SamaTulyataII: Translation Validation of Loop involving Code Optimizing Transformations using Petri Net based Models of Program

Rakshit Mittal[1], Rochishnu Banerjee[1], Santonu Sarkar[2], and Soumyadip Bandyopadhyay[1,3] *

[1] BITS Pilani, K K Birla Goa Campus, Goa, India,
[2] ABB Corporate Research INCRC, Bangalore, India
[3] Hasso Plattner Institute, Potsdam, Germany

Abstract. Translation validation is the process of proving semantic equivalence between source and source-translation, i.e., checking the semantic equivalence between the target code (which is a translation of the source program being compiled) and the source code. In this paper, we propose a translation validation technique for Petri net based models of programs which verify several code optimizing transformations involving loop. These types of transformation have been used in several application domains such as scheduling phase of High level synthesis, high performance computations etc. Our Petri net based equivalence checker checks the computational equivalence between two one-safe colour Petri nets. In this work, we have taken two versions of CPNs one corresponds to the source program and the other, the target programs. Using path based analysis technique, we have developed a sound method for proving several code optimizing transformations involving loop. We have also compared our results with other Petri net based equivalence checkers.The experimental result shows the efficacy of the method.

Keywords: Equivalence checking, CPN, Path based analysis, Translation validation.

1 Introduction

General applications when executed on parallel and embedded systems often go through a series of semantic preserving transformations. This is done so that the resulting translated program can optimally utilize the underlying computing architecture like multi-core and vector registers. There are various methods of code transformations like code motions, common sub-expression eliminations, dead code elimination, etc and several loop based transformation techniques such as loop distribution, loop parallelization, etc. These transformations are either carried out automatically by some compilers, or done semi-automatically/manually be design experts. Validation of a compiler, to ensure correctness of code-translation, by the construction property is a difficult task. Using, behavioral and semantic verification techniques, it is possible to verify whether the translated (optimized) program has the same functionality as that of the original code.

Hence, there is a need for proving behavioral equivalence between the original and the translated programs. This process of proving semantic equivalence between

source and source-translation, i.e., checking the semantic equivalence between the target code (which is a translation of the source program being compiled) and the source code is called translation validation. Conventionally, the basic method for validation is to check the equality in input and output pairs between the source and the translated program. However, this is not a sound proof that strictly establishes equivalence between the programs.

Petri Nets have been a popular paradigm for modelling instruction-level parallel behaviours [1]. The un-timed one safe CPN (Colour Petri Net) model enhances the classical Petri net to capture natural concurrency present in programs; they have well defined semantics of computations over integers, reals and general data structures. A CPN model involves permitting places in a Petri net, to hold tokens with data values, and the transitions to have associated conditions of executions and the data transformation is associated with out going edges (transition to place). Being value based with a natural capability of capturing parallelism, CPN models depict data dependencies vividly; so they are more convenient as an Immediate Representation of both, the source and translated programs.

In [1], the researchers have proposed an Eclipse plugin based verification tool called SamaTulyata, that verifies the semantic equivalence of two programs using Petri net-based model. However, it cannot handle the loop involving code optimizing transformations.

In this paper, we propose a translation validation technique based on a restricted CPN model of programs which verifies several code optimizing transformations involving loop. Through a small set of experimentations, we have compared our method with both SamaTulyata and CDFG based equivalence checkers. The major contributions of our work are as follows:

- Development of efficient Petri net based models for programs.
- Development of efficient equivalence checking algorithm which can handle various loop involving code-optimizing transformations.

This paper is organised as follows: Section 2 presents an overview of the general workflow of our method. Through a motivating example, we have illustrated our equivalence checking mechanism in Section 3. Section 4 provides the experimental results, comparing our tool with SamaTulyata and two other CDFG-based equivalence checking tools. Section 5 describes the related work. Finally, we conclude our paper by summarizing our results in Section 6.

2 Workflow

Fig. 1 displays the workflow of the current work. A high level program, P_s, is compiled using some compiler transformation techniques which generates an optimized intermediate (translated) code, P_t.

It is important to validate the translation. For analysis of this translation, it is necessary to convert these programs into an equivalent formal model. In this work, we have chosen CPN (Color Petri Net) as our modelling paradigm. This is because a CPN can vividly capture instruction-level parallelism in a vivid manner. The Petri Net Model Constructor module in our work generates the CPN models M_s and M_t corresponding to the programs P_s and P_t, respectively.

