Symbolic Solver for Live Variable Analysis of High Level Design Languages

Debasish Das
Department of EECS
Northwestern University, USA
dda902@ece.northwestern.edu

ABSTRACT

This paper presents an efficient binary decision diagram solver for live variable analysis of high level design languages like SpecC. A novel representation of the program as a global structure using efficient data structure called Binary Decision Diagrams is presented. Based on the global structure of the program, a relational framework for live variable analysis is proposed and the solver algorithm is developed based on the relational framework. The algorithm obtains the fixed point of live variable analysis in iterations significantly less than the traditional chaotic iteration approach. The global representation of the program using Binary Decision Diagrams and the ease of doing set theoretic operations in the domain of Binary Decision Diagrams helps the solver algorithm to determine the fixed points of the analysis efficiently. Our experimental evaluations using the solver demonstrate a marked decrease in iterations over traditional approach for fixed point calculations.

General Terms

Algorithms

Keywords

System Level Design, High Level Synthesis, Algorithms, Binary Decision Diagrams, Live Variable Analysis

1. INTRODUCTION

Dataflow analysis is an important part of compiler optimization procedure for high level languages. Substantial work [5] has been done in the dataflow analysis of high level languages like C and C++ but the application of dataflow analysis is not studied in greater detail for high level design languages like SpecC. Live variable analysis is one of the several approaches used for dataflow analysis. This paper proposes a new relational framework to do live variable analysis and presents a symbolic solver to find the fixed points of live variable analysis. To the best of our knowledge there is no work that proposed a framework other than chaotic iterations to find the solution of live variable analysis. The relational framework is implemented using Binary Decision Diagrams and the symbolic solver algorithm works on the Binary Decision Diagrams to produce the solution of Live Variable Analysis. The whole structure of the program is taken into account while doing the iterations and looking at the complete structure helps in reducing the number of iterations needed to get the fixed points of the analysis. We considered a subset of C language to do our experiments and with minor modifications the framework can be used to do dataflow analysis of high level design languages like SpecC [13].

High level design languages like SpecC are increasingly gaining importance in industry as a mean to specify designs. System level models written in languages like SpecC and C++ are used to specify design and then such system level models are synthesized to RTL. Our analyzer can optimize the designs written in languages like SpecC which can be used to generate efficient RTL. As Electronic System Level Design [14] is getting more popular among designers efficient dataflow analysis framework for such languages are required and our framework provides a novel solution to this problem.

BDDs [1-3, 12] are a representation of Quantified Boolean Formulae, and can be used to encode relations on finite domains. BDDs also exhibit efficient encodings of various operations on these relations. The traditional application of BDDs is in the program verification community, where they are used to provide compact encodings of very large state spaces and transition relations. The use of BDDs has enabled the verification of very large circuits having hundreds of millions of states.

Binary Decision Diagrams provides a technique for efficiently representing predicates over finite domains, and for manipulating such predicates. Binary Decision Diagrams has been used by [4] to do points-to analysis of high level languages.

This paper gives an approach to encode the live variable analysis by BDDs and then develops an algorithm that works on the encoding to produce a result. The chaotic iteration approach [5] is traditionally used to solve live variable analysis which is an equational approach [5]. Chaotic Iteration approach finds the solution by finding the fixed points iteratively. BDDs help in obtaining an efficient representation for the live variable analysis and that is because of the set theoretic operations that can be performed very efficiently over BDDs [2]. After that the proposed algorithm operates over the BDDs produced to obtain the fixed point and thus gives the live variable analysis results. Finding fixed point over BDDs is an efficient approach compared to getting the fixed points in equational approach because of the fact that we are looking at the complete structure of the program rather than looking only at the present node in control flow graph and its frontier.

This paper is organized as follows. Section 2 presents the live variable analysis using BDDs. Section 3 describes the algorithm to find the live variable information. Section 4 presents performance analysis of the algorithm. Finally the paper concludes with giving future directions to take advantage of the relational framework which helped in reducing the number of iterations significantly to find the fixed points to do efficient dataflow analysis.

2. BDD TO REPRESENT LIVE VARIABLE ANALYSIS

This section formally defines the live variable analysis and also proposes the relational framework for doing live variable analysis.

2.1 Live variable analysis:

Live variable analysis is one of the several approaches that are used in Dataflow analysis which are namely Available Expressions, Reaching Definitions and Very Busy Expressions. To understand live variable analysis one should be familiar with the following definitions related to a program. [5] deals with a number of dataflow analysis approaches along with the live variable analysis.

