 2(a). It is placed out of the loop in the transformed FSMD M_1 in Fig. 2(b). There are three possible computations, $c_1 = \langle q_{00} \xrightarrow{n \geq 0} q_{01} \xrightarrow{\neg i \leq n} q_{03} \Rightarrow q_{00} \rangle$, $c_2 = \langle q_{00} \xrightarrow{n \geq 0} (q_{01} \xrightarrow{i \leq n} q_{01})^{+} \xrightarrow{\neg i \leq n} q_{03} \Rightarrow q_{00} \rangle$ and $c_3 = \langle q_{00} \xrightarrow{\neg n \geq 0} q_{03} \Rightarrow q_{00} \rangle$

```
if(n≥0){
  x=0,y=0;
  for(i=0;i≤n;i++){
   x=5;
   y=y+i;}
  out=x+y;}
else
  out=-1;
```

```
if(n≥0){
  x=5,y=0;
  for(i=0;i≤n;i++){
   y=y+i;}
  out=x+y;}
else
  out=-1;
```

Fig. 1: Loop-invariant code motion

for the FSMD in Fig. 2(a). The computation c_1 executes if the loop condition $i \leq n$ is `False` for $n \geq 0$. The computation c_2 executes if the loop condition $i \leq n$ is `true` for the input $n \geq 0$. The computation c_3 executes for $n < 0$. In this example, when the state q_{01} is reached for the first time, n is always greater than or equal to 0 and i is equal to 0. Therefore, the computation c_1 will never execute. In other words, the loop will *execute at least once* for all possible $n \geq 0$ and $i = 0$. The computation c_1 is, therefore, a *false computation*. The VP



(a) M_0 (b) M_1

Fig. 2: The FSMDs of the behaviors in Fig. 1

based equivalence method in [1] explores all possible computations of a given FSMD M_0. It does not check whether a computation is a false computation or not. During equivalence checking of these two programs, an existing tool will try to prove that these two behaviors are equivalent for all possible computations. Thus, the equivalence checker will try to find the equivalence of computations c_1, c_2 and c_3 in the other FSMD M_1. It finds that the computation c_2 and c_3

of FSMD M_0 are equivalent to the computation $\langle q_{10} \overset{n\geq 0}{\Longrightarrow} (q_{11} \overset{i\leq n}{\Longrightarrow} q_{11})^+ \overset{\neg i\leq n}{\Longrightarrow} q_{13} \Rightarrow q_{10}\rangle$, $\langle q_{10} \overset{\neg n\geq 0}{\Longrightarrow} q_{13} \Rightarrow q_{10}\rangle$ of FSMD M_1, respectively. However the equivalence checking method finds that the computation c_1 of FSMD M_0 is not equivalent to the computation $\langle q_{10} \overset{n\geq 0}{\Longrightarrow} q_{11} \overset{\neg i\leq n}{\Longrightarrow} q_{13} \Rightarrow q_{10}\rangle$ of FSMD M_1, since they differ in the final value of the variable x. It may be noted that the final value of x would be 0 after execution of c_1 in M_0 and 5 after the execution of $\langle q_{10} \overset{n\geq 0}{\Longrightarrow} q_{11} \overset{\neg i\leq n}{\Longrightarrow} q_{13} \Rightarrow q_{10}\rangle$ in M_1. In this example, as described above, the computation c_1 will never execute. The non-equivalence of FSMDs reported by this equivalence checking method is due to this false computation. If we can ignore this false computation during equivalence checking, we can establish the equivalence between these two behaviors. Most of the existing state-of-the-art equivalence checking methods fail for such scenario.

In this work, we have enhanced the equivalence checking method reported in [1] to handle the above situation. During equivalence checking, our method will automatically identify the false computation by checking whether a loop always executes or not. If the loop is executed at least once, then our method will ignore the false computation during equivalence checking. Our method is strong enough to handle any nested loops as well.

3 Value Propagation Based Equivalence of FSMDs

In this section, the FSMD model and the VP based equivalence checking method presented in [1] are briefly explained. The details can be found in [1].