In a general program with loop/s, we do not know how many times the loop/s will be executed. To analyze translations involving loops, it is necessary to express the CPN models computations (with a possibly infinite number of loop traversals)

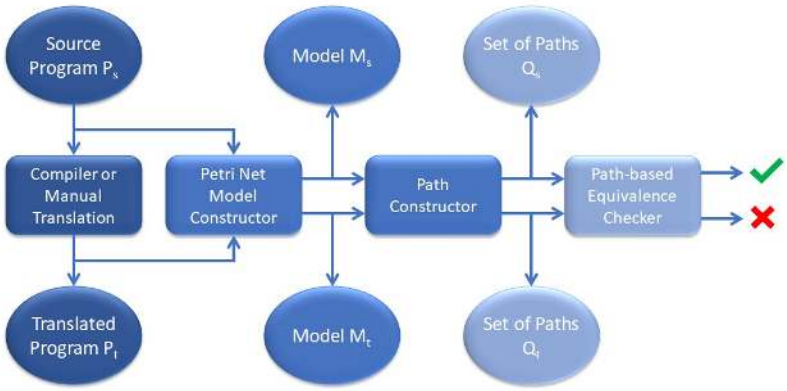

Fig. 1. Basic workflow

into a finite number of paths. This is facilitated by the Path Constructor module that gives us the set of paths, Q_s for M_s, and Q_t for M_t. A path is characterized by its corresponding data transformation functions and related conditions of execution.

The notion of equivalence checking used is as follows: "$\forall$ paths $\in M_s$ $\exists$ an equivalent path in M_t". The Path-based Equivalence Checker module takes the sets of paths for both the CPN models, and using the concept of extension of paths that we have developed, checks for equivalence between the paths and returns a Yes/No answer for the semantic equivalence between source and translated programs. The module never gives a false positive result.

3 Methodology

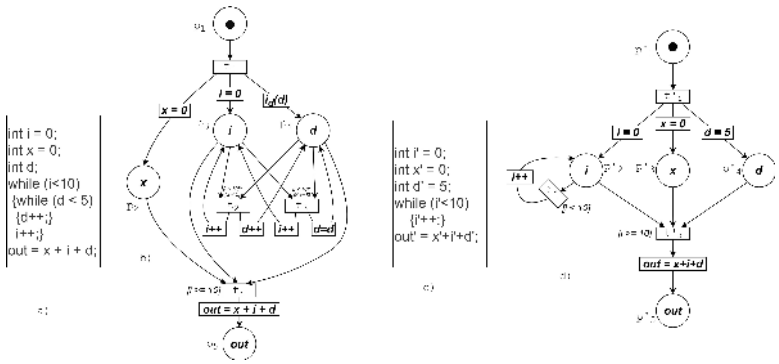

Fig. 2. Motivating example a) Source program b) Petri net model of the source program c) Target program d) Petri net model of the target program

The motivating example for this paper is depicted in Fig. 2. The source program depicted in Fig. 2.(a) computes the statement $out = x + i + d;$. In this program, the statement $d++$ increments d repeatedly with the loop variable i, until its value reaches 5. Consequently, the loop variable i reaches its maximum specified value, 10, and the program finally computes the value of the variable *out*. In the translated version, which is depicted in Fig. 2.(c), the variable d is directly initialised with the value 5. It is to be noted that the two programs are semantically equivalent. This translation is commonly known as loop-independent code-optimizing transformation. To prove the semantic equivalence we have derived the following steps.

3.1 Formalism of Restricted CPN Model

A restricted CPN Model is an eight tuple $N = \langle P, V, f_{pv}, T, I, O, inP, outP \rangle$,

- The set $P = \{p_1, p_2, ..., p_m\}$ is a finite non-empty set of *places*.
- The set V is the set of variables of the program which N seeks to model.
- The item $f_{pv} : P \rightarrow V \cup \{\delta\}$ depicts an association of the places of N to the program variables V; the role of δ is explained shortly. $f_{pv}(p)$ assumes values from a domain D_p. Depending on the type of variable $f_{pv}(p)$, the token value at the place p may be of type Boolean, integer, structure, etc.
- The set $T = \{t_1, t_2, ..., t_n\}$ is a finite non-empty set of *transitions*.
  - Each transition $t \in T$ is associated with a guard condition $g_t : D_{p_1} \times D_{p_2} \times ... \times D_{p_n} \rightarrow \{\top, \bot\} \mid p_1, p_2, ..., p_n$ are pre-places of transition t.
- $I \subset P \times T$ is a finite non-empty set of input arcs which define the flow relation between places and transitions.
- $O \subset T \times P$ is a finite non-empty set of output arcs which define the flow relation between transitions and places.
  - Each outgoing edge $o = (t, p)$ is associated with a function $f_o : D_{p_1} \times D_{p_2} \times ... \times D_{p_n} \rightarrow D_p \mid p_1, p_2, ..., p_n$ are pre-places of transition t
- The set inP $\subset$ P, is the set of input places of the program. These are the places which are initially marked before the program is formally executed.
- The set outP $\subset$ P, is the set of output places of the program. These are the places whose token value represents the output of the program.