2.1.1 Definitions

a) Label: The program is parsed completely and a unique label is associated to each statement.

$$\text{Label: Stmt} \rightarrow \text{Lab}$$

b) Initial label: init returns the initial label of a statement. The statement can be a composite statement when it returns the label of the statement that is coming first out of all the statement of the block.

$$\text{Init: Stmt} \rightarrow \text{Lab}$$

c) Final label: Final returns the final label of a statement. If the statement is a composite statement it returns the label of the last statement of the composite block under consideration. Note that there might be more than one label if the composite statement is a decision block.

$$\text{Final: Stmt} \rightarrow \text{PowerSet(Lab)}$$

d) Flows and reverse flows: For live analysis we have to find the control flow information from the program. So the flow function is defined as following:

$$\text{Flow: Stmt} \rightarrow \text{Powerset(Lab X Lab)}$$

This function maps statement to sets of flows. Examples are the following:

$$\text{Flow(x=a;)} = \emptyset$$

where Ø represents NULL flow. There is no flow information in an atomic statement like *x=a;*

$$\text{Flow(S1;S2)} = \text{Flow(S1) U Flow(S2) U } \{(l,\text{init(S2)}) \mid l \in \text{Final(S1)}\}$$

$$\text{Flow(if(S1) S2;)} = \text{Flow(S2) U } \{(\text{Label(S1)},\text{init(S2)})\}$$

$$\text{Flow(if(S1) S2; else S3;)} = \text{Flow(S2) U Flow(S3) U}\{(\text{Label(S1)},\text{init(S2)}),(\text{Label(S1)},\text{Init(S3)})\}$$

Note that the function Flow uses the definitions of functions Label, Init and Final described above.
Once the flow graph is obtained the reverse flow graph can be obtained as follows:

$$\text{ReverseFlow: Stmt} \rightarrow \text{Powerset(Lab X Lab)}$$

$$\text{ReverseFlow(S)} = \{(l , m) \mid (m , l) \in \text{Flow(S)}\}$$

where S is the whole program.

These definitions help to generate the control flow information of the program. A prerequisite to this step is a program which is uniquely labeled.

2.1.2 Mathematical definition of Live Variable Analysis

A variable is *live* at the exit from a label if there is a path from the label to a use of the variable that doesn't re-define the variable. The Live Variable Analysis determines set of variables that are *live* at a program point or Label (as defined in section 2.1.1) after the control exits from that particular Label.

This analysis is used as the basis of Dead Code elimination. Note that for synthesizing a specified design into RTL, Dead Code Elimination can result in a RTL with less resources compared to a flow that do not takes into account live variable analysis results. Thus live variable analysis of SpecC results into a RTL that has optimum number of resources. It motivated us to consider live variable analysis in our work.

In Live Variable Analysis Kill and Gen expressions [5] corresponding to each Label are generated. The following table gives the mathematical formulations to generate Kill and Gen expression:
For each unique label in the program Kill and Gen expressions are computed.

<u>KILL FUNCTIONS</u>

$$\text{KILL (x = a ;)} = \{x\}$$

$$\text{KILL (S1)} = \emptyset \qquad \text{//S1 is not an assignment statement}$$

<u>GEN FUNCTIONS</u>

$$\text{GEN (x = a ;)} = \text{FV (a)}$$

$$\text{GEN (S1)} = \text{FV (S1) //S1 is not an assignment statement}$$

The function FV takes a, which can be a function of $y_1, y_2, \ldots, y_n$ and then FV returns all the $y_1, y_2, \ldots, y_n$ which are the constituents of a.

After calculating the KILL and GEN functions corresponding to each label, for each label of the program LVentry and LVexit equations are obtained.

$$LV_{exit}(l) = \emptyset \text{ if } l \in \text{Final(S)}$$

$$\text{U}\{LV_{entry}(m) \mid (m , l) \in \text{ReverseFlow(S)}\}$$

$$LV_{entry}(l) = (LVexit(l)\backslash KILL(l)) \text{ U } GEN(l)$$

where U is the global union, and \ is the set difference.

Live Variable Analysis is a backward analysis so the reverse flow graph is to be iterated to solve the data flow equations. To solve the dataflow equations two algorithms are used normally which are called MFP and MOP [5]. Both iterate through the control flow graph until a fixed point is reached.

2.2 Representing live variable analysis using BDDs

This section formally defines the relational framework that is proposed to do live variable analysis. Equational approaches [5] do not consider the whole analysis as a single relation but in our proposed approach analysis are represented as relations over the semi lattice defined.