3.1 FSMD Model

FSMDs [5] are an extension of the finite state machine (FSM) model with data/variables used to model behaviors. Unlike FSMs that model the control flow, FSMDs capture the data-flow aspect of the behavior as well. Each transition of an FSMD includes a condition over the data variables and a set of operations transform the variable values.

Definition 1 (FSMD). *A 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 expression and boolean literals and U represents a set of storage and output assignments. An FSMD is an inherently deterministic model.

A *walk* from q_i to q_j is a sequence of state transitions of the form $\langle q_i \xrightarrow[c_i]{} q_{i+1} \xrightarrow[c_{i+1}]{} \cdots \xrightarrow[c_{i+n-1}]{} q_{i+n} = q_j \rangle$ where $q_k \in Q$ for all k, $i \leq k \leq i+n$, and $\exists c_k \in 2^S$ such that $f_k(q_k, c_k) = q_{k+1}$ for all k, $i \leq k \leq i+n-1$. A *(finite) path* α is a walk where all the states are different, except the end state q_j may be the same as the start state q_i. The *condition of execution* R_α of a path α is a logical expression over $I \cup V$, which must be satisfied by initial data state in order to traverse the path α. The *data transformation* r_α of a path α is an ordered pair $\langle s_\alpha, O_\alpha \rangle$, where s_α is an updated variable vector and O_α is an updated output list after executing α. Thus R_α and r_α are the weakest precondition of the path α [3]. For a path α, R_α and r_α are computed by forward or backward substitution based on symbolic execution.

3.2 Equivalence of FSMDs

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. For an FSMD M, any computation μ is the concatenation $[\alpha_1 \alpha_2 \cdots \alpha_n]$ of paths of M where for all k, $1 \leq k < n$, α_k terminates in the start state of the path α_{k+1}, q_0 is the start state of α_1 and the end state of α_n. Two paths β and α are equivalent, denoted as $\beta \simeq \alpha$ if $R_\beta \equiv R_\alpha$ and $r_\beta = r_\alpha$. The equivalence of two computations can be defined in a similar fashion. The definition of path cover is as follows.

Definition 2 (Path cover of an FSMD [9]). *A finite set of paths $P = \{p_0, p_1, \ldots, p_k\}$ is said to be a path cover of an FSMD M if any computation μ of M can be looked upon as a concatenation of paths from P.*

To obtain a path cover, the paper [1] breaks down an FSMD into smaller segments by introducing cut-points so that each loop in an FSMD is cut in at least one cutpoint. This is based on the Flyod–Hoare method of program verification [4, 7]. The set of all paths from a cutpoint to another cutpoint without any intermediary occurrences of cutpoint is a path cover of the FSMD. The reset state and all the branching states (state with more than one outgoing transition) are cutpoints of an FSMD.

Let $M_0 = \langle Q_0, q_{00}, I, O, V_0, f_0, h_0 \rangle$ and $M_1 = \langle Q_1, q_{10}, I, O, V_1, f_1, h_1 \rangle$ be two FSMDs having same input(s)/output(s). The correspondence of states and equivalence between M_0 and M_1 are defined as follow.

Definition 3 (Corresponding States [9]).

1. *The reset states q_{00} and q_{10} corresponding states.*
2. *The states $q_{0k} \in Q_0$ and $q_{1l} \in Q_1$ are corresponding states if the state $q_{0i} \in Q_0$ and $q_{1j} \in Q_1$ are corresponding states and there exists paths, β from q_{0i} to q_{0k} and α from q_{1j} to q_{1l}, such that $\beta \simeq \alpha$.*

Theorem 1 ([9]). *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, $0 \leq i \leq k$.*

Two FSMDs M_0 and M_1 are equivalent, denoted as $M_0 \equiv M_1$, if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$. Since FSMDs are deterministic. It can be shown that $M_0 \sqsubseteq M_1$ implies $M_1 \sqsubseteq M_0$.