Definition 1. *A marking is a function $M : P \rightarrow \{0, 1\}$ that denotes the absence or presence of a token in the places of the net. The restricted CPN model is one-safe.*

Definition 2. *The firing of an enabled transition t, changes the marking M into a new marking M^+. Let the input-set $^\circ t = \{p_1, p2, ..., p_a\}$ and output set $t^\circ = \{q_1, q2, ..., q_b\}$, these events occur simultaneously:*

- *Tokens from the input-set $^\circ t$ are removed. $M^+(p_i) = 0 \; \forall \; p_i \in {}^\circ t$.*
- *One token is added to each place in its output-set t°. $M^+(q_i) = 1 \; \forall \; q_i \in t^\circ$.*
- *Each new token in t° has a token value which is calculated evaluating the respective output function.*

3.2 Model Constructor

Fig. 2.(b) depicts the Petri net model corresponding to the source program in Fig. 2.(a). The model is derived from the rudimentary automated model constructor as reported in [SamaTulyata]. When the program starts, the token is initially in the *inPort* place p_1 and the transition t_1 is executed. The token is removed from p_1 and

tokens go parallel, albeit with different values that are dependent on the function associated with the respective outgoing edge, to p_2, p_3, and p_4 Transitions t_2, t_3 and t_4 have their associated guard conditions. A particular transition is only executed when its respective guard condition is true. In this scenario, the current value of the token in p_3 is 0. Transition t_2 fires because it's guard condition is true, the value of the tokens in p_3 and p_4 is incremented by 1. This loop executes repeatedly, incrementing p_3 and p_4, till the guard condition of t_3 is satisfied. Then t_3 fires to increment the value of the token in p_3 till the guard condition of t_4 is satisified. Finally, the guard condition of t_4 is true and the token with value x + i + d is sent to $outPort$ p_5.

In Fig. 2.(d), places p'_2, p'_3, and p'_4 are associated with the values of the variables, i, x, and d respectively. Transition t'_1 assigns the values 0, 0, and 5 to their respective tokens. t'_2 is now enabled since p'_2 is it's pre-place and the associated guard condition for firing t'_2, $i < 10$ is true. t'_2 fires to increment the value of the token in p'_2. This cycle executes another 9 times until the value of the token in p_2, indicating the value of the variable i, equals 10. Once the token value equals 10, t'_3 is enabled and fired, since its guard condition corresponding to p'_2, $i \geq 10$ is satisfied. t'_3 sums the values of the tokens in p'_2, p'_3, and p'_4 and transfers this value as a token to place p'_5, which indicates the value of the variable out.

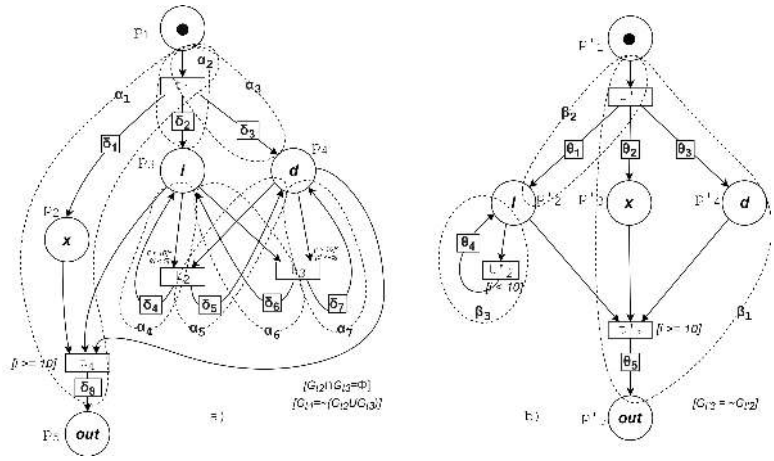

Fig. 3. Simplified Petri nets with the paths marked for a) The source program and b) The target program.