2.2.1 Relational framework for live variable analysis:

The analyses expressed in terms of data-flow frameworks [5] like live variable analysis have a very regular structure. Let us first restrict ourselves to problems where the solution of the analysis can be represented as sets, and further to problems that can be represented in terms of *Gen* and *Kill* functions on sets.

a) <u>Analysis Relation</u>: In the above setting, let the meet semi-lattice be *L* (or more completely $<L, \subseteq, \cup, \top>$). In this case the result of the analysis of a program P can be represented as a relation from the program point (label) of P to the lattice of L.

$$Analysis_{P:}\ Pt_P \leftrightarrow L$$

b) <u>CFG Relation</u>: The control flow information of the program P can also be considered as a relation. CFG_P which is a relation from program points(label) to program points(label) ,where a point l_1 is related to point l_2 by CFG_P if there is an edge in the control-flow graph of P taking node l_1 to node l_2.

$$CFG_P: Pt_P \leftrightarrow Pt_P$$

c) <u>Init Relation</u>: The initialization required for the problem can also be represented as a relation. $Init_P$, relating the entry(or exit) program-point of the program with the $\cap$ or $\cup$ value of the semi lattice.

$$Init_P\text{: } Pt_p \leftrightarrow L$$

For live variable analysis the operation is set-union on which the semi-lattice is defined. As it had already been mentioned that live variable analysis is a backward analysis.

d) Live Variable Analysis Relation: Using the relations defined above the Live Variable Analysis can be expressed as the following equations:

$LV_{entry}(pt) = (\bigcup LV(pt') - Kill_{pt}) \bigcup Gen_{pt}$

Where $pt' \in$ successor(pt)

Given a relation, S, representing an approximation to the analysis solution, and the relation for the control-flow graph, the expression $\bigcup LV(pt')$ in the data-flow framework can be represented as the relational composition

$$CFG_P; S$$

Intuitively, relational composition with CFG_P represents the backward flow information in the analysis. The entire analysis equation can be represented by the expression

$((CFG_P; S) - Kill_P) \bigcup Gen_P \bigcup Init_P$

The analysis equation is now a global equation which takes into the account the whole program P rather than taking into account each of the program points of labels which was the initial representation of live variable analysis in the dataflow framework.

The analysis results can be itself now expressed as the fix-point of the function

$$\lambda S. ((CFG_P; S) - Kill_P) \bigcup Gen_P \bigcup Init_P$$

2.2.2 Verification of the proposed framework

In this section we are presenting the soundness and correctness of the mathematical formulation described in section 2.2.1

Proposition 2.2.2.1: Soundness

Proof: Note that the only part of equation 1 that depends on the analysis being a backward analysis and having the set union as the operator is

$$(CFG_P; S)$$

Only this part of the equation needs to be changed to address analysis that is not backward analyses or those that have set intersection as the defined operator on the semi-lattice.

It is quite straight forward to see that in order to consider forward analysis, it is sufficient to consider the transpose, CFG^{-1}_P, of the relation representing the control-flow graph for relational composition.

Proposition 2.2.2.2 Correctness with other operators

Proof: The case of problems where the operator defined on semi-lattice is set intersection is slightly more complex. In this case, the relational composition expression has to be of the form

$$inv(CFG_P; inv(S))$$

where inv(S) represents the inverse of the set S (with respect to some universal set).

Proposition 2.2.2.3 Intersection over Finite Domains

Proof: The above results are tabulated in the table shown below. Note that the relational approach to analysis requires the solution of different equations for different problems – it, in effect, results in four different frameworks for different classes of analysis problems. This is in contrast to data-flow frameworks where the same framework and equation can be used to solve a large class of analysis problems.

	Forward	Backward
$\bigcup$	$(CFG_P; S)$	$(CFG^{-1}_P; S)$
$\bigcap$	$inv(CFG_P; inv(S))$	$inv(CFG^{-1}_P;inv(S))$

3. BDD TO REPRESENT LIVE VARIABLE ANALYSIS

The last section described the relational framework developed to do live variable analysis using the binary decision diagram. This section presents a case study and shows the respective relations encoded in binary decision digrams which are used by the BDD solver developed. The section concludes by showing the execution of the solver algorithm developed.