3.3 VP Based Equivalence Checking

The VP based equivalence checking method of FSMDs [1] 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. 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 these paths are equal, then they are declared as unconditionally equivalent (U-equivalent, represented as $\beta \simeq \alpha$). If some mismatch in data transformation is detected then they are declared to be conditionally equivalent (C-equivalent, represented as $\beta \simeq_c \alpha$) 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.

An abstract version of the VP based equivalence checking scheme is given in Algorithm 1. The details can be found in [1]. The function `containmentChecker`

Algorithm 1: `containmentChecker`(FSMD M_0, FSMD M_1)

```
1 Identify the cutpoints in M0 and M1 and compute their path cover P0 and P1;
  Wcsp is a set of corresponding state pairs and initially contains (q00, q10);
2 foreach (q0i, q1j) ∈ Wcsp do
3 |   if correspondenceChecker(q0i, q1j, P0, P1, Wcsp) returns "Failure" then
4 |   |   Report "unable to decide M0 ⊑ M1" and exit;
5 |   end if
6 end foreach
7 Report "M0 ⊑ M1";
```

(Algorithm 1) identifies the cutpoints and a path cover in an FSMD. It invokes `correspondenceChecker` (Algorithm 2) for each corresponding state pairs, one by one. The `correspondenceChecker` function checks whether for every path emanating from a state in the pair, there is a U- or C-equivalent path from the other member of the pair. Based on the output returned by `correspondenceChecker`, `containmentChecker` reports whether the initial FSMD is contained in the transformed FSMD or not.

4 Proposed Enhancement

As described in Section 2, we can establish the equivalence between two behaviors shown in Fig. 2 by ignoring the false computation c_2 during equivalence checking.

Algorithm 2: `correspondenceChecker`$(q_{0i}, q_{1j}, P_0, P_1, W_{csp})$

```
1  foreach path β : (q_0i ⇒ q_0m) in P_0 do
2  |  if path α : (q_1j ⇒ q_1n) can be found in P_1 such that β ≃ α then
3  |  |  W_csp = W_csp ∪ {(q_0m, q_1n)};
4  |  else if path α : (q_1j ⇒ q_1n) can be found P_1 such that β ≃_c α then
5  |  |  if q_0m or q_1n is reset state then
6  |  |  |  return failure;
7  |  |  else
8  |  |  |  correspondenceChecker(q_0m, q_1n, P_0, P_1, W_csp);
9  |  |  end if
10 |  else
11 |  |  return failure;
12 |  end if
13 end foreach
14 if any path of P_1 exists which does not pair with a path of P_0 then
15 |  return failure;
16 else
17 |  return success;
18 end if
```

In this section, we propose a solution to identify a false computation in an FSMD during equivalence checking. Let us consider the generalized nested loop structure of depth n as shown in Fig. 3 for this purpose. Each iterator i_x, $1 \leq x \leq n$, 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 expressions over the

```
for(i1=L1;i1≤H1;i1+=r1)
 for(i2=L2;i2≤H2;i2+=r2)
                ⋮
  for(in=Ln;in≤Hn;in+=rn)
          Sn: ...
```

Fig. 3: Generalized nested loop structure

input variables, constants or previous loop iterators $i_1 \cdots i_{x-1}$. Let us assume that C_p is a propagated condition at the start of the nested loop structure. Conceptually, the propagated condition 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, the C_p is $n \geq 0$ at state q_{01}. We will elaborate on the propagated condition once we discuss the enhanced value propagation method. Under the condition C_p, the

initial value of the loop iterator ($i_1 = L_1$) must satisfy the initial loop condition (i.e., $L_1 \leq H_1$) to execute a nested loop structure at least once. We can specify this condition by the following formula 1. If formula 1 is valid then a nested loop structure with nesting depth 1 will always execute at least once.

$$C_p \implies L_1 \leq H_1 \tag{1}$$

In other words, if the formula 1 is valid then the outer most loop of the nested loop structure will always execute at least once. The formula 2 is generalized form of the formula 1. If formula 2 is valid then the statement S_n at the generalized loop structure of nesting depth n, will always execute at least once.