The abstraction begins with defining the function δ which is associated with each outgoing edge of the Petri net. It is a mapping from the edge e, such that $e \in (t, p), \forall t \in T, p \in P$, to the particular function concerning token value transformation, associated with the edge. This is in contrast to the previous model used in [1] where each data transformation function is associated to a particular transition instead.

3.3 Validity of Path-based Equivalence Checker

In a general program with loops, we do not know how many times the loop will be executed. To prove semantic equivalence and represent the computation in the terms

of finite number of paths, we cut the loops and construct the paths in the graph from cut-point to cut-point without any intermediate cut-point. In our equivalence checking mechanism, the notion of cut-points is as follows:

- *inPorts* are cut-points.
- *outPorts* are cut-points.
- The places with back-edges are cut-points.

Using backward traversal and cone of influence method, we find the sequence of parallelizable functional *paths*, from cut-point to cut-point. If a function has been covered in one path, it need not be considered as part of another path. The corresponding paths in Fig. 2.(b) have been marked in Fig. 3.(a). They are:

$$\alpha_1 = \langle\{\delta_1\}, \{\delta_8\}\rangle,\ \alpha_2 = \langle\{\delta_2\}\rangle,\ \alpha_3 = \langle\{\delta_3\}\rangle,\ \alpha_4 = \langle\{\delta_4\}\rangle,\ \alpha_5 = \langle\{\delta_5\}\rangle,\ \alpha_6 = \langle\{\delta_6\}\rangle,\ \alpha_7 = \langle\{\delta_7\}\rangle$$

Similarly, the paths in Fig. 2.(c) have been marked in Fig. 3.(b). They are:

$$\beta_1 = \langle\{\theta_2, \theta_3\}, \{\theta_5\}\rangle,\ \beta_2 = \langle\{\theta_1\}\rangle,\ \beta_3 = \langle\{\theta_4\}\rangle$$

Now we show the validity of the path-based equivalence checker i.e., any computation can be captured as a concatenation of parallel paths. The computation from the Petri net model of the source code in Fig. 3.(a) is $\mu_{p_5} = \langle\{p_1, \{p_2, p_3, p_4\}^{n+m}, p_5\rangle$. Similarly, the computation for the Petri net model of the translated code in Fig. 2(b) is $\mu_{p'_5} = \langle\{p'_1, \{p'_2, p'_3, p'_4\}^n, p'_5\rangle$.

The same computations can be written in terms of the sequence of transitions fired. The method to do the same is as follows: The ith element of the computation in terms of the transitions is the transition/s that fires when moving from marking i to i+1 in the previous version of the computation. So $\mu_{p_5} = \langle\{t_1\}, \{t_2\}^n, \{t_3\}^m, \{t_4\}\rangle$. Similarly, $\mu_{p'_5} = \langle\{t'_1\}, \{t'_2\}^{n+m}, \{t'_3\}\rangle$.

The same computation can the be expressed in terms of the delta functions we defined earlier. To do so, we iterate once over the computation in terms of the transitions. For the new computation, we, append the deltas associated with each transition in the iteration, to the new computation.

So, $\mu_{p_5} = \langle\{\delta_1, \delta_2, \delta_3\}, \{\delta_4, \delta_5\}^n, \{\delta_6, \delta_7\}^m, \delta_8\rangle$.

Similarly, $\mu_{p'_5} = \langle\{\theta_1, \theta_2, \theta_3\}, \{\theta_4\}^{n+m}, \theta_5\rangle$.

The steps to capture the same in terms of paths is as follows: δ_8 is the last member of the computation sequence μ_{p5} and is also the last member of the path α_1. So α_1 is appended to the computation sequence $\mu^r_{p_5}$ and all the δ-function members of α_1 will be deleted from μ_{p_5}. The method terminates when $\mu_{p_5} = \emptyset$. For this computation, $\mu^r_{p_5} = \langle\{\alpha_2 \parallel \alpha_3\}, \{\alpha_4 \parallel \alpha_5\}^n, \{\alpha_6 \parallel \alpha_7\}^m, \alpha_1\rangle$. Similarly, $\mu^r_{p'_5} = \langle\{\beta_2\}, \{\beta_3\}^{n+m}, \beta_1\rangle$.