3.1 Case study

Consider the following high level program code:

```
//Hypothetical start node: 0
{
  x=1;   //1
  y=2;   //2
  z=x;   //3
  while (x>z)   //4
  {
    if(y>z)   //5
    {
      x=y+1;   //6
      z=x+1;   //7
    }
    x=y+z;   //8
  }
  x=x-y;   //9
}//Hypothetical finish node: 10
```

The labelize module takes as input such a code written in high level language and produces a unique label for each statement. It also finds the unique variables in the program code. The control flow graph of the program code is taken as input which represents the flow information in terms of the labels. For example in this case the CFG is:

0->1;1->2;2->3;3->4;4->5,9;5->6,8;6->7;7->4,9;8->4,9;9->10

3.2 Global Binary Decision Digrams for BDD solver

In this paper BuDDy [6] is used as the package to do the operations that are related to binary decision diagrams. The binary decision digrams thus generated respective to each relation is then visually shown using Graphviz [7] package.

3.2.1 Relation encoding using BDD

Suppose a relation with 3 2-tuples {(0,1),(1,2),(2,3)} is to be encoded using BDDs. Now BDDs are function $f:B^n \rightarrow B$, each 2-tuple of the relation can be encoded using binary bits. In the given relation it can be encoded using 2 bits. Thus if 0->00,1->01,2->10,3->11, then 0001, 0110 and 1011 are true in the BDD corresponding to this relation while all other combinations lead to a false value.

3.2.2 Encoding CFG relation

Control flow graph is defined as ProgramPoint X ProgramPoint. The 2-tuple (0,1) is in CFG relation and that's because in the control flow graph, the control flows from label 0 to label 1. Therefore in the BDD corresponding to CFG relation, the binary encoding corresponding to (0,1) should evaluate to true. As there are 11 unique labels therefore 4 bits are taken by the BuDDy package to represent the CFG BDD. The respective CFG BDD for the high level program code shown in Figure 1.

The 2-tuple (8,9) should be in the CFG BDD. 8 is encoded by 1000 and 9 is encoded by 1001. BuDDy [6] encoded the relation by taking one bit from one member of the 2-tuple's binary representation and the next bit from the next member of 2-tuple. This process is carried out alternatively to generate the whole encoding. For this example the encoding is:

9[3]8[3]9[2]8[2]9[1]8[1] 9[0]8[0] = 11000010.

The inspection of the CFG BDD shows that the bit pattern 11000010 leads to true which means that the 2-tuple (8,9) is in the CFG BDD.

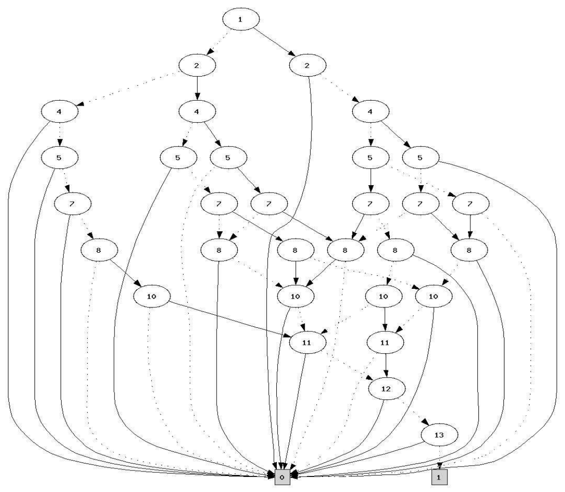

Figure 1: CFG Binary Decision Diagram

3.2.3 Encoding GlobalKill relation

GlobalKill relation captures the KILL function as described in Section 2.1.2 in a global view. To encode the GlobalKill relation a set of labels are found related to each unique variable. From the example program the variable x is killed at the label subset {1,6,8,9}. The variables are also assigned a unique id and therefore the relation GlobalKill is defined on Var X ProgramPoint where Var is the set related to unique variable id and ProgramPoint represent the labels. The GlobalKill BDD is shown in Figure 2.

3.2.4 Encoding GlobalGen relation

GlobalGen relation like GlobalKill captures GEN function described in Section 2.1.2 in a global view. It is also obtained in a similar manner as the GlobalKill relation. For both of them the kill and gen sets are calculated by the definitions as given in section 2. The corresponding BDD for GlobalGen is shown in Figure 3.

3.3 BDD solver algorithm

The algorithm takes as input GlobalCFG, GlobalKill, and GlobalGen Binary Decision Diagrams and finds the LV_{entry} and LV_{exit} information globally for the given high level code.