$$C_p \implies \left(\exists i_1, \exists i_2, \cdots, \exists i_{n-1}, \exists a_1, \exists a_2, \cdots, \exists a_{n-1} \Big(\big(L_n \leq H_n\big) \wedge \Big(\bigwedge_{x=1}^{n-1} f_x \Big) \Big) \right) \tag{2}$$

where $f_x = \Big((L_x \leq i_x \leq H_x) \wedge (i_x = a_x r_x + L_x) \wedge (a_x \geq 0) \Big)$. The C_p is the propagated condition before entering the nested loop of depth n. We use these formulas to identify the false computation as mentioned in Section 2 during equivalence checking. For checking the validity of these formulas, we use the SMT solver Z3 [18] in the theory of linear integer arithmetic. These formulas can be automatically generated in our equivalence checking framework. For example, in Fig. 2 there is a loop $q_{01} \overset{i \leq n}{\Longrightarrow} q_{01}$ of nesting depth 1. At state q_{01} of FSMD M_0, the propagation condition C_p is $n \geq 0$. To verify whether the loop $q_{01} \overset{i \leq n}{\Longrightarrow} q_{01}$ will execute at least once, we should check the validity of the formula $n \geq 0 \implies 0 \leq n$. This formula is valid. Thus, the loop will always execute at least once for all possible values of $n \geq 0$. We can say that the computation $\langle q_{00} \overset{n \geq 0}{\Longrightarrow} q_{01} \overset{\neg i \leq n}{\Longrightarrow} q_{03} \Rightarrow q_{00} \rangle$ is a false computation. During equivalence checking, our method will ignore this false computation. By ignoring this false computation, we can show the equivalence between the two behaviors shown in Fig. 2.

5 Enhanced Value Propagation (EVP) Based Equivalence of FSMDs

In this section, we present the enhanced VP based equivalence checking method. The existing method gives *false negative* results when some loop invariant operation `op` is moved before the loop from inside it, some operation after the loop depends on `op` and there is a guarantee the loop will execute at least once. Let assume that we are at $\langle q_{0i}, q_{1j} \rangle$, a corresponding state pair, during equivalence checking and q_{0i} is a loop header. To handle this type of scenario, we should first check whether a loop starting at state q_{0i} with propagated condition $C_{q_{0i}}$ will execute at least once or not, over all possible inputs in an FSMD. This can be checked by generating the formula 2 at q_{0i}, say $f_{q_{0i}}$, as discussed in Section 4. If the formula $f_{q_{0i}}$ is valid under the propagated condition $C_{q_{0i}}$, then there is a

guarantee that the loop at the state q_{0i} will execute at least once. In this case, any computation till q_{0i} will be always followed by the loop body for some finite number of iterations and then take the exit path of the loop. This is ensured during equivalence checking by:

1. ensuring the path from q_{0i} representing the loop body is considered first for equivalence checking.
2. updating the propagated vector at q_{0i} with the propagated vector after the execution of the loop provided we have already found the U- or C-equivalent path from q_{1j} for the path starting from q_{0i} representing the loop body and propagated vector is loop invariant.

The enhanced version of Algorithm 2 incorporating this idea is presented as Algorithm 3. The point 2 above is reflected by lines 4–6 (for U-equivalence) and