3.4 Equivalence Checking Mechanism

The intuitive idea of the the equivalence checking mechanism is described in the following algorithmic steps:

step 1) The set of paths in the first program and the second program are respectively: $\pi_0 = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7\}$, $\pi_1 = \{\beta_1, \beta_2, \beta_3\}$

step 2) For the path α_1, we have to find the corresponding candidate path. There is no candidate path path for α_1 because the set of pre-places of α_1, has no direct correspondence with any path in π_1. So, we have to extend α_1.

step 3) Extension: The parallel paths set of α_1 is $\{\alpha_2, \alpha_3\}$. For the path α_2, the pre-place of α_2 has direct correspondence with the pre-place of β_2 and their data transformation and condition of execution are equivalent. Hence, $\alpha_2 \cong \beta_2$. So, α_2 is removed from the parallel list set of α_1. Therefore, p_3 corresponds with p'_2.

Now, the pre-places of α_4 correspond with pre-place of β_3, but their data transformation and condition of execution do not match. So we have to extend α_4. The parallel list set of α_4 is the singleton $\{\alpha_6\}$. α_6 also does not directly correspond to any path from π_1. Hence we have to extend the path α_4. The pre-places and post-places of α_4 and α_6 match. The concatenated path is of the form $\alpha' = \alpha_4 \parallel \alpha_6$. This $\alpha' \cong \beta_3$, since the data transformation and condition of execution of the paths match.

We are left with $\{\alpha_1, \alpha_3, \alpha_5, \alpha_7\}$ and $\{\beta_1\}$ For α_3, the post-path of α_3 is $\{\alpha_5, \alpha_7\}$ and $\{\alpha_5, \alpha_7\}$ are the pre-path of α_1. Now, the concatenated path is of the form $\alpha'' = (\alpha_1 \parallel (\alpha_3 \cdot (\alpha_5 \parallel \alpha_7)))$. The pre-places of α'' correspond with the pre-places β_1 and data-transformation is matched and the condition of execution is of the form $(R_{\alpha 1} \vee (R_{\alpha 3} \wedge (R_{\alpha 5} \vee R_{\alpha 7})))$. Hence, $\alpha'' \cong \beta_1$

$$\alpha_2 \cong \beta_2, (\alpha_4 \parallel \alpha_6) \cong \beta_3, (\alpha_1 \parallel (\alpha_3 \cdot (\alpha_5 \parallel \alpha_7))) \cong \beta_1$$

4 Experimental results

4.1 Preparation of Benchmarks

We have taken five benchmark programs from HLS benchmark suite provided in [6]. The programs are: LCM - Calculates the least common multiple of two input numbers, GCD - Calculates the greatest common divisor of two input numbers, MODN - Calculates (a*b) modulo n, where $a, b < n$, PERFECT - Checks whether the input number is perfect or not, SOD - Carries out repetitive summation of the digits of the input number and of the number obtained in each iteration until the sum becomes a single digit MINMAX - Returns the maximum of three numbers and minimum of the next three numbers.

4.2 Observation

We have tested the prototype of our tool, SamaTulyataII [4], on a 2.5GHz Intel(R) Core(TM)-i5-7200U processor. We feed the programs into [1] and our proposed tool. A comparison of the results is given in Table 1. We have compared the model size in terms of places and transitions, and the capability for handling code optimizing transformations. In our experimentation, we have taken four transformations: duplicating up, duplicating down, boosting up, and loop involving code optimizing transformation. SamaTulyata [1] cannot handle loop involving code optimizing transformation. In our experimentation we have taken two data intensive benchmarks GCD and LCM, and the loop involving code optimization transformation has been applied on them. From Table 1, it is to be noted that the size of the model (in terms of places and transitions) is lesser than the model constructed in SamaTulyata[1]. We have compared our equivalence checking tool with SamaTulyata and two other CDFG-based equivalence checking tools, 1) FSMDEQX-VP: Finite-State Machine with Datapath based Equivalence Checking tool with Value Propogation based network as described in [2] and 2) FSMDEQX-EVP an extended version of FSMDEQX-VP as given in [3].

[4] https://github.com/soumyadipcsis/SamaTulyata/blob/patch-3/SamaTulyataII.zip

Table 1. Model size and Capabilities and for several sequential benchmarks

Benchmark	Transformation	SamaTulyata [1]			SamaTulyataII			FSMDEQX-VP [2]	FSMDEQX-EVP [3]
		place	trans	Capable	place	trans	Capable	Capable	Capable
LCM	Code optimizing	34	28	X	6	6	✓	X	✓
GCD	transform involving loop	31	27	X	7	7	✓	X	✓
SOD	Dynamic loop	11	9	✓	4	2	✓	✓	✓
PERFECT	scheduling	19	14	✓	6	4	✓	✓	✓
MODN	Loop swapping	28	21	✓	6	4	✓	X	X
MINMAX	transformation	28	21	✓	7	7	✓	X	X

Our proposed tool, SamaTulyataII, is able to validate several code-optimzing transformations involving loop. The CDFG-based methods fail to validate loop-swapping transformation since the control structure is changed. Both SamaTulyata and FSMDEQX-VP cannot validate code-optimizing transformation involving loop, but the extended version of FSMDEQX-VP ie. FSMDEQX-EVP is able to validate.