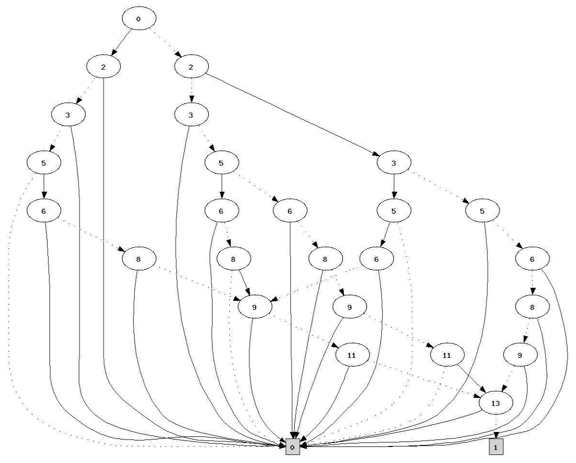

Figure 2: GlobalKill Binary Decision Diagram

<u>*BDDSolver Pseudocode*</u>

```
Step 1. Initiliaze BDDs solver and
        tempsolver as empty
Step 2. while(solver is not equal to
        tempsolver)
Step 3. tempsolver = (relational_product
        (GlobalCFG,solver)-GlobalKill) U
        GlobalGen
Step 4.endwhile
```

This algorithm provides the LV_{entry} information. The BDD given by this algorithm gives the information that which variables are live at respective program points. To find the LV_{exit} information the line 3 of the algorithm is modified as the following:

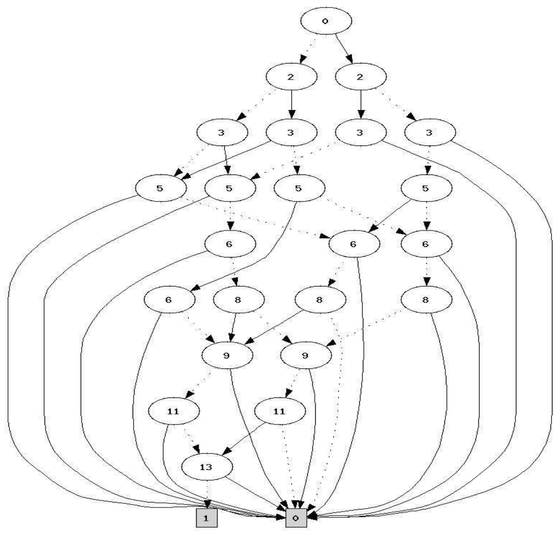

Figure 3: GlobalGen Binary Decision Diagram

`tempsolver= (~relational_product(GlobalCFG, ~solver)-GlobalKill) U GlobalGen`,

where ~ is the negation operation in BDDs. A complete description of operations defined on BDD can be found in [1-2].

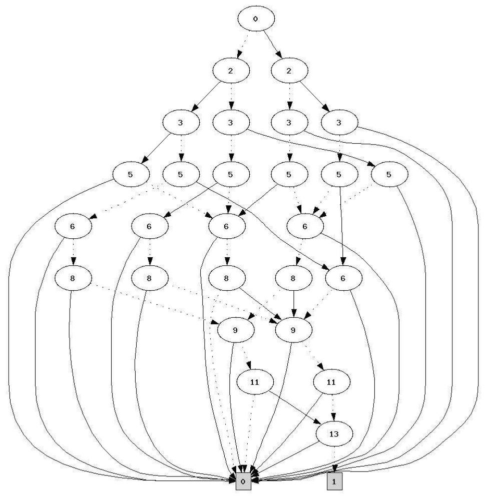

Figure 4: Solver Binary Decision Diagram

3.3.1 Correctness of the solver algorithm

In section 2.2.1 while defining the relational framework for live variable analysis the correctness of the respective equations are shown. The equations are analogous to the live variable analysis defined in the previous section.

3.3.2 Termination of the solver algorithm

Since the live variable analysis is defined as a lattice structure therefore there is always a fixed point of the lattice and therefore the algorithm is guaranteed to terminate.

3.3.3 Revisiting the example

The LV_{entry} BDD corresponding to the example is shown in Figure 4.

This provides the same result as the result provided by the traditional worklist [8] algorithm which iteratively tries to find out the fixed point and thus the LV_{entry} and LV_{exit} information.