Algorithm 3: `enhancedCorrespondenceChecker`$(q_{0i}, q_{1j}, P_0, P_1, W_{csp})$

```
   /* If q_0i is a loop header, then the paths from q_0i are ordered such
      that the paths corresponding to the loop body are considered
      first                                                          */
 1 foreach path β : (q_0i ⇒ q_0m) in P_0 do
 2     if path α : (q_1j ⇒ q_1n) can be found in P_1 such that β ≃ α then
 3         W_csp = W_csp ∪ {(q_0m, q_1n)}
 4         if q_0i is a loop header and loopTest(q_0i) returns valid and R_β is
           equivalent to loop condition at q_0i then
 5             updatePropagatedVector(q_0i);
 6         end if
 7     else if path α : (q_1j ⇒ q_1n) can be found in P_1 such that β ≃_c α then
 8         if q_1m or q_1n is reset state then
 9             return failure;
10         else
11             enhancedCorrespondenceChecker(q_0m, q_1n, P_0, P_1, W_csp);
12             if q_0i is a loop header and loopTest(q_0i) returns valid and R_β is
               equivalent to loop condition at q_0i then
13                 updatePropagatedVector(q_0i);
14             end if
15         end if
16     else
17         return failure;
18     end if
19 end foreach
20 if any path of P_1 exists which does not pair with a path of P_0 then
21     return failure;
22 else
23     return success;
24 end if
```

by line 12–14 (for C-equivalence). By updating the propagated vectors, we are effectively ensuring that the loop exit path is always preceded by the loop body. This is inherently a guarantee that any computation till q_{0i} is always followed by the loop body and the exit path. Thus, computation c_1 as described in Section 2 will never be checked during equivalence checking. In Algorithm 3, the `loopTest` function uses the SMT solver Z3 to check the validity of the formula $f_{q_{0i}}$. The `updatePropagatedVector` function updates the propagated vector at the loop header q_{0i} with the propagated vector after the loop.

6 Experimental Results

In our experimental setup, we replaced the existing `correspondenceChecker` function with the `enhancedCorrespondenceChecker` function. We have tested our implementation on several benchmarks presented in [1, 9]. We assume that loop information like nesting depth and loop header are available along with the test cases. This information can easily be obtained during extraction of the FSMD from the input behavior using dominator tree analysis [15, 14]. The test cases are run on a laptop with Intel core 2 Duo processor with 2 GHz and 3GB of RAM. The results of the experiments are shown in Table 1 and 2. The

Table 1: Experimental results on the benchmarks presented in [1, 9]

Benchmarks	#loops	VP		EVP	
		Equivalent	Time (in ms)	Equivalent	Time (in ms)
ASSORT	2	Yes	84	Yes	96
DIFFEQ	1	Yes	24	Yes	24
MODN	1	Yes	28	Yes	28
PERFECT	1	Yes	20	Yes	20
QRS	0	Yes	232	Yes	232
TLC	0	Yes	60	Yes	60
ASSORT-1	2	No	32	No	32
DIFFEQ-1	1	No	100	No	136
MODN-1	1	No	40	No	40
PERFECT-1	1	No	32	No	32
QRS-1	0	No	220	No	220
TLC-1	0	No	48	No	48

comparison of the execution time required by the VP based equivalence checking method [1] and our EVP method for the benchmarks are tabulated in Table 1. The second column "#loops" represents the number of loops in the behavior. For each benchmark, we have recorded the equivalence result and runtime in milliseconds (ms) obtained by executing these benchmarks in both the tools. Rows 1–6 represent the equivalent scenarios. Both tools are able to establish the

Table 2: Experimental results on test cases with loop invariant code motion

Benchmarks	#loops	VP		EVP	
		Equivalent	Time (in ms)	Equivalent	Time (in ms)
Test 1	1	No	4	Yes	16
Test 2	1	No	8	Yes	16
Test 3	2	No	16	Yes	20
Test 4	2	No	16	Yes	20
Test 5	2	No	16	Yes	16

equivalence in all these scenarios. We manually introduce some changes in the benchmarks listed in rows 1–6 so that their original and transformed FSMDs become inequivalent. These modified benchmarks are listed in rows 7–12. Again both the tools reported non-equivalence in all these scenarios. If a benchmark has no loop then the execution time obtained by our method is the same as the existing method. When a benchmark has some loop then our method needs a little extra time since at each loop header we invoke the SMT solver Z3 to check whether the loop will execute at least once.