In case the control structure of the program is altered, our tool is able to handle such cases. This is because our method captures instruction level parallelism vividly.

5 Related Works

Translation validation was first introduced by Pnueli et al. in [10] and was demonstrated by Necula et al. [8] and Rinard et al. [9]. The approach was then further enhanced by Kundu et al. [7] where they verified a high-level synthesis tool named SPARK. All the techniques mentioned above are basically bisimulation-based methods. The basic idea of a bisimulation-based method is to find a one-one correspondence between the loops and iterations of a program. The number of iterations of the corresponding loops in the source and translated programs must be the same. The major limitation of this method is that it can only validate structural preserving transformations, i.e., if code moves beyond the basic block level boundaries it fails to validate. Inductive inference based equivalence method only handle structural preserving transformations [5].

Tvoc is a translation validation tool developed by Goldberg, et al. for optimizing compilers that utilize structure preserving and structure modifying transformations [4]. Tvoc uses several proof rules to check equivalence depending on the type of transformation such as interchange, tiling, skewing, etc. However, Tvoc is unable to validate combinations of structure preserving and structure modifying transformations.

Path based validation methods handle both structure preserving transformations and some non-structure preserving code optimizing transformations. One class of the non structure preserving code optimizing transformation is loop involving code optimization which is commonly used in scheduling phase of high level synthesis. A path based equivalence checking mechanism using CDFG-based technique is reported in [2] which can handle the code motion across loop, but it cannot handle loop optimizing transformations because the control structure is altered. To overcome this, Petri net based equivalence checking methods was proposed in [1]. But the major drawbacks of this method are the blow-up in the model size, and its inability to handle the code optimizing transformations involving loops.

6 Conclusion

We have developed an efficient method for checking equivalence between two scalar handling programs using Petri net based model. This method can handle several loop involved code optimizing transformations, code motion across loop, duplicating up, duplicating down, boosting up, boosting down, and loop swapping transformations. In this work we have considered those programs which works only for integer type variables with no function calls, arrays, or pointers. The major limitations of this method are that it cannot handle loop shifting, loop reversal, software pipe lining based transformations. Further work is aimed at overcoming these limitations and extension of our method to capture array handling programs.

References

1. Bandyopadhyay, S., Sarkar, S., Sarkar, D., Mandal, C.: Samatulyata: An efficient path based equivalence checking tool. In: International Symposium on Automated Technology for Verification and Analysis. pp. 109–116 (09 2017). https://doi.org/10.1007/978-3-319-68167-28
2. Banerjee, K., Karfa, C., Sarkar, D., Mandal, C.: Verification of code motion techniques using value propagation. IEEE TCAD **33**(8) (2014)
3. Banerjee, K., Sarkar, D., Mandal, C.: Extending the fsmd framework for validating code motions of array-handling programs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **33**(12), 2015–2019 (2014)
4. Barrett, C., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.: Tvoc: A translation validator for optimizing compilers. In: International Conference on Computer Aided Verification. vol. 3576, pp. 291–295 (07 2005)
5. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 349–360 (2014), https://doi.org/10.1145/2642937.2642987
6. Gupta, S., Dutt, N., Gupta, R., Nicolau, A.: Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In: Proc. of Int. Conf. on VLSI Design. pp. 461–466. IEEE Computer Society, Washington, DC, USA (Jan 2003)
7. 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). https://doi.org/10.1109/TCAD.2010.2042889, http://dx.doi.org/10.1109/TCAD.2010.2042889
8. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI. pp. 83–94 (2000)
9. Rinard, M., Diniz, P.: Credible compilation. Tech. Rep. MIT-LCS-TR-776, MIT (1999)
10. Zuck, L., Pnueli, A., Goldberg, B., Barrett, C., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Form. Methods Syst. Des. **27**(3), 335–360 (2005). https://doi.org/http://dx.doi.org/10.1007/s10703-005-3402-z