4. PERFORMANCE ANALYSES

We have proposed a new approach to do Live Variable Analysis. The traditional worklist algorithm [8-10] is one of the oldest and heavily studied algorithm for performing dataflow analysis. This algorithm finds the fixed point by iterations on the control flow graph but this algorithm doesn't look into the whole structure of the program at one time. The relational framework captures the whole structure of the program and the BDD Solver finds out the fixed points of the analysis in much less number of iterations than worklist algorithm. [11] gives a survey of algorithms that use iterative approach to do dataflow analysis. To compare our solver we have implemented the iterative worklist algorithm mentioned in [11].

We considered intra-procedural Live Variable Analysis in this paper. For intra-procedural analysis generally the code length is not more than 50 or 100 lines. We checked the solver on codes of varying lengths to compare the performance of our algorithm over the traditional iterative algorithm. Note that the framework we proposed is more general and can be applied to both inter-procedural and intra-procedural analysis with appropriate modifications.

The results we obtained are provided in the Table 2. We obtained a significant gain in the iterations that are required to find the fixed points of live variable analysis. Table 3 shows the runtime of the BDD Solver. From Table 2 we notice a marked decrease in number of iterations compared to the traditional chaotic iterations. The run time of BDD Solver includes the time spent in initialization of the Binary Decision Diagram package and formation of global binary decision diagrams corresponding to Control Flow Graph, Kill and Gen relations. The next section shows how we can use the gain from the relational framework and decrease the run time of the BDD Solver.

5. CONCLUSION AND FUTURE WORK

In this paper, we proposed a relational framework to do Live Variable Analysis and gave an algorithm to do Live Variable Analysis. The Binary Decision Diagram based solver finds out the fixed points of Live Variable Analysis in iterations much less than the traditional iterative method of finding the fixed points of Live Variable Analysis. This paper obtained a marked improvement in iterations to find the fixed points and therefore the relational framework can help in finding the fixed points faster than the equational approach. The Symbolic Solver algorithm is implemented using Binary Decision Diagrams and due to the initialization routines of Binary Decision Diagram package the time taken is higher.

We are looking at other symbolic methods to use the relational framework so that we can gain in time. One important future research direction is to map the relational framework to a SAT solver and the Solver algorithm will query the SAT solver to do membership queries for the fixed points. Since SAT based frameworks are proven much faster in the verification community, which traditionally used Binary Decision Diagrams, we also expect a gain in time when a SAT based symbolic solver will be used to query the fixed points.

6. REFERENCES

[1] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, 8(C-35):677-691, 1986.

[2] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys, 24(3):293-318, September 1992.

[3] Henrik Reif Anderson. An introduction to Binary Decision Diagrams. Lecture notes for 49285 Advanced Algorithms E97, October 1997.

[4] Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren, and Navindra Umanee. *Points-to analysis using BDDs*. In PLDI 2003.

[5] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. Principles of Program Analysis. Springer Verlag.

[6] Jørn Lind-Nielsen. BuDDy, A Binary Decision Diagram Package. Department of Information Technology, Technical University of Denmark,http://www.itu.dk/research/buddy/.

[7] AT&T Lab Research. Graphviz – Graph Visualization Software. http://www.graphviz.org/.

[8] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. Journal of the ACM, 23(1):158–171, January 1976.

[9] Ken Kennedy and Scott K. Warren. Automatic generation of efficient evaluators for attribute grammars. In Conference Record of the Third ACM Symposium on Principles of Programming Languages, pages 32–49, Atlanta, Georgia, January 1976.

[10] John B. Kam and Jeffrey D. Ullman. Monotone dataflow analysis frameworks. Acta Informatica,7:305–317, 1977.

[11] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. Iterative Data-flow Analysis, Revisited. Rice Technical Report, TR04-100.

[12] Somenzi, Binary Decision Diagrams. M. Broy and R. Steinbruggen, editors, Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences, pages 303--366. IOS Press, 1999.

[13] SpecC System, http://www.ics.uci.edu/~specc/.

[14] ESL: Tales from the trenches, Panel Discussion, Proceedings of Design Automation Conference 2005, June 13-17, 2005, Anaheim, California, USA

Table 2. Performance Analysis of BDDSolver

Program	C-- Tokens	Constructs	Chaotic Iterations	BDDSolver Iterations
P1	65	While,if	20	3
P2	69	While,if, else	13	5
P3	124	While,if, else	29	7
P4	129	While,if, else	29	7
P5	139	While,if, else	34	9

Table 3. Runtime of BDDSolver

Program	C-- Tokens	Execution Time
P1	65	6.357ms
P2	69	6.59ms
P3	124	7.266ms
P4	129	7.501ms
P5	139	7.903ms