Table 2 presents some test cases where the VP based method fails to establish the equivalence, but our EVP based method is able to show the equivalence. All the test cases were created manually by us. In all these test cases, some loop invariant operation `op` is moved before the loop from inside it, some operation after the loop depends on `op` and there is a guarantee the loop will execute at least once. It is evident from this table that our method outperforms the existing method. The results in Table 1 and 2 show that our method can handle all the scenarios which can be handled by the VP based equivalence checking method. It can additionally handle the scenarios mentioned in this paper where the existing method gives false negative results.

7 Conclusion

In this paper we have presented an equivalence checking method to verify loop invariant code transformations. This work is an enhancement to the VP based equivalence checking method presented in [1]. Like the VP based equivalence checking method, our method is also capable of handling control structure modification of input behavior and uniform and non-uniform code motion and code motion across loops. In addition, our method can also handle loop invariant code motions. At a loop header, our method automatically extracts the formula that encodes that a loop executes at least once. It then, invokes SMT solver Z3 to check the validity of the formula. If the formula is valid, our method ignores the false computation during equivalence checking. Thus, our method can prove the equivalence between two FSMDs for cases where the VP based equivalence checking method gives false-negative result. Experimental results demonstrate the advantage of our method over the VP based method.

References

1. Banerjee, K., Karfa, C., Sarkar, D., Mandal, C.A.: Verification of code motion techniques using value propagation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **33**(8), 1180–1193 (2014)
2. Camposano, R.: Path-based scheduling for synthesis. IEEE Trans. on CAD of Integrated Circuits and Systems **10**(1), 85–93 (1991)
3. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
4. Floyd, R.W.: Assigning meanings to programs. Mathematical aspects of computer science **19**(1), 19–32 (1967)
5. Gajski, D.D., Dutt, N.D., Wu, A.C.H., Lin, S.Y.L.: High-level Synthesis: Introduction to Chip and System Design. Kluwer Academic Publishers, Norwell, MA, USA (1992)
6. Gupta, S., Savoiu, N., Dutt, N.D., Gupta, R.K., Nicolau, A.: Using global code motions to improve the quality of results for high-level synthesis. IEEE Trans. on CAD of Integrated Circuits and Systems **23**(2), 302–312 (2004)
7. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
8. Karfa, C., Mandal, C.A., Sarkar, D.: Formal verification of code motion techniques using data-flow-driven equivalence checking. ACM Transactions on Design Automation of Electronic Systems (TODAES) **17**(3), 30 (2012)
9. Karfa, C., Sarkar, D., Mandal, C., Kumar, P.: An equivalence-checking method for scheduling verification in high-level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **27**(3), 556–569 (2008)
10. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation, PLDI '92, pp. 224–234. ACM, New York, USA (1992)
11. Kundu, S., Lerner, S., Gupta, R.: Validating high-level synthesis. In: Computer Aided Verification, 20th International Conference, CAV'00, pp. 459–472. Springer (2008)
12. Kundu, S., Lerner, S., Gupta, R.K.: Translation validation of high-level synthesis. IEEE Trans. on CAD of Integrated Circuits and Systems **29**(4), 566–579 (2010)
13. Lee, C., Shih, C., Huang, J., Jou, J.: Equivalence checking of scheduling with speculative code transformations in high-level synthesis. In: Proceedings of the 16th Asia South Pacific Design Automation Conference, ASP-DAC 2011, PLDI '92, pp. 497–502. IEEE, Yokohama, Japan (2011)
14. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems (TOPLAS) **1**(1), 121–141 (1979)
15. Lowry, E.S., Medlock, C.W.: Object code optimization. Commun. ACM **12**(1), 13–22 (1969)
16. Rahmouni, M., Jerraya, A.A.: Formulation and evaluation of scheduling techniques for control flow graphs. In: Proceedings of EURO-DAC. European Design Automation Conference, EURO-DAC'95, pp. 386–391. IEEE, England, UK (1995)
17. Rüthing, O., Knoop, J., Steffen, B.: Sparse code motion. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'00, pp. 170–183. ACM, New York, USA (2000)
18. Z3. https://github.com/Z3Prover/z3