PROJECT REPORT

DEVELOPMENT OF A BINARY DECISION DIAGRAM SOLVER FOR LIVE VARIABLE ANALYSIS

DEBASISH DAS
ROLL NO : 00CS1004
DEPARTMENT OF COMPUTER SCIENCE AND ENGG
INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR
e-mail : ddas@cse.iitkgp.dhs.org
debasish_83@yahoo.co.uk
Homepage :http://www.iitian.iitkgp.ernet.in/~u0cs1004

PROJECT GUIDE : DR PRAHLADAVARADAN SAMPATH
TRDDC , PUNE
INDIA

ORGANISATION: TATA RESEARCH DESIGN AND DEVELOPMENT CENTRE
TATA CONSULTANCY SERVICES
HADAPSAR, PUNE
INDIA

PROJECT DURATION : 2 months
5th May 2003–4th July 2003

DONE AS INDUSTRIAL TRAINING FOR FULFILLING THE COURSE REQUIREMENT OF BTECH(H) DEGREE FROM IIT KHARAGPUR

COURSE NAME : INDUSTRIAL TRAINING(Credits-2)

Abstract: This report presents development of a technique to solve program analysis problems using BDDs (Binary Decision Diagrams). BDDs are a technique for efficiently representing predicates over finite domains, and for manipulating such predicates.

BDDs present an opportunity to symbolically represent the analysis of a program as a predicate over the program. This symbolic representation has a normal form, and hopefully is compact. This is likely to be of considerable interest for inter-procedural analysis, where the "effect" of a procedure in every possible dynamic context can be symbolically represented as a single BDD -- BDDs provide an efficient representation
of analysis information for a procedure.

Solution of an analysis problem represented as a BDD reduces to solving a fix-point equation over BDDs. I have encoded Live Variable analysis using BDDs and provided two different encodings of the analysis problem into BDDs.

The first approach solves the Live Variable equations by iterating through the CFG(Control Flow Graph).This approach is an extension of the solution of Live Variable equations using traditional methods like chaotic iteration. The use of BDD is a new
approach and as BDDs are well known in hardware community for decreasing complexity for the set theoretic operations like union, intersection, set difference etc,so this approach will decrease the cost of iteration in traditional approach.

The second approach solves the live variable analysis at a global level. It computes the fixed points by doing set theoretic operations on global BDDs. Encoding the live
variable analysis using the BDDs is a new approach and in future work it would be extended to various other analyses of the montone framework of dataflow analysis.

The theoretical and implementation details of both the methods are provided later in the report. The approach of encoding analyses using BDDs would be extended to alias analysis also. Alias is a relation which is both symmetric and transitive and BDDs can be used efficiently to depict relation and for all operations on relations efficient BDD algorithms exists. So BDDs can represent and manipulate relations easily and this would be the main crux of my future work on program analysis.

Introduction: The report deals with the development of a solver for Live Variable analysis using Binary Decision Diagram. Live variable analysis is one of several analysis of monotone dataflow framework. Other analysis of the framework includes available expression analysis, reaching definitions analysis, very busy expression analysis etc. To understand the concept one should know the mathematical background of dataflow analysis in particular live variable analysis. Since the solver is developed using Binary Decision Diagram familiarity with BDDs is also necessary. Hence the introduction will describe dataflow analysis and binary decision diagrams. Interested readers can go through the references for a detailed study.

1. Introduction to dataflow analysis (in particular live variable analysis)

In dataflow analysis it is customary to think of a program as a graph: the nodes are the elementary blocks and the edges describe how control might pass from one elementary block to other. This graph is called control flow graph or CFG and it represents the flow of control in a program. The program might be a intra-procedural program or a inter-procedural program. For intra-procedural program the CFG is fairly simple but for inter-procedural program CFG can be quite complex. There are two basic approaches for dataflow analysis:

a) Equational approach
b) Constraint approach

I have worked on equational approach therefore I would give a brief introduction to
equational approach.

The Equational Approach

The equation system: An analysis like live variable analysis(or reaching definitions) can be specified by extracting a number of equations from a program. There are two classes of equations. One class of equations relate entry information of a node to exit information for the same node.

$$LVentry(l) = (LVexit(l) \setminus kill(l)) \cup gen(l)$$

The other class of equations relate exit information of a node to entry information of nodes from which there is an edge to the node of interest; that is exit information is obtained from all the entry information where control could have come from. To generate the second class of equations we need the CFG since the control information of the program is captured by control flow graph only.

$$LVexit(l) = \begin{cases} \emptyset & \text{if } l \in final(S) \\ \cup \{LVentry(m) \mid (l, m) \in flow(S)\} \end{cases}$$

After generation of the two class of equations we will find that the equations are mutually dependent. As we know that solution of a mutually dependent set of equations might not be a unique solution so the problem of finding the solution comes down to finding the fixed points of a lattice ref [1].

Let's take a small example program and show how the problem of finding the solution of dependent equation changes into finding the fixed points of lattice. A small program written in C-- is given below along with the kill and gen sets. How the kill and gen sets will be computed will be described later.

```
void main( )
{
   int x, y, z;
   x=2;
   y=4;
   x=1;
   if(y>x)
     z=y;
   else
     z=y*y;
}
```

Kill and Gen sets

Label	Kill(l)	Gen(l)
1	{x}	Ø
2	{y}	Ø
3	{x}	Ø
4	Ø	{x,y}
5	{z}	{y}
6	{z}	{y}
7	{x}	{z}

Hand computation of the equation gives us the following equations:

First class of equations

$LVentry(1) = LVexit(1) \setminus \{x\}$
$LVentry(2) = LVexit(2) \setminus \{y\}$
$LVentry(3) = LVexit(3) \setminus \{x\}$
$LVentry(4) = LVexit(4) \cup \{x,y\}$
$LVentry(5) = (LVexit(5) \setminus \{z\}) \cup \{y\}$
$LVentry(6) = (LVexit(6) \setminus \{z\}) \cup \{y\}$
$LVentry(7) = \{z\}$

Second class of equations

$LVexit(1) = LVentry(2)$
$LVexit(2) = LVentry(3)$
$LVexit(3) = LVentry(4)$
$LVexit(4) = LVentry(5) \cup LVentry(6)$
$LVexit(5) = LVentry(7)$

LVexit(6) = LVentry(7)
LVexit(7) = Ø

The above system of equations defines the 14 sets

LVentry(1), LVexit(1), …….., LVentry(7), LVexit(7)

in terms of each other. Writing a vector LV for this 14 tuple of sets we can regard the equation system as defining a function F and demanding that:

LV = F(LV)

More specifically,

F(LV) = (Fentry(1)(LV), Fexit(1)(LV),…….,Fentry(7)(LV), Fexit(7)(LV))

For example,

Fexit(4)(……, LVentry(5), …, LVentry(6), ….) = LVentry(5) U LVentry(6)

F operates over fourteen-tuples of sets of pairs of variables and labels; It can be written as

F : $(\text{PowerSet}(\text{Var X Lab}))^{12}$ → $(\text{PowerSet}(\text{Var X Lab}))^{12}$

It implies that:

i) $(\text{PowerSet}(\text{Var X Lab}))^{12}$ can be partially ordered by setting

LV≤LV' iff for all i : LV (subset of) LV'

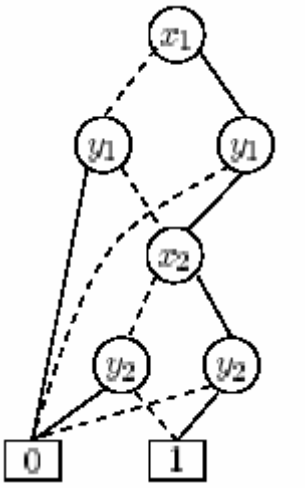
where LV = $(LV_1,\ldots\ldots LV_{14})$ and similarly LV' = $(LV'_1,\ldots\ldots LV'_{14})$

ii) It turns $(\text{PowerSet}(\text{Var X Lab}))^{12}$ into a complete lattice and so the solution to the equation becomes the solution of the equation LV = F(LV) which will find the fixed points of the given lattice ref[8].

Live variable analysis is one of several analyses that is based on an equational approach. Since my work is centered on live variable analysis, I will present the detailed theory of live variable analysis.

2. Introduction to binary decision diagrams: A Binary Decision Diagram(BDD) is a representation of a set of binary strings of length n that is often equivalently represented as a binary-valued function that maps strings of length n to 1 if they are in the set and to 0 if they are not in the set.

BDD are used to find the truth values of any Boolean expression. Suppose x and y are Boolean variables then we can represent any expression involving the Boolean variables x and y using a BDD. The BDD operations are of less complexity compared to similar operations done using other approaches. BDDs are used a lot in the field of computer hardware like VLSI, Electronic Design Automation for verification of the circuit. Use of BDDs in software engineering is a relatively new topic.



Example: A BDD for the Boolean formula (x1 ⇔ y1) ∧(x2⇔y2) with ordering x1<y1<x2<y2. Low edges are drawn as dotted lines and high edges are drawn as solid lines.

Past Studies: Sable Research Group, McGill University have done the work on representing Points-to Analysis using Binary Decision Diagram. They have developed a solver to find the points-to relation of a

given program. The Points-to Analysis was developed initially by Anderson for C programming language and later it was extended to object oriented language like Java. Sable research group developed the solver using BDD for subset-based points-to analysis for Java. A brief description of points-to analysis theory is given in theory section. Interested readers can go through the paper of Points-to Analysis using BDD ref [2] and Anderson's PhD thesis ref [3].Till now no work has been done on the representation of dataflow analysis using BDD and sable research group is also looking forward to extend the BDD approach to other analyses.

Theory: The theory portion will present the mathematical aspects of dataflow analysis, a brief description of binary decision diagrams, description of Points-to Analysis using BDD suggested by Sable Group and Anderson's dataflow analysis.

1. Data Flow Analysis: Dataflow analysis is the traditional form of program analysis which is described in several textbooks on compiler writing [4].I will present analyses for a subset of C language(called C--) [5].Dataflow analysis includes several different analysis namely Available Expressions , Reaching Definitions, Very Busy Expressions and Live Variables. I will introduce the operational semantics for C-- (which is like imperative WHILE language [1]) for doing live variable analysis. Since this is an extension from the operational semantics defined for WHILE language the theoretical validity of the operational semantics and correctness of live variable analysis can be found in ref [1].

Before defining the operational semantics of live variable analysis we will need the definition for certain terms that would be helpful in understanding the analysis:

a) Label: The program written in C-- should be parsed completely and a unique label should be associated with each statement. Later on when doing the data flow analysis each label would be an unique entry and we will solve the analysis according the labels.

Label: Stmt → Lab

Label will take a statement and return the unique label associated with the statement.

b) Initial and final labels: When presenting live variable analysis a number of operations would be done on program and labels. The first of these is:

init: Stmt → Lab

init returns the initial label of a statement. Each statement has to be assigned an unique label in a program to do live variable analysis. init will return the initial label of a statement. The statement can be an composite statement when it will return the label of the statement that is coming first out of all the statement of the block. The examples are following:

Statement	Label	Result after applying init	Comments
x=a;	Label(x=a;)	Label(x=a;)	assignment
S1;S2	Label(S1) Label(S2)	Label(S1)	composite
if(S1) S2;	Label(S1) Label(S2)	Label(S1)	conditional

if(S1)
S2;
else
S3;
	Label(S1)
	Label(S2)
	Label(S3)
		Label(S1)	conditional

while(S1) S2;	Label(S1) Label(S2)	Label(S1)	iteration

Similarly a function called final is also needed

Final: Stmt → PowerSet(Lab)

Example of the application of Final function:

Final(x=a;) = Label(x=a;) //unique label associated with the assignment statement

Final(S1;S2) = Final(S2)

Final(if(S1)
S2;) = Final(S2);

Final(if(S1) = Final(S2) U Final(S3)
S2;
else
S3;)

Final(while(S1) = Label(S1);
S2;)

c) Flows and reverse flows: For dataflow analysis we have to find the flow in the program. So the flow function is defined as following:

flow: Stmt → Powerset(Lab X Lab)

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

flow(x=a;) = Ø

flow(S1;S2) = flow(S1) U flow(S2) U {(l,init(S2))| l ϵ final(S1)}

flow(if(S1) S2;) = flow(S2) U {(Label(S1),init(S2))}

flow(if(S1) S2; else S3;) = flow(S2) U flow(S3) U
{(Label(S1),init(S2)),(Label(S1),init(S3))}

Once we have got the flow graph we can find the reverse flow graph as following:

ReverseFlow: Stmt → Powerset(Lab X Lab)

We need the reverse flow graph to formulate the backward analyses. Once the flow graph is obtained generating the reverse flow graph is simple.

ReverseFlow(S) = {(l , m) | (m , l) ϵ flow(S)}

where S is the whole program.

Till now we have generated the control flow information of the program and the program is now uniquely labeled. We can proceed now to take a look at live variable equations. Live variable is an equational approach as I have mentioned earlier so we need 2 types of equations.

Live variable analysis: A variable is live at the exit from a label if there exists a path from the label to a use of the variable that doesn't re-define the variable. The Live Variable Analysis will determine:

For each program point, which variables may be live at the exit from the point.

This analysis might be used as the basis of Dead Code elimination. If the variable is not live at the exit from a label then, if the elementary block is an assignment to the variable, the elementary block can be eliminated. Many simple statements can be put in a block and the whole block can be assigned a unique label [4].In the approach of this report the simple statements are not kept in a block. Each statement is assigned a unique label.

In Live Variable Analysis we generate Kill and Gen expressions corresponding to each labels. The following table gives the mathematical formulations to generate Kill and Gen expressions.

For each unique label in the program Kill and Gen expressions are computed.

KILL FUNCTIONS

KILL(x = a;) = {x}
KILL(S1) = Ø //S1 is not an assignment statement

GEN FUNCTIONS

GEN(x = a;) = FV(a)
GEN(S1) = FV(S1) //S1 is not an assignment statement

Once the KILL and GEN equations for each unique label of the program is computed the next work is to find the data flow equations. There should be one equation for LVentry and one equation for LVexit.

For each label of the program we have to find LVentry and LVexit.So the equations are written for a label l.

LVexit(l) = Ø if l ϵ final(S) //l is the last label of the program S
U{LVentry(m) | (m , l) ϵ ReverseFlow(S)}

LVentry(l) = (LVexit(l)\KILL(l))UGEN(l)

Live Variable Analysis is a backward analysis so we have to iterate through the reverse flow graph to solve the data flow equations. An example will be provided that shows the labels of the program and the kill and gen sets. To solve the dataflow equations two algorithms are used normally MFP and MOP [1]. Both iterate through the control flow graph until a fixed point is reached. The main aim is to reduce the number of iteration through the control flow graph and for that I have proposed the two methods using Binary Decision Diagrams. They are the following :

a)Extension of Chaotic iteration using BDDs.
b)Global BDD solver to compute fixed points.

Both the methods will be explained completely after the theory section.

2. Binary Decision Diagrams and their use in Program Analysis [6]

As I have stated earlier Binary decision diagram are used to represent a set of binary strings of length n that maps a string of length n to 1 if they are in the set and maps the string to 0 if they are not in the set. So BDD are used to represent Boolean expressions that have a value either 0 or 1.
To understand binary decision diagrams we should have an idea of the following:

a)Boolean Expressions:

The classical calculus for dealing with truth values consists of Boolean variables x, y, ... , the constants true 1 and false 0 the operators of conjunction $\wedge$, disjunction $\vee$, negation $\neg$, implication $\Rightarrow$ and bi-implication $\Leftrightarrow$ which together form the Boolean expressions .Sometimes the variables are called propositional variables or propositional letters and the Boolean expressions are then known as Propositional Logic. Formally, Boolean expressions are generated from the following grammar

$$t ::= x \mid 0 \mid 1 \mid \neg t \mid t \wedge t \mid t \vee t \mid t \Rightarrow t \mid t \Leftrightarrow t,$$

where x ranges over a set of Boolean variable .This is called the abstract syntax of Boolean expressions .The concrete syntax includes parentheses to solve ambiguities .Moreover, as a common convention it is assumed that the operators bind according to their relative priority .The priorities are, with the highest first:

$$\neg , \wedge , \vee , \Rightarrow , \Leftrightarrow$$

Hence for example,

$$\neg x1 \wedge x2 \vee x3 \Rightarrow x4 = (((\neg x1) \wedge x2) \vee x3) \Rightarrow x4$$

A Boolean expression with variables x1,……,xn denotes for each assignment of truth values to the variables itself a truth value according to the standard truth tables see figure 1 .Truth assignments are written as sequences of assignments of values to variables eg. 0/x1,1/x2,0/x3,1/x4 which assigns 0 to x1 and x3,1 to x2 and x4.With this particular truth assignment the above expression has value 1,whereas [0/x1,1/x2,0/x3,0/x4] yields 0.

	$\neg$
0	1
1	0

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\Rightarrow$	0	1
0	1	1
1	0	1

$\Leftrightarrow$	0	1
0	1	0
1	0	1

Figure 1: Truth Tables

b) Normal forms:

A Boolean expression is in *Disjunctive Normal Form (DNF)* if it consists of a disjunction of conjunctions of variables and negations of variables, i.e., if it is of the form

$$(t_1^1 \wedge t_2^1 \wedge \cdots \wedge t_{k_1}^1) \vee \cdots \vee (t_1^l \wedge t_2^l \wedge \cdots \wedge t_{k_l}^l) \qquad (1)$$

where each t_i^j is either a variable x_i^j or a negation of a variable $\neg x_i^j$.

A more succinct presentation of (1) is to write it using indexed versions of $\wedge$ and $\vee$:

$$\bigvee_{j=1}^{l} \left(\bigwedge_{i=1}^{k_j} t_i^j \right).$$

Similarly, a *Conjunctive Normal Form (CNF)* is an expression that can be written as

$$\bigwedge_{j=1}^{l} \left(\bigvee_{i=1}^{k_j} t_i^j \right)$$

c) Binary Decision Diagrams

Let $x \rightarrow y_0, y_1$ be the *if-then-else* operator defined by

$$x \rightarrow y_0, y_1 \;=\; (x \wedge y_0) \vee (\neg x \wedge y_1)$$

hence, $t \rightarrow t_0, t_1$ is true if t and t_0 are true or if t is false and t_1 is true. We call t the *test expression*. All operators can easily be expressed using only the if-then-else operator and the constants 0 and 1. Moreover, this can be done in such a way that all tests are performed only on (un-negated) variables and variables occur in no other places. Hence the operator gives rise to a new kind of normal form. For example, $\neg x$ is $(x \rightarrow 0, 1)$, $x \Leftrightarrow y$ is $x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 0, 1)$. Since variables must only occur in tests the Boolean expression x is represented as $x \rightarrow 1, 0$.

An *If-then-else Normal Form (INF)* is a Boolean expression built entirely from the if-then-else operator and the constants 0 and 1 such that all tests are performed only on variables.

If we by $t[0/x]$ denote the Boolean expression obtained by replacing x with 0 in t then it is not hard to see that the following equivalence holds:

$$t = x \to t[1/x], t[0/x]. \tag{2}$$

This is known as the *Shannon expansion* of t with respect to x. This simple equation has a lot of useful applications. The first is to generate an INF from any expression t. If t contains no variables it is either equivalent to 0 or 1 which is an INF. Otherwise we form the Shannon expansion of t with respect to one of the variables x in t. Thus since $t[0/x]$ and $t[1/x]$ both contain one less variable than t, we can recursively find INFs for both of these; call them t_0 and t_1. An INF for t is now simply

$$x \to t_1, t_0.$$

So till now we know that every Boolean expression can be equivalently represented as an expression in INF. Let's see an example of making a binary decision diagram given an Boolean expression.

Example: Consider the Boolean expression t = (x $\Leftrightarrow$ y) $\wedge$ (x $\Leftrightarrow$ y). If we find an INF of t by selecting in order the variables x1, y1 , x2 , y2 on which to perform Shannon expansions , we get the expressions

$$\begin{aligned}
t &= x_1 \to t_1, t_0 \\
t_0 &= y_1 \to 0, t_{00} \\
t_1 &= y_1 \to t_{11}, 0 \\
t_{00} &= x_2 \to t_{001}, t_{000} \\
t_{11} &= x_2 \to t_{111}, t_{110} \\
t_{000} &= y_2 \to 0, 1 \\
t_{001} &= y_2 \to 1, 0 \\
t_{110} &= y_2 \to 0, 1 \\
t_{111} &= y_2 \to 1, 0
\end{aligned}$$

Figure 2 shows the expression as a tree. Such a tree is also called a decision tree. A lot of the expressions are easily seen to be identical,so it is tempting to identify them. For example, instead of t110 we can use t000 and instead of t111 we can use t001.If we substitute t000 for t110 in the right_hand side of t11 and also t001 for t111 we in fact see that t00 and t11 are identical and in t1 we can replace t11 with t00.

If we in fact identify all equal subexpressions we end up with what is known as a binary decision diagram (a BDD),It is no longer a tree of Boolean expressions but a directed acyclic graph DAG.

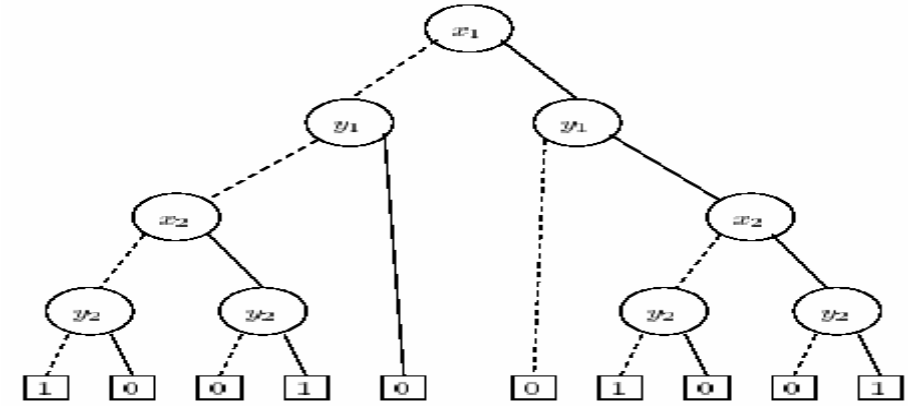


Figure 2: A decision tree for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. Dashed lines denote low-branches, solid lines high-branches.

Applying this idea of sharing, t can now be written as:

$$
\begin{array}{rcl}
t & = & x_1 \longrightarrow t_1, t_0 \\
t_0 & = & y_1 \longrightarrow 0, t_{00} \\
t_1 & = & y_1 \longrightarrow t_{00}, 0 \\
t_{00} & = & x_2 \longrightarrow t_{001}, t_{000} \\
t_{000} & = & y_2 \longrightarrow 0, 1 \\
t_{001} & = & y_2 \longrightarrow 1, 0
\end{array}
$$

Each subexpression can be viewed as the node of a graph. Such a node is either terminal in the case of the constants 0 and 1 or non-terminal. A non- terminal node has a low-edge corresponding to the else-part and a high-edge corresponding to the then-part. See figure 3.Notice, that the number of nodes has decreased from 9 in the decision tree to 6 in the BDD. It is not hard to imagine that if each of the terminal nodes were other big decision trees the savings would be dramatic. Since we have chosen to consistently select variables in the same order in the recursive calls during the construction of the INF of t, the variables occur in the same orderings on all paths from the root of the BDD. In this situation the binary decision diagram is said to be ordered (an OBDD).Figure 3 shows a BDD that is also an OBDD.

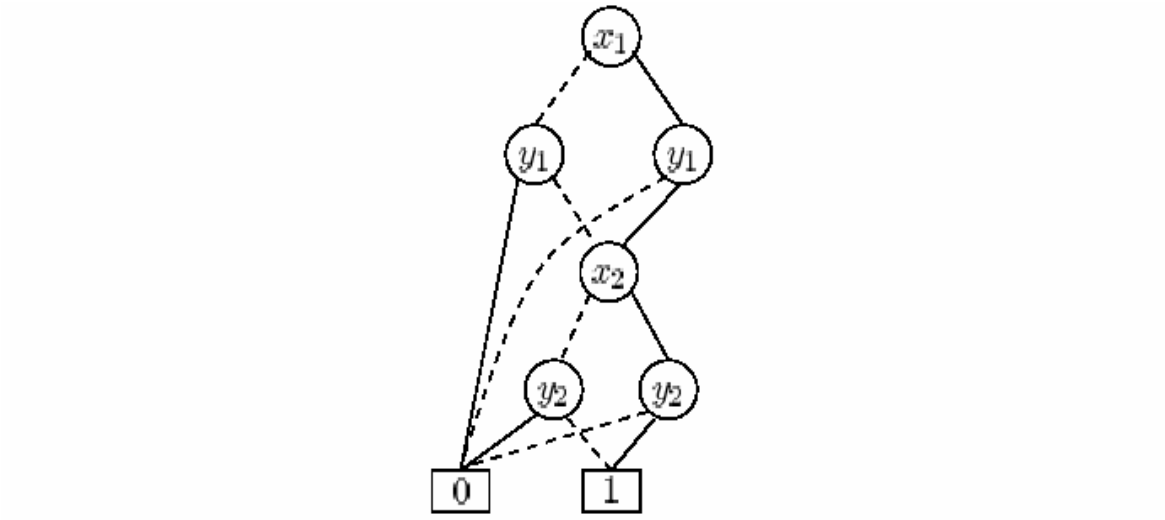


Figure 3: A BDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ with ordering $x_1 < y_1 < x_2 < y_2$. Low-edges are drawn as dotted lines and high-edges as solid lines.

If all identical nodes are shared and all redundant tests are eliminated, the
OBDD is said to be reduced (an ROBDD).Often when people speak about BDDs they really mean ROBDDs. To summarize:

A *Binary Decision Diagram (BDD)* is a rooted, directed acyclic graph with

- one or two terminal nodes of out-degree zero labeled 0 or 1, and
- a set of variable nodes u of out-degree two. The two outgoing edges are given by two functions *low*(u) and *high*(u). (In pictures, these are shown as dotted and solid lines, respectively.) A variable *var*(u) is associated with each variable node.

A BDD is *Ordered* (OBDD) if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < \cdots < x_n$. An (O)BDD is *Reduced* (R(O)BDD) if

- **(uniqueness)** no two distinct nodes u and v have the same variable name and low- and high-successor, i.e.,

  $$var(u) = var(v), low(u) = low(v), high(u) = high(v) \text{ implies } u = v,$$

  and
- **(non-redundant tests)** no variable node u has identical low- and high-successor, i.e.,

  $$low(u) \neq high(u).$$

ROBDDs have some interesting properties. They provide compact representations of Boolean expressions and there are efficient algorithms for performing all kinds of logical operations on ROBDDs. They are all based on the crucial fact that for any function:

$$f : \mathbb{B}^n \longrightarrow \mathbb{B}$$

there is exactly one ROBDD representing it. This means, in particular, that there is exactly one ROBDD for the constant true (and constant false) function on $\mathbb{B}^n$: the terminal node 1(and 0 in case of false).Hence , it is possible to test in constant time whether an ROBDD is constantly true or false. Recall that for Boolean expressions this problem is NP-complete.

There are several efficient algorithms to implement all the possible set theoretic operations on ROBDDs. The above description will provide the reader with an insight of what are BDDs in particular ROBDDs. To know about the algorithms that manipulate the ROBDDs go through ref [6]. There are several packages that construct and manipulates BDDs. I have used BuDDy package for developing the solver. I will describe in detail how to use BuDDy package to do all sorts of manipulations on ROBDDs and most important of all FDDs (finite domain BDDs) which are used in constructing and manipulating relations. Several other packages are there. Some links for the packages are given below:

- Fabio Somenzis's CUDD from Colorado University.
- Armin Biere's ABCD package
- Geert Janssen's BDD package from Eindhoven University of Technology.
- Carnegie Mellon's BDD package.
- The CAL package from Berkeley.
- K. Milvang-Jensen's parallel package BDDNOW .
- Bwolen Yang's PBF package.
- Projects for integrating BuDDy with both Moscow ML and New Jersey ML have resulted in really efficient BDD operations in ML. See the MuDDy page.

3. Points-to Analysis Using Binary Decision Diagram: Sable research group [2] has implemented the solver for points-to analysis using Binary Decision Diagrams. Points-to analysis has been developed by Anderson for C programming language and later it was extended to Java. The BDD solver developed by sable research group found the solution for points-to Analysis for Java. Let us understand about the points-to Analysis using a small program segment written in Java.

```
A: a = new O();
B: b = new O();
C: c = new O();
   a = b;
   b = a;
   c = b;
```

The points-to relation we would compute for this code is*{(a,A), (a,B), (b,A), (b,B), (c,A), (c,B), (c,C)}*, where (*a*,*A*) indicatesthat variable *a* may point to objects allocated at allocation site A.Using 00 to represent a and A, 01 to represent b and B, and 10 to represent c and C, we can encode this points-to relation using the set {0000,0001,0100,0101,1000,1001,1010}.Figure 2(a) shows an unreduced BDD representing this set where the variables *a*, *b* and *c* are encoded at BDD node levels *V*0 and *V*1. and the heap objects *A*, *B* and *C* are encoded at the *H*0 and *H*1 levels. As a convention, 0-successors are indicated by dotted edges and 1-successors are indicated by solid edges.
Notice that nodes marked x, y, and z in Figure 2(a) are at the same level and have the same 0- and 1-successors. This is because they represent the subset *{A,B}*, which is shared by all three program variables. Because they are at the same level and share the same successors, they could be merged into a single node, reducing the size of the BDD. Furthermore, since their two successors are the same (the 1 node), their successor does not depend on the bit being tested, so the nodes could be removed entirely. Simplifying other nodes in this manner, we get the BDD in Figure 2(b). The BDD with the fewest nodes is unique if we maintain a consistent ordering of the nodes; it is called a *reduced* BDD. When BDDs are used for computation, they are always kept in a reduced form. In the examples so far, the bits of strings were tested in the order in which they were written. However, any ordering can be used, as long as it is consistent over all strings represented by the BDD. For example, Figure 2(c) shows the BDD that represents the same relation, but tests the bits in a different order. This BDD requires 8 nodes, rather than 5 nodes as in Figure 2(b). In general, choosing a bit ordering which keeps the BDDs small is very important for efficient computation; however, determining the optimal ordering is NP-hard [23]. BDDs support the usual set operations (union, intersection, complement, difference) and can be maintained in reduced form during each operation. A binary operation on BDDs *A* and *B*, such as AUB, takes time proportional to the number of nodes in the BDDs representing the operands and result. In the worst case, the number of nodes in the BDD representing the result can be the product of the number of nodes in the two operands, but in most cases, the reduced BDD is much smaller [6].

Sable research group has also used BuDDy to code the BDDs. BuDDy[7] is one of several publicly-available BDD packages.Instead of requiring the programmer to manipulate individual bit positions in BDDs, `BuDDy` provides an interface for grouping bit positions together. The term *domain* is used to refer to such a group.In the example in Figure 2, domain *V* is used to represent variables, and *H* to represent pointed-to heap locations.

BuDDy supports several important operations like existential quantification and relational product.The definition of existential quantification and relational product are given below:

i)Existential quantification: given a points-to relation $P \subseteq V * H$, we can existentially quantify over *H* to find the set *S* of variables with non-empty points-sets:

$$S = \{v \mid \exists h.(v,h) \in P\}$$

ii)Relational product: The *relational product* operation implemented in `BuDDy` composes set intersection with existential quantification, but is implemented more efficiently than these two operations composed. Specifically,

$$relprod(X,Y,V1) = \{(v2,h) \mid \exists v1.((v1,v2) \in X \wedge (v1,h) \in Y)\}$$

To illustrate this with an example, for the code fragment Figure 1, consider the initial points-to set *{*(*a*,*A*), (*b*,*B*), (*c*,*C*)*}* corresponding to the first three lines of code) and the assignment edge set *{*(*b*,*a*), (*a*,*b*), (*b*,*c*)*}* (corresponding to the last three lines code). The pair (*a*,*b*) corresponds to the statement *b* := *a*; that we write the variables in reverse order, indicating that all allocation sites reaching *a* also reach *b*. The initial points-to set is represented in the BDD in Figure 3(a) using the domains *V*1 and *H*.

The edge set contains pairs of variables, so *two* variable domains (*V*1 and *V*2) are required to represent it; its representation is shown in Figure 3(b). Given these two BDDs, we can apply the relational product with respect to *V*1 to obtain the BDD of the points-to sets after propagation along the edges (Figure 3(c)), using the domains *V*2 and *H*.

The *replace* operation creates a BDD in which information that was stored in one domain is moved into a different domain. For example, we would like to find the union of the points-to relations in parts (a) and (c) of Figure 3, but the former uses the domains *V*1

and *H*, while the latter uses *V*2 and *H*.

Before finding the union, we applying the replace operation to (c) to obtain (d), which, like (a), uses domains *V*1 and *H*. We can now find (e)=(a)∪(d), the points-to set after one step of propagation. If we repeated these steps a second time, we would obtain the final points-to set BDD from Figure 2(b).Note that it is possible for a BDD for a large set to have fewer nodes than the BDD for a smaller set. In this case, although the points-to set grows from three, to six, to seven pairs, the BDD representing it goes from eight to six to five nodes (see Figures 3(a), 3(e), and 2(b), respectively).

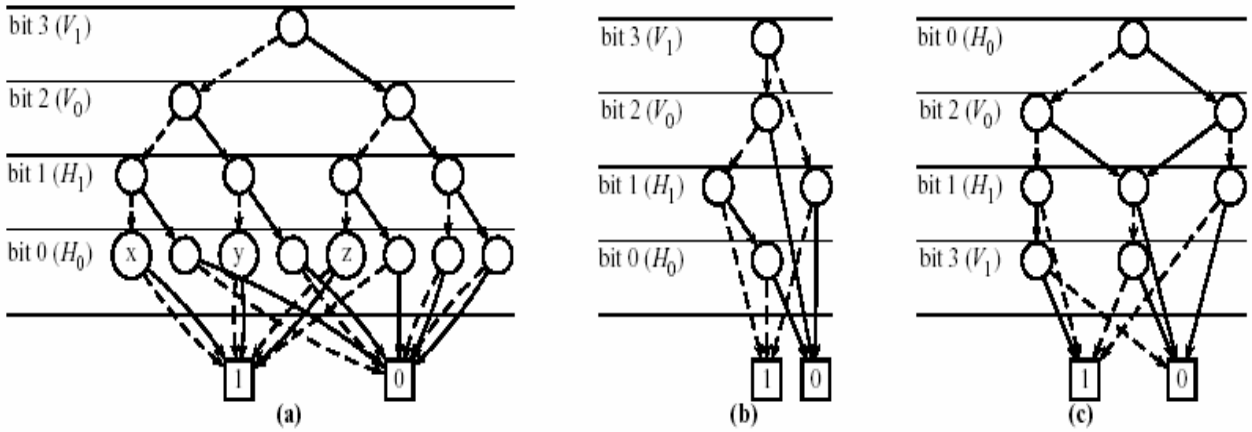


Figure 2: BDDs for points-to relation $\{(a,A),(a,B),(b,A),(b,B),(c,A),(c,B),(c,C)\}$ (a) unreduced using ordering $V_1V_0H_1H_0$, (b) reduced using ordering $V_1V_0H_1H_0$, (c) reduced using alternative ordering $H_0V_0H_1V_1$

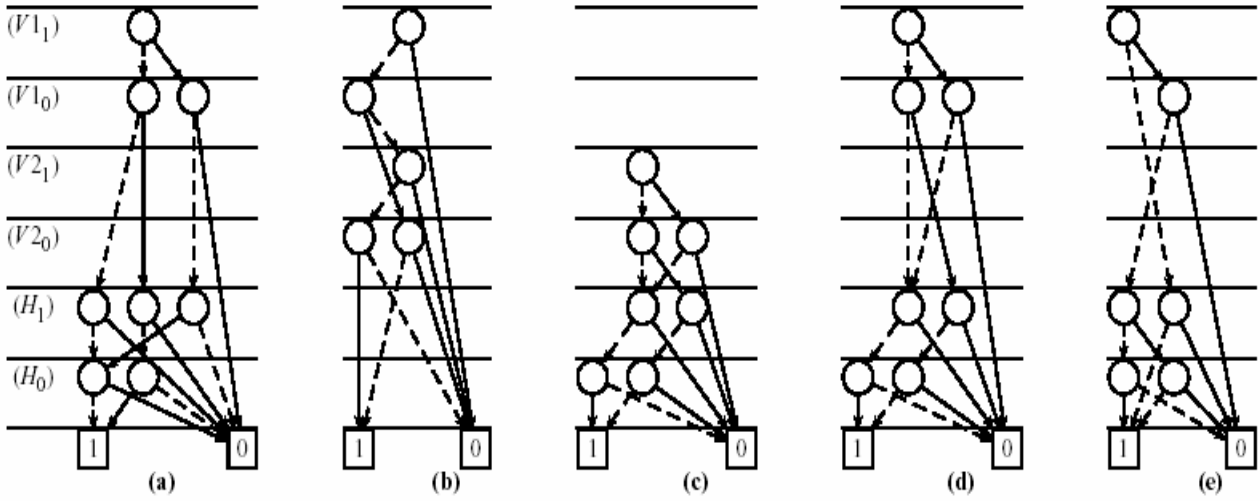


Figure 3: (a) BDD for initial points-to set $\{(a,A),(b,B),(c,C)\}$ (b) BDD for edge set $\{(a \rightarrow b),(b \rightarrow a),(b \rightarrow c)\}$ (c) result of relprod((a),(b),V1) (the points-to set $\{(a,B),(b,A),(c,B)\}$) (d) result of replace((c),V2ToV1) (e) result of (a)∪(d) (the points-to set $\{(a,A),(a,B),(b,A),(b,B),(c,B),(c,C)\}$

Implementation details: This sections describes all the implementation details as well as explains the 2 approaches that I have developed to solve live variable analysis. The implementation details are divided into 3 phases :

1. Phase I (initialization): In this phase I have developed tools to automate the generation of kill and gen sets and uniquely labelize the program. Also syntax based control flow information is obtained to generate the control flow graph. This phase creates the base to do the live variable analysis.

2. Phase II: In this phase I transformed the live variable analysis to BDD domain and developed a solver which can be called an extension to chaotic iteration. BDDs reduced the complexity to find LVentry and LVexit sets but doesn't reduces the number of iterations of control flow graph to generate the fixed points. This approach have almost the same complexity as the traditional solvers like worklist algorithm and chaotic iteration.

3. Phase III: In this phase I have developed a global bdd solver which captures the kill and gen information globally not as the traditional approaches which captures kill and gen information for each label. Given appropriate ordering it has the capability to reduce the number of iterations over the control flow graph.

Source Language: To do the live variable analysis programs written in C-- are used. C-- is a small subset of C language which doesn't have structures and pointers. The live variable analysis framework took a program written in C-- and then generates the data structures needed to do the analysis. A complete description of C-- can be found at [5].The language constructs supported by C-- are the following :

i) assignment statement.
ii) conditional statement (if statement and if-else statement)
iii) iteration statement (while).

Only integer datatype is supported by C-- language.
Syntax of C-- is exactly the same C. Only for and switch was not supported since they can be easily implemented by while and if-else statements. An example of valid C-- program is given below :

```
void main( )
{
    int x;
    int y,z;
    z=1;
    y=2;
    x=y+z;
    while(x>y)
    {
        if(y>z)
            y=y-1;
        else
            y=z-1;
    }
}
```

Hence programs written in C-- language will be accepted by the BDD framework for program analysis. After this introduction to source language used for the analysis problem I will now provide details of the work done at each phase. Details of each phase is given below:

Phase I: Phase I includes the following steps:-

1. Development of lexical analyzer for the language: The lexical analyzer identifies each token from the program written in C--.The lexical analyzer was written in lex. After lexical analyzer returns each token to the parser, it matches them with the grammer defined for C-- and verifies if the program is written according to C-- grammer or not. The grammer for C-- is given in ref [5].

2. Development of Parser for the language: To match whether the program is written according to the C-- grammer or not a parser for C-- is used. The parser is written using yacc. After generating the C code for lexer and parser, the code is modified to store all the tokens in a data structure called token_table. This data structure contains 2 types of information :

a) The name of the token

b) The unique identifier of the token

For example : if there is a variable declaration called int x; then the name of the token will be x and the unique identifier will be var. This token_table data structure will be used by the next module which is called dataflow. Dataflow module does the following

Steps:

a)It labelizes the program completely.

b)It finds the number of variables in the program

c)It finds the kill and gen sets which are the parameter for the live variable analysis.

d)It finds the control flow information to generate the control flow graph.

3. Development of dataflow module: Dataflow module generates all the necessary parameters to make live variable analysis possible. As I have already mentioned in the previous step it does all the steps given above. Let's see how the module does the above mentioned operations :

a) Dataflow module uses the information stored in the token_table data structure. Since now whole information about the program is stored in token_table data structure.

b) Dataflow module recursively labelizes the program and finds kill and gen information. From the token_table it extract a whole statement (like C in C-- also the statement terminates with a semicolon).The number of tokens in the program are stored globally. The function that extracts the information is makedflist who makes the dataflow list(in short dflist)

c) After extracting a whole statement it identifies the type of statement it corresponds to. Remember the following type of statements are supported by C-- :

i) assignment.

ii) conditional (both if & if-else)

iii) iteration (while iteration is supported)

d) After identifying the type of statement it calls the handler for the appropriate expression. The handler for each type of expressions then checks what to do: it means that if "while" handler is called it gives a unique label to the condition checking statement in the while loop and finds the Kill and Gen sets. Kill and Gen sets are computed according to the mathematical definitions specified in theory part. After completing it's work the handler checks if some more tokens are remains to be tested or not(remember that the handler knows how many tokens are there in the given program).

Similar handlers are written for "else", "while" and "assignment" statement.

e) If some more tokens are left to be checked then it calls again the main function(which is makedflist) and recursively it again calls the appropriate handler.

f) As it finds the kill and gen sets it also finds the control flow information from the syntax of the program. The flow information states that from label l the control can go to which labels. The program will generate the control flow information correctly only if certain constraints are applied to the program. The constraints are the following :

1. Use structured code only i.e if ,else and while statements should be accompanied with curly braces.

2. The Control Flow Graph should be obtained from parse tree generated from the program. In that case no constraints are needed to be imposed on the syntax of the program. The next version of the code will generate the CFG using that approach.

3.The CFG can also be generated easily if we have the three address code (TAR code) or some sort of intermediate code. This approach is used in several text books of compiler construction ref [4]. Three-address-code is useful since it contains only JUMP statement for both the conditional and while statements that changes the control flow of the program. Thus either of one method (2 or 3) will be used to generate the control flow graph in subsequent refinements of the source code.

g) The dataflow module extracts the information needed for live variable analysis and stores it in a structure called df. A df structure contains the following entries:

i) Unique label
ii) kill set : It contains all the identifiers of the variables that are killed at that particular label terminated by -1.
iii) gen set : It contains all the identifiers of the variables that are generated at that particular label also terminated by -1.
iv) flow set : It contains the flow information. The flow information is defined earlier.
v) It also contains some more variable that will be used in next modules. It includes:
    a) kill bdd for each label.
    b) gen bdd for each label.
    c) LVentry bdd which will contain the LVentry information for the first approach of BDD solver described earlier.
    d) LVexit bdd which will contain the LVexit information for the first approach of BDD solver.

It completes the implementation details of Phase I which just prepares the base to proceed for the development of bdd solvers. Phase II incorporates the introduction to BuDDy package to manipulate the BDD and the first approach to develop a bdd solver.

Phase II: Development of BDD solver which is an extension of chaotic iteration (the traditional algorithm) to BDD domain: At this point of time I must give a brief description of BuDDy package which I have used to code the BDDs. Since the rest of the work will be done in BDD domain let's take a look at the BuDDy package to understand the implementation details:

BuDDy package is a complete BDD package developed by Jorn Lind-Nielsen of University of Copenhegan. The package implements all the operations defined on binary decision diagram(and, or, set difference etc.). A complete manual for buddy can be obtained at [7]. BuDDy even supports finite domain blocks which is very useful in encoding the relations. I would describe only the operations that were used by me during implementing the solver. First of all a brief introduction to BuDDy:

BuDDy is a Binary Decision Diagram package that provides all of the most used functions for manipulating BDDs. The package also includes functions for integer arithmetics such as addition and relational operators. BuDDy started as a technology transfer project between the Technical University of Denmark and Bann Visualstate. The later is now using the techniques from BuDDy in their software. The following description will give only the details of the C interface of BuDDy. Theory about the BDDs are briefly provided in theory section and can be found in ref [6].

Installing and compiling BuDDy:

1.To install the BuDDy package just obtain the gunzipped package from http://www.itu.dk/research/buddy.
2. Then unzip the package.
3. In the buddy22 folder there will be a configuration file and a make file.
4. I have installed it on SPARC solaris machine so I have done the following changes in config file:
    i) Change the INCDIR to the directory where you want to store the header files for running the programs using the BuDDy package.
    ii) Change the LIBDIR to the directory where you want to store the BuDDy library.
5. After doing these changes compile the package by giving make command.
6. Make install will copy the bdd library to the destination specified in the configuration file and the header files to the include directory specified.
7. Now to compile a program written using BuDDy use the following command:

```
gcc -I ~/include/ -L ~/lib myfile.c -o myfile -lbdd -lm
```

These are the steps which are needed to install the BuDDy package and to compile a program written using BuDDy BDDs.

Now let's take a brief look at the BuDDy functions used during the implementation of the BDD solver:

1. bdd_allsat
Finds all satisfying variable assignments.
Description:
Iterates through all legal variable assignments (those that make the BDD come true) for the bdd r and calls the callback handler handler for each of them. The array passed to handler contains one entry for each of the globaly dened variables. Each entry is either 0 if the variable is false, 1 if it is true, and -1 if it is a don't care. Example of a callback handler written in C:

```
void allsatPrintHandler(char* varset,int size)
{
 int v;
 for(v=0;v<size;v++)
   {
    switch(varset[v])
           {
           case -1:printf("X");
             break;
           case 0:printf("0");
             break;
           case 1:printf("1");
             break;
           }
   }
 printf("\n");
}
```

The handler can be used like this: bdd_allsat(r, allsatPrintHandler);
See also bdd_satone, bdd_satoneset

2. bdd_and : The logical 'and' of two BDDs
BDD bdd_and(BDD l, BDD r)
Description
This a wrapper that calls bdd apply(l,r,bddop and).
Return value
The logical 'and' of l and r.
See also
bdd_apply, bdd_or

3. bdd apply : It does basic bdd operations
BDD bdd_apply(BDD left, BDD right, int opr)
Description:
The bdd apply function performs all of the basic bdd operations with two operands, such as AND, OR etc. The left argument is the left bdd operand and right is the right operand. The opr argument is the requested operation and must be one of the following:

Identifier	Description	Truth table	C++ opr.
bddop_and	logical and ($A \wedge B$)	[0,0,0,1]	&
bddop_xor	logical xor ($A \oplus B$)	[0,1,1,0]	^
bddop_or	logical or ($A \vee B$)	[0,1,1,1]	|
bddop_nand	logical not-and	[1,1,1,0]	
bddop_nor	logical not-or	[1,0,0,0]	
bddop_imp	implication ($A \Rightarrow B$)	[1,1,0,1]	>>
bddop_biimp	bi-implication ($A \Leftrightarrow B$)	[1,0,0,1]	
bddop_diff	set difference ($A \setminus B$)	[0,0,1,0]	-
bddop_less	less than ($A < B$)	[0,1,0,0]	<
bddop_invimp	reverse implication ($A \Leftarrow B$)	[1,0,1,1]	<<

Return value

The result of the operation.

4. bdd_delref: decreases the reference count on a node
BDD bdd_delref(BDD r)
Description
Reference counting is done on externaly referenced nodes only and the count for a specific node r can and must be decreased using this function to make it possible to reclaim the node in the next garbage collection.
Return value
The BDD node r.
See also
Bdd_addref

5. bdd_done:resets the bdd package
void bdd_done(void)
Description
This function frees all memory used by the bdd package and resets the package to it's initial state.
See also
bdd_init

6. bdd_false: returns the constant false bdd
BDD bdd_false(void)
Description
This function returns the constant false bdd and can freely be used together with the
bddtrue and bddfalse constants.
Return value
The constant false bdd
See also
bdd_true, bdd_true, bdd_false

7. bdd_init { initializes the BDD package
int bdd_init(int nodesize, int cachesize)
Description
This function initiates the bdd package and must be called before any bdd operations are done. The argument nodesize is the initial number of nodes in the nodetable and cachesize is the fixed size of the internal caches. Typical values for nodesize are 10000 nodes for small test examples and up to 1000000 nodes for large examples. A cache size of 10000 seems to work good even for large examples, but lesser values should do it for smaller examples. The number of cache entries can also be set to depend on the size of the nodetable using a call to bdd setcacheratio. The initial number of nodes is not critical for any bdd operation as the table will be resized whenever there are to few nodes left after a garbage collection. But it does have some impact on the efficiency of the
operations.
Return value
If no errors occur then 0 is returned, otherwise a negative error code.
See also
bdd_done

8. bdd_newpair: creates an empty variable pair table
bddPair *bdd_newpair(void)
Description
Variable pairs of the type bddPair are used in bdd replace to define which variables to replace with other variables. This function allocates such an empty table. The table can be freed by a call to bdd freepair.
Return value
Returns a new table of pairs.
See also
bdd_freepair, bdd_replace, bdd_setpair

9. bdd_nithvar: returns a bdd representing the negation of the I'th variable
BDD bdd_nithvar(int var)
Description
This function is used to get a bdd representing the negation of the I'th variable (one node with the children false and true). The requested variable must be in the range dene by bdd setvarnum starting with 0 being the rest. For ease of use then the bdd returned from bdd nithvar does not have to be referenced counted with a call to bdd addref.
Return value
The negated I'th variable on succes, otherwise the constant false bdd
See also
bdd_ithvar, bddtrue, bddfalse

10. bdd_not: negates a bdd
BDD bdd_not(BDD r)
Description
Negates the BDD r by exchanging all references to the zero-terminal with references to the one-terminal and vice versa.
Return value
The negated bdd.

11. bdd_or: The logical 'or' of two BDDs
BDD bdd_or(BDD l, BDD r)
Description
This a wrapper that calls bdd apply(l,r,bddop or).
Return value
The logical 'or' of l and r.
See also
bdd_apply, bdd_and

12. bdd_pathcount: count the number of paths leading to the true terminal
double bdd_pathcount(BDD r)
Description
Counts the number of paths from the root node r leading to the terminal true node.
Return value
The number of paths
See also
bdd_nodecount

13. bdd_printdot: prints a description of a BDD in DOT format
void bdd_printdot(BDD r)
int bdd_fnprintdot(char* fname, BDD r)
void bdd_fprintdot(FILE* ofile, BDD r)
Description
Prints a BDD in a format suitable for use with the graph drawing program DOT to either stdout, a designated le ofile or the le named by fname. In the last case the file will be opened for writing, any previous contents destroyed and then closed again. Important: I have used Graphviz software (DOT) by AT&T to visualize the BDDs.

14. bdd_replace { replaces variables with other variables
BDD bdd_replace(BDD r, bddPair *pair)
Description
Replaces all variables in the BDD r with the variables defined by pair. Each entry in pair consists of a old and a new variable. Whenever the old variable is found in r then a new node with the new variable is inserted instead.
Return value
The result of the operation.

See also
bdd_newpair

15. bdd_setpair: set one variable pair
int bdd_setpair(bddPair *pair, int oldvar, int newvar)
int bdd_setbddpair(bddPair *pair, BDD oldvar, BDD newvar)
Description
Adds the pair (oldvar,newvar) to the table of pairs pair. This results in oldvar being substituted with newvar in a call to bdd replace. In the first version newvar is an integer representing the variable to be replaced with the old variable. In the second version oldvar is a BDD. In this case the variable oldvar is substituted with the BDD newvar. The possibility to substitute with any BDD as newvar is utilized in bdd compose, whereas only the topmost variable in the BDD is used in bdd replace.
Return value
Zero on success, otherwise a negative error code.
See also
bdd newpair

16. bdd_true: returns the constant true bdd
BDD bdd_true(void)
Description
This function returns the constant true bdd and can freely be used together with the
bddtrue and bddfalse constants.
Return value
The constant true bdd
See also
bdd_false, bddtrue, bddfalse

Some BuDDy operations for finite domain are given below that will be used in Phase III to develop the global bdd solver.

17. fdd_extdomain: adds another set of finite domain blocks
int fdd_extdomain(int *dom, int num)
Description
Extends the set of finite domain blocks with the num domains in dom. Each entry in dom defines the size of a new finite domain which later on can be used for finite state machine traversal and other operations on finite domains. Each domain allocates log 2 (jdom[i]j) BDD variables to be used later. The ordering is interleaved for the domains defined in each call to bdd extdomain. This means that assuming domain D0 needs 2 BDD variables x1 and x2 , and another domain D1 needs 4 BDD variables y1 ; y2 ; y3 and y4 , then the order will be x1 ; y1 ; x2 ; y2 ; y3 ; y4 . The index of the first domain in dom is returned. The index of the other domains are offset from this index with the same offset as in dom. The BDD variables needed to encode the domain are created for the purpose and do not interfere with the BDD variables already in use.
Return value
The index of the first domain or a negative error code.
See also
fdd_ithvar

18. BDD fdd_ithvar(int var, int val)
Description:
Returns the BDD that defines the value val for the finite domain block var. The encoding
places the Least Signicant Bit at the top of the BDD tree (which means they will have
the lowest variable index). The returned BDD will be V 0 ^ V 1 ^: : : ^ VN where each V i will be in positive or negative form depending on the value of val.
Return value
The correct BDD or the constant false BDD on error.

19. fdd_setpair: Defines a pair for two finite domain blocks
int fdd_setpair(bddPair *pair, int p1, int p2)
Description
Defines each variable in the finite domain block p1 to be paired with the corresponding variable in p2. The result is stored in pair which must be allocated using bdd makepair.
Return value
Zero on success or a negative error code on error.

This completes a brief introduction to BuDDy package and the various functions which I have used to model the binary decision diagrams. Now let's see the first approach i.e. extension of chaotic iteration to BDD domain. To understand it properly let's take an example C-- code. The number of unique variables in the program are obtained by the module developed in Phase I. For each unique variable a varid is assigned.

```
//hypothetical start node: 0
{
   int x,y,z;
   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
```

List of variables in program segment
(Generated by the tool)
variable : (name,id) : (x,0)
variable : (name,id) : (y,1)
variable : (name,id) : (z,2)

Phase I module will generate the kill and gen sets for each label along with the control flow information. The kill and gen sets along with the control flow information is given below:

Label	Kill list	Gen list	Flow nodes
0	(#,#)	(#,#)	
1	(x,0)	(#,#)	0
2	(y,1)	(#,#)	1
3	(z,2)	(x,0)	2
4	(#,#)	(x,0)(z,2)	3
5	(#,#)	(y,1)(z,2)	4
6	(x,0)	(y,1)	5
7	(z,2)	(x,0)	6
8	(x,0)	(y,1)(z,2)	5 4
9	(x,0)	(x,0)(y,1)	8
10	(#,#)	(#,#)	

In this version of the code the automated generation of CFG was n't done. The subsequent version of the code will generate the CFG automatically. The control flow graph of the C-- program is given below:

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

6->7
7->4, 9
8->4, 9
9->10
10

CFG is given as a pair of tuple which shows that the control flow from label l1 to label l2 when there is an arrow between two labels like 0->1.For example 4->5, 9 conveys the information that the control flows from label 4 to label 5 and label 9 in the C-- source program.

Now let's take a look at the steps of the algorithm which is the extension of chaotic iteration:

1. First of all make a bdd variable for each of the unique variable defined in the program. In the example program above three variables are to be made each corresponding to one variable. So we will have a bdd x ,bdd y and bdd z.
2. Now find the kill and gen bdd for each label from the kill and gen bdd sets defined for each label. If two variables are generated then take the union of 2 bdd variables to represent that set. For example for label 4 there are two entries in the gen set {(x,0) , (z,2)}.Hence the gen bdd of label 4 will contain the union of bdd variables x and z. Thus the gen bdd will be calculated by :

gen bdd = (bdd x) U (bdd z)

3. After step 2 we will have kill and gen bdd for each label. After that start solving the live variable equations. For each label LVentry and LVexit BDD will be defined. For the last label LVexit BDD will be false. Starting from the last label propagate the generation upwards. If there are 3 labels in the program namely 0 ,1 ,2 then generate the LVexit and LVentry BDDs in the following way:

generate LVexit(2) then generate LVentry(2)
after that generate LVexit(1) then generate LVentry(1)
after that generate LVexit(0) then generate LVentry(0)

4. Make a universal BDD U which will be the union of all the variable bdd defined in the program.
5. Initially make the LVentry and LVexit BDD false for every label.
6. Let's take a look at the equations of LVentry and LVexit again and see how the BDD transformation will take place:

$$LVexit(l) = \begin{cases} \emptyset & \text{if } l \in final(S) \text{ //l is the last label of the program S} \\ \cup\{LVentry(m) \mid (m, l) \in ReverseFlow(S)\} & \end{cases}$$

$$LVentry(l) = (LVexit(l) \setminus KILL(l)) \cup GEN(l)$$

From the equations we can see that LVexit will be the union of several LVentry BDDs according to the control flow graph of the program. The problem will come when finding LVexit of a label l we will get an LVentry which is not defined previously i.e a false bdd.

Finding the entry BDD is not a problem if we have exit BDD since entry and exit BDD are related by simple set theoretic operands set difference and union. Kill and Gen BDD for each label is already defined so once we have the LVexit BDD LVentry can be easily computed.

Let's take the example program and see where such a problem will come. Take a look at the 7->4,9 entry of control flow graph. When we will find LVexit(7) then LVexit(9) bdd will be well defined but LVexit(4) BDD will be false. How to deal with this problem ?

7. We have already made an universal BDD U. Whenever such an above mentioned situation arises make the false BDD equivalent to U BDD and then continue the computation.
8. After carrying out the computation we may/mayn't find that the LVexit(4) BDD which is initialized to U will come to some set other than U. If it calculates some set other that U then change the LVexit BDD to the new one and recompute all the LVexit and LVentry BDD. If we find that the BDD is same then that means we have already reached a fixed point. Otherwise the above process on repetition will find the fixed point and the process will terminate when the set computed and the set before the computation will be the same.

Advantages of the algorithm:

a) It gives a new approach of modeling the live variable equations using the BDDs.
b) The set theoretic operations are carried out over the BDDs not on the sets as done in traditional worklist algorithm.

Disadvantages of the algorithm:
a) It changes the data structure and made the operations efficient but it doesn't decrease any iteration over the CFG. The iterations are the same as the worklist algorithm so it doesn't changes the complexity of the original algorithm.

It completes my Phase II details and at this point I have not been able to reduce the number of iterations but only used efficient data structure called BDD. In phase III I have developed a global bdd solver which captures the live variable analysis problem globally and tries the increase the efficiency compared the worklist algorithm and other traditional algorithms to solve the live variable equations.

Phase III: In this phase I have developed a global bdd solver that captures the live variable analysis globally. What does that mean ? Just take a look at the kill and gen sets. Instead of making kill and gen sets for each label we can specify how many label does a variable is related with in all the kill sets. Take a look at the occourance of x in kill set.

Label	kill set
1	(x,0)
6	(x,0)
8	(x,0)
9	(x,0)

From this we can say that globally variable x is killed at labels 1, 6, 8 and 9.Similar such sets can be found out for each of the variables defined in the program for kill and gen sets respectively. It leads to the definition of following relations:

i) globalkill: Which will contain all the variable, label set for the kill set. Hence the globalkill can be defined as the relation of the form Var X Labels/ProgramPoint.
ii) globalgen: globalgen is computed exactly the same way as globalkill except that it is related to gen sets rather than kill sets in globalkill.

A third relation is also defined as globalcfg which captures the control flow graph information. The control flow graph can represented as tuples (l, m) which says that control flows from label l to label m. So control flow graph can be represented by a relation of the form ProgramPoint X ProgramPoint.

Now for a given program 3 relations are defined:

a) globalkill: Var X ProgramPoint
b) globalgen: Var X ProgramPoint
c) globalcfg: ProgramPoint X ProgramPoint

BDDs are quite effective at representing relations specially using the finite domain blocks. Representing relations using BDD are discussed in detail in the perspective of points-to analysis in theory part. Let's take a quick recap:

Relation encoding using BDDs:
Suppose we have to encode a relation which has 3 2-tuples{(0,1),(1,2),(2,3)}.As i have already mentioned bdds are functions f : $B^n \rightarrow B$, each entry of the relation can be encoded using binary bits. In the example relation maximum entry is 3 so it can be
represented by 2 bits.

Using 2 bits : 0 → 00
1 → 01
2 → 10
3 → 11

Now the tuple (0,1) can be represented as 0001
Similarly (1,2) can be represented as 0110
& (2,3) can be represented as 1011

As the definition of bdds suggests for the set of 4 bits which will have 16 entries, 0001, 0110,1011 will map to 1(true).For all other entries it would map to 0(false).
I have used BuDDy package to code the BDDs and DOT package to visualize it. Let's now take a brief look at how to program the relational bdds using finite domain blocks or fdds of BuDDy package.
1. The first step is to declare the domain sizes and the number of domains. In our example we need 3 domains which are respectively Var , ProgramPoint1, ProgramPoint2. ProgramPoint1 and ProgramPoint2 are the same domains of labels but we need them since the cfg bdd represents the relation ProgramPoint X ProgramPoint and to represent it we need two domains of labels. Now globalcfg can be said to depicting the relation ProgramPoint1 X ProgramPoint2. The sizes of the domain are important. To determine the size look at the maximum entry of the domain. In our example program max entry in Var domain is 2 so it can be represented by 2 bits since 2 bits can represent upto 3 i.e. 11(in binary).Similarly maximum entry at ProgramPoint1 and ProgramPoint2 is 10 hence it can be represented by 4 binary bits. The domain sizes will be :

Var	:	2 bits
ProgramPoint1	:	4 bits
ProgramPoint2	:	4 bits

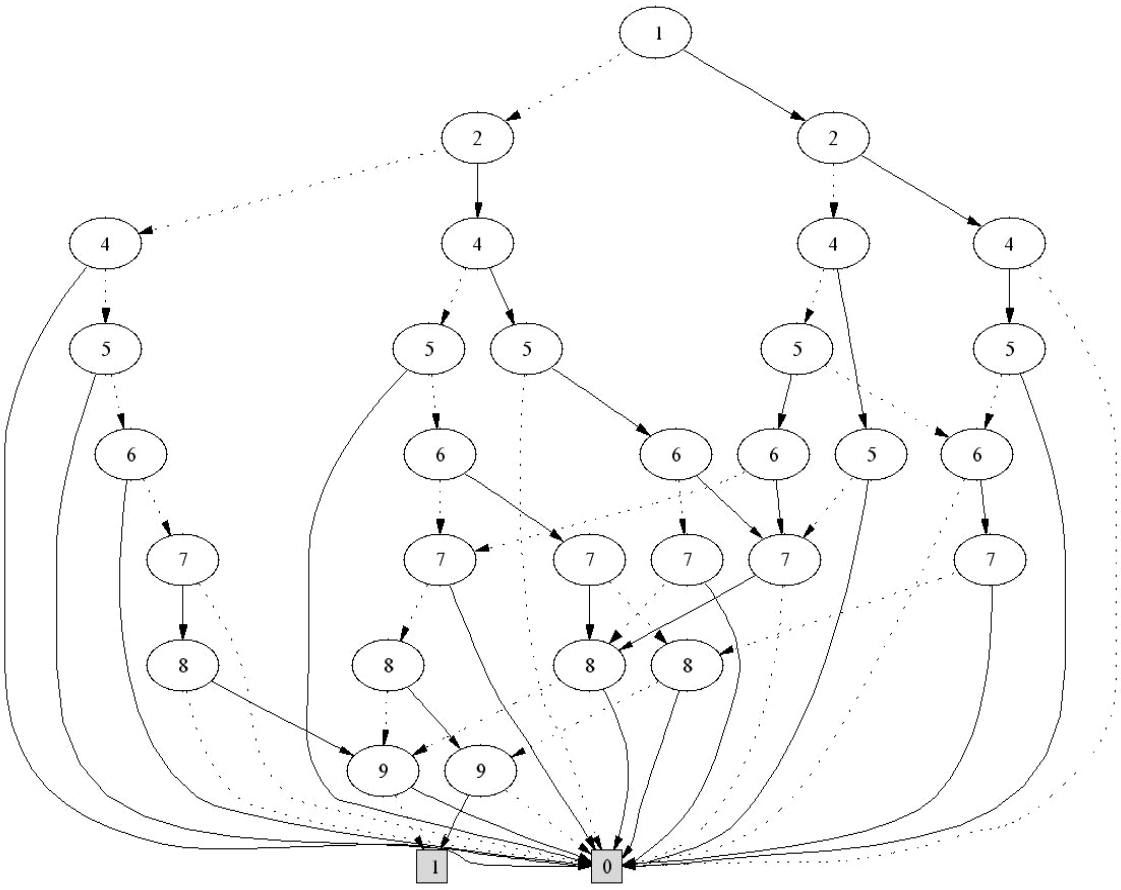
2. Next is to initialize the domains with the respective sizes. A unique integer value would be assigned to each domain which can be used later to do operations on that domain. In our example program Var is assigned an unique integer 0, ProgramPoint1 and ProgramPoint2 are assigned unique integer value 1 and 2.
3. The initialization code is provided in globalSysInit() function in Globalsolver.c of the source code. fdd_extdomain functions declares the fdd domain. Rest operations are same as described above. Take a look at how the function works in the introduction of BuDDy package.
4.After initialization the next step is the construction of the relation bdds for globalkill, globalgen and globalcfg . To construct one tuple say (1,2) in globalcfg take one bdd identified with unique id 1 from ProgramPoint1 domain and take another with unique id 2 from ProgramPoint2 domain and find their logical and. The logical and will form the bdd for the tuple (1,2) as the encoding described above. The default encoding of BuDDy is a bit different from the encoding described above. BuDDy forms the encoding as following:
a) (1,2) is in gloabalcfg and globalcfg is relation of form ProgramPoint1 X ProgramPoint2 which are domains each of size 4. So size of the encoding of the tuple will be 8 bits.

	x_4	x_3	x_2	x_1
1 in 4 bits can be represented as	0	0	0	1

	y_4	y_3	y_2	y_1
2 in 4 bits can be represented as	0	0	1	0

Default encoding of BuDDy construct the tuple as $y_4x_4y_3x_3y_2x_2y_1x_1$ which will be 00001001. Hence in globalcfg BDD, 00001001 will result in a output true.
b) Suppose we have to construct the set {(0,1), (1,2), (2,3)} then each of the set elements (0,1) , (1,2) and (2,3) are constructed in the above mentioned way and then the union of all of the elements will be taken.
Hence by this way we can make the relation bdds for the three relations mentioned above. On next few pages I will present the bdds for all the relation and take one example in one case to justify that the tuple is indeed in the relation bdd.
BDDs for the respective relations .Image drawn using DOT software from AT&T which is freely available on internet for educational and research purpose:

1.globalcfg bdd:

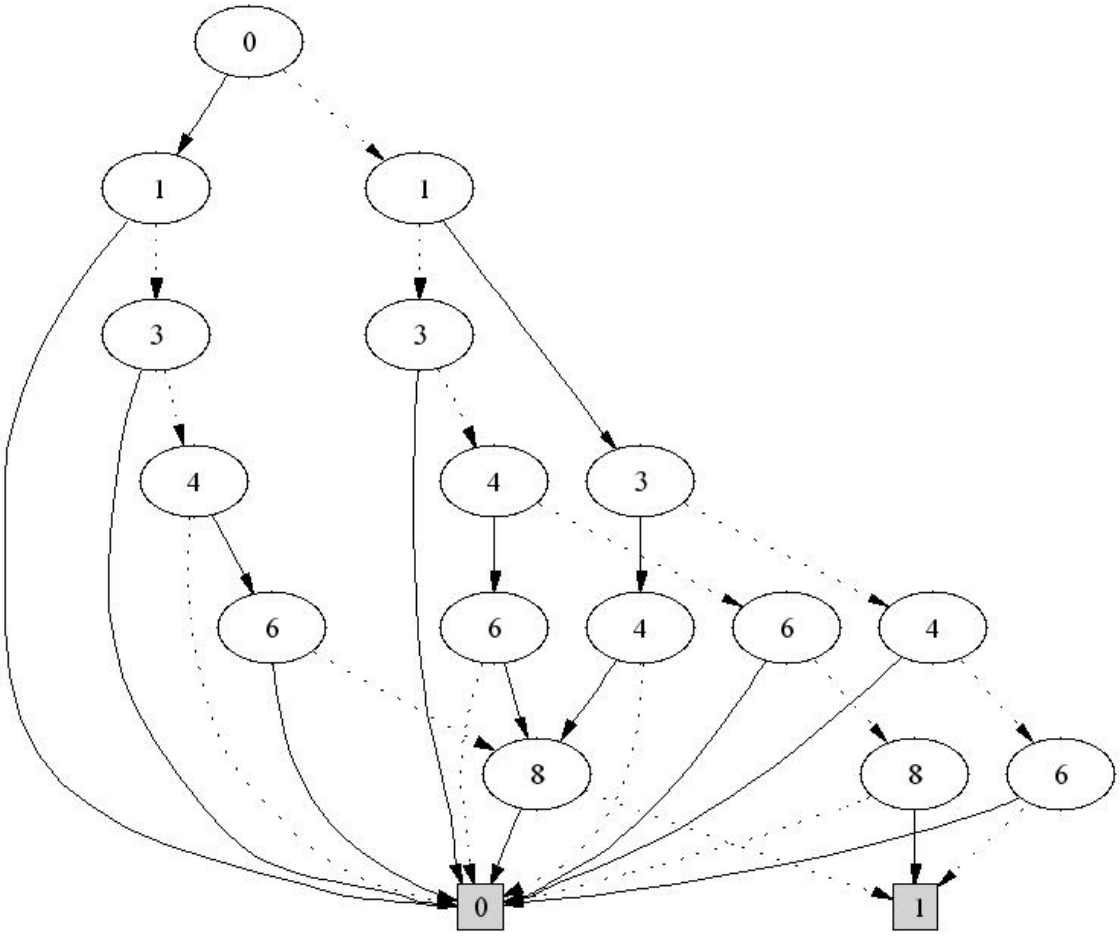


Let's take one tuple of control flow graph and justify that it's indeed true. One tuple in CFG should be (8,9). 8 can be encoded in 4 bits as 1000 and 9 can be encoded as 1001.Using the default encoding of BuDDy 11000010 should be true. Starting from node 1 taking the appropriate values we get :

Source Node	Truth value	Destination Node
1	0	2
2	1	4
4	0	5
5	0	6
6	0	7
7	0	8
8	1	9
9	1	leaf node(true)

Hence the tuple (8,9) is valid in the globalcfg bdd. The LSB of the encoded relation should be the starting point of check and ends at the MSB. Important point is that while checking don't care X can come at any point not only at the MSB. For example while checking if a particular node is not present a X can be added there. An encoding 11000X10 can give two different tuples 11000010 and 11000100. To find the truth assignments a function called bdd_allsat can be used which prints all the satisfying encodings of the BDD. Take a look at the source code to get more details.

2. globalkill bdd:

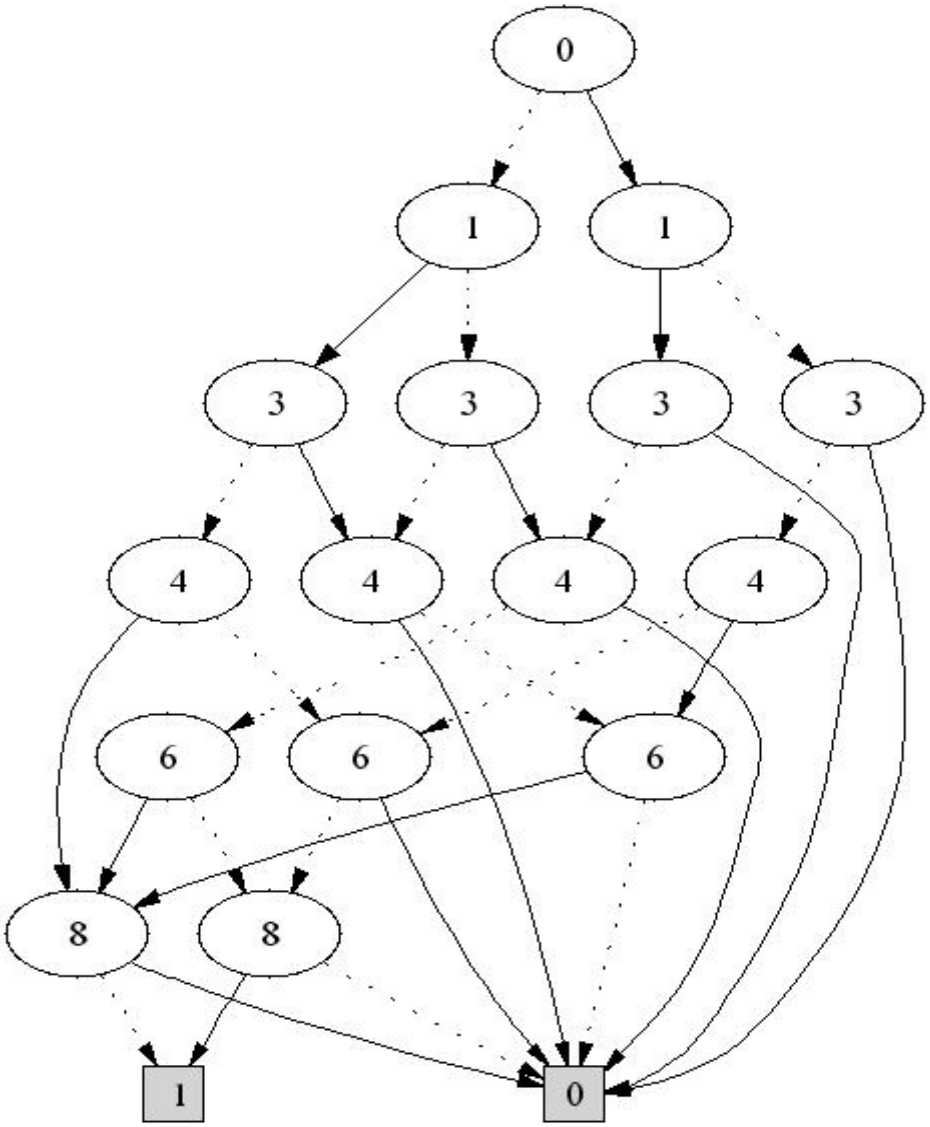


As we have done in globalcfg let's take a tuple and check whether the tuple exists in killcfg or not. The readers can take some other example of tuple and check it on their own. (2,7) is a tuple that should be in globalkill bdd where 2 comes from Var domain and 7 comes from ProgramPoint domain. ProgramPoint domain has size 4 bits while Var is of size 2 bits. So the encoding of 2 will be 10 and 7 can be encoded as 0111. The resultant entry will be 011110. Starting from LSB 0 from the node 0:

Source Node	Truth value	Destination Node
0	0	1
1	1	3
3	1	4
4	1X	8
8	0	leaf node(true)

From this example it's clear that don't care can be inserted at any point where any node is missing. At the path 01X110 node 6 is don't care. Hence both 010110 and 011110 will give true values.

3. globalgen bdd:



Readers please take a gen tuple and check whether it's coming to true or not.

Now I have three BDDs which capture globally the kill information , gen information and the control flow graph. Now let's take a look at the solver's algorithm which uses these BDDs and finds the solution to the live variable equations globally:

1. Start with two bdds solver and tempsolver initially both assigned to bdd_false() which is a function in BuDDy that returns false BDD.
2. Now calculate tempsolver as following:
   tempsolver = (relational_product(globalcfg,solver) – globalkill) U (globalgen)
3. After that check if tempsolver is equal to solver or not. If tempsolver is equal to solver that means we have found a fixed point and then print the truth values in solver and print out the solver bdd.
4. If tempsolver is n't equal to solver then equate solver to tempsolver and iterate the process again.
5. There will be fixed point in the lattice so the termination of the algorithm will take place.

Validity of the equation:

1. Take a look at the live variable equation of LVexit once again.

$$LVexit(l) = \emptyset \quad \text{if } l \in final(S)$$
$$\cup \{ LVentry(m) \mid (l,m) \in flow(S) \}$$

U {LVentry(m) | (l,m) ϵ flow(S)}will be captured by the relational product of solver and the control flow graph. Hence the relational product captures the LVexit information globally. Relational product is defined in BuDDy and described in detail in points-to analysis theory.

2. Now take a look at the LVentry equation.

LVentry(l) = (LVexit(l) \ kill (l)) U gen(l)

This information is captured by the remaining equation which is

(rel_prod(globalcfg,solver)-globalkill)-globalgen

Therefore it captures the LVentry equation.

There are some other analysis that needs to capture the intersection of equations i.e. ∩ANAentry(m).To capture the intersection the following formula should be used:

~(relational_product(globalcfg , ~solver))

where ~ is the negation operator in BDDs. It directs all the true edges to false and vice-versa.

Advantages of the algorithm:

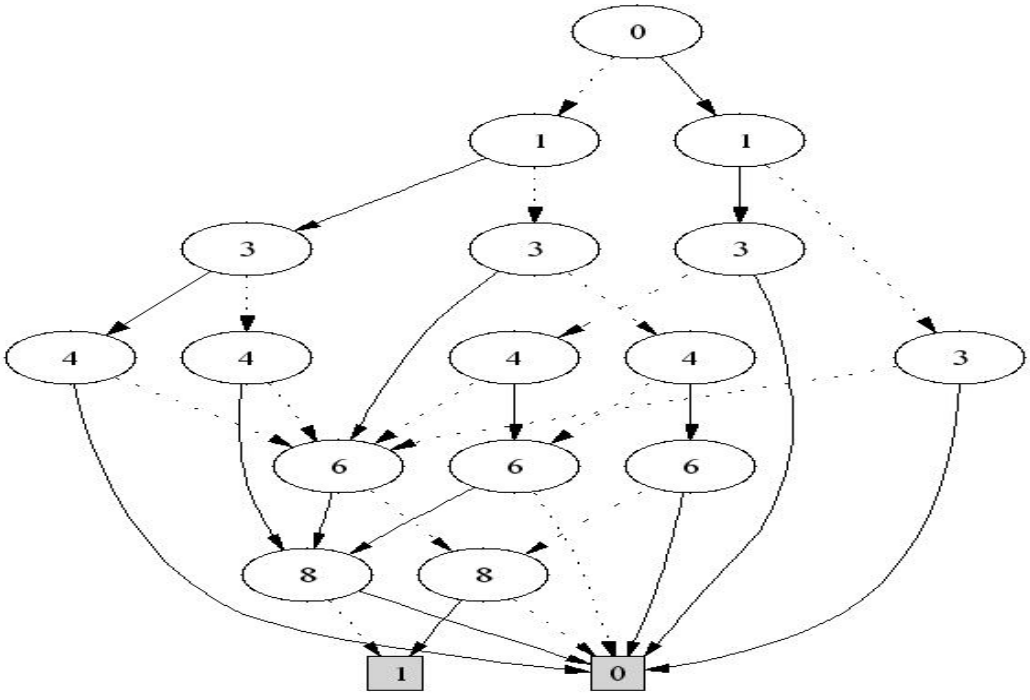
1. This algorithm is not tested on benchmarks. So at this point I can't firmly say that it would be efficient compared to traditional worklist algorithm but it should work faster on large benchmarks compared to worklist since it uses efficient data structures which are BDDs.
2. It's a new approach and it can be extended to several other analyses specially in inter-procedural and alias analysis.

Disadvantages of the algorithm:

1. Number of iterations at the worst case would be the same as worklist algorithm. So on the front of number of iterations it's not of much help.

Solver bdd for our example program: The tuples corresponding to the solver bdd which leads to true are also provided with it.

Solver bdd :



The tuples that satisfy the solver are the following:

(2,0)

(3,0)(3,1)
(4,0)(4,1)(4,2)
(5,1)(5,2)
(6,1)
(7,0)(7,1)
(8,1)(8,2)
(9,0)(9,1)

The tuples are of the form (ProgramPoint,Var).The computation by worklist algorithm also yields the same results. Hence this is indeed the LVentry set of the example program. It completes the complete description of all the phases and the work done during each phase along with the results. For more details take a look at the source code.

Future Works: 1.The solver should be tested on benchmarks to compare the increase in efficiency from traditional approach of finding the solution of live variable equations using worklist algorithm.

2. Development of a bdd framework to do other instances of data flow analysis like REACHING DEFS, VERY BUSY EXPRESSIONS, etc.

3. Application of BDDs to Alias Analysis. Alias is a reflexive, transitive and symmetric relation. BDDs can represent relations and are efficient in finding the operations on relations. Capturing the global alias information will be my next target.

References:

1. Program Analysis Book, Springer publication.
2. Points-to Analysis using BDDs *by* Marc Berndl, Ond¡rej Lhot´ak, Feng Qian, Laurie Hendren and Navindra Umanee , Sable Research Group, School of Computer Science McGill University, Montreal, Quebec, Canada
3. Anderson's Phd thesis on Program Analysis.
4. Compilers Principles, Techniques and Tools by Alfred V.Aho, Ravi Sethi , Jeffrey D. Ullman
5. C-- language developed as part of compiler course at IIT Kharagpur, Details can be found on the following website: http://www.facweb.iitkgp.ernet.in/~rkumar/compiler
6. An Introduction to Binary Decision Diagrams by Henrik Reif Andersen
7. BuDDy: Binary Decision Diagram Package Release 2.2 by Jorn Lind-Neilsen. IT- University of Copenhegan(ITU).
8. Appendix A1, Program Analysis Book , Springer